# Compiler 2016

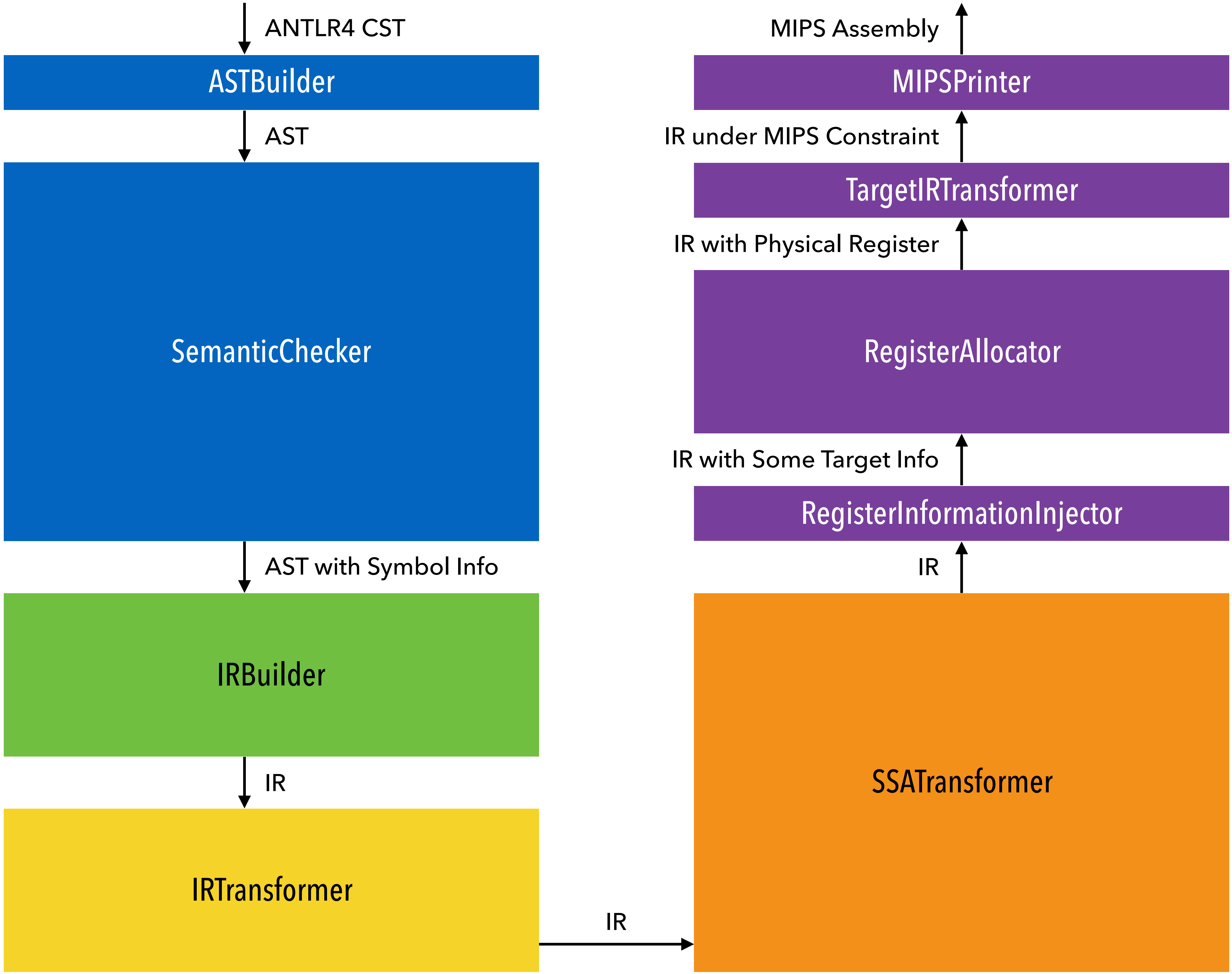
What I learnt & What I implemented
during the period of the compiler course

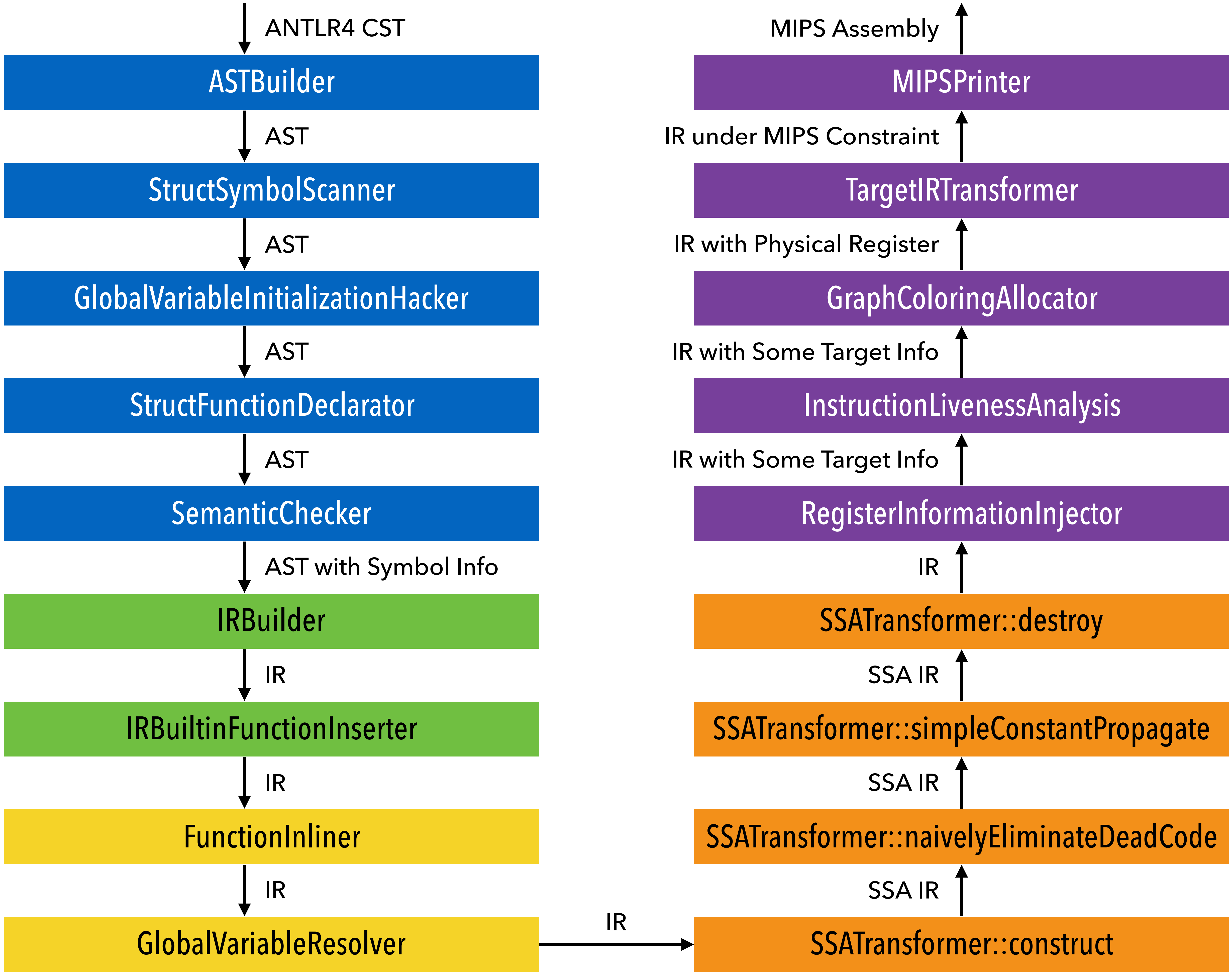
Lequn Chen
May 2016

# Overview

# Compiler

- Modularization
- Low coupling
- Multiple passes


ANTLR4 CST
ASTBuilder
AST
SemanticChecker
AST with Symbol Info
IRBuilder
IR
IRTransformer
IR
SSATransformer
IR
RegisterInformationInjector
IR with Some Target Info
RegisterAllocator
IR with Physical Register
TargetIRTransformer
IR under MIPS Constraint
MIPSPrinter
MIPS Assembly


ANTLR4 CST
ASTBuilder
AST
StructSymbolScanner
AST
GlobalVariableInitializationHacker
AST
StructFunctionDeclarator
AST
SemanticChecker
AST with Symbol Info
IRBuilder
IR
IRBuiltinFunctionInserter
IR
FunctionInliner
IR
GlobalVariableResolver
IR
SSATransformer::construct
SSA IR
SSATransformer::naivelyEliminateDeadCode
SSA IR
SSATransformer::simpleConstantPropagate
SSA IR
SSATransformer::destroy
IR
RegisterInformationInjector
IR with Some Target Info
InstructionLivenessAnalysis
IR with Some Target Info
GraphColoringAllocator
IR with Physical Register
TargetIRTransformer
IR under MIPS Constraint
MIPSPrinter
MIPS Assembly

# Package Hierarchy

- `FrontEnd:` CST/AST related transforms
- `BackEnd:` IR and backend related transforms
- `MIPS:` MIPS related transforms

- `AST:` Data structures that hold AST
- `IR:` Data structures that hold IR
- `Symbol:` Data structures that hold symbol table

- `Parser:` ANTLR4 Generated Lexer/Parser

# ANTLR 4

# A Powerful Tool

- I wrote 2 articles about it
    - https://abcdabcd987.com/notes-on-antlr4/
    - https://abcdabcd987.com/using-antlr4/
- Reference:
    - The Definitive ANTLR 4 Reference

# Official Examples

- https://github.com/antlr/grammars-v4
- See how to write grammars
- Copy & Paste common parts (lexer fragments…)
- These two helped me a lot:
  - https://github.com/antlr/grammars-v4/blob/master/c/C.g4
  - https://github.com/antlr/grammars-v4/blob/master/java/Java.g4

# Deal with Expression

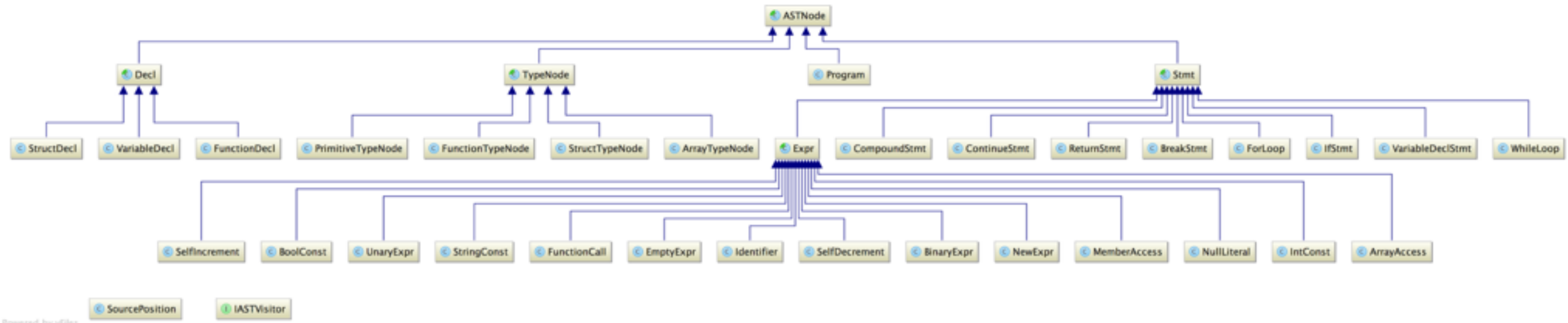
```
expression
    :   expression op=('++' | '--')                    # PostfixIncDec    // Precedence 1
    |   expression '(' parameterList? ')'              # FunctionCall
    |   expression '[' expression ']'                  # Subscript
    |   expression '.' Identifier                      # MemberAccess

    |   <assoc=right> op=('++'|'--') expression        # UnaryExpr        // Precedence 2
    |   <assoc=right> op=('+' | '-') expression        # UnaryExpr
    |   <assoc=right> op=('!' | '~') expression        # UnaryExpr
    |   <assoc=right> 'new' creator                    # New

    |   expression op=('*' | '/' | '%') expression     # BinaryExpr       // Precedence 3
    |   expression op=('+' | '-') expression           # BinaryExpr       // Precedence 4
    |   expression op=('<<'|'>>') expression           # BinaryExpr       // Precedence 5
    |   expression op=('<' | '>') expression           # BinaryExpr       // Precedence 6
    |   expression op=('<='|'>=') expression           # BinaryExpr
    |   expression op=('=='|'!=') expression           # BinaryExpr       // Precedence 7
    |   expression op='&' expression                   # BinaryExpr       // Precedence 8
    |   expression op='^' expression                   # BinaryExpr       // Precedence 9
    |   expression op='|' expression                   # BinaryExpr       // Precedence 10
    |   expression op='&&' expression                  # BinaryExpr       // Precedence 11
    |   expression op='||' expression                  # BinaryExpr       // Precedence 12

    |   <assoc=right> expression op='=' expression     # BinaryExpr       // Precedence 14

    |   Identifier                                     # Identifier
    |   constant                                       # Literal
    |   '(' expression ')'                             # SubExpression
    ;
//------ Reference: http://en.cppreference.com/w/cpp/language/operator_precedence
```

# Deal with Expression

```
@Override
public void exitBinaryExpr(MillParser.BinaryExprContext ctx) {
    BinaryExpr.BinaryOp op;
    switch (ctx.op.getType()) {
        case MillParser.Star:   op = BinaryExpr.BinaryOp.MUL;         break;
        case MillParser.Div:    op = BinaryExpr.BinaryOp.DIV;         break;
        case MillParser.Mod:    op = BinaryExpr.BinaryOp.MOD;         break;
        case MillParser.Plus:   op = BinaryExpr.BinaryOp.ADD;         break;
        ......
        case MillParser.And:    op = BinaryExpr.BinaryOp.BITWISE_AND; break;
        case MillParser.Caret:  op = BinaryExpr.BinaryOp.XOR;         break;
        case MillParser.Or:     op = BinaryExpr.BinaryOp.BITWISE_OR;  break;
        case MillParser.AndAnd: op = BinaryExpr.BinaryOp.LOGICAL_AND; break;
        case MillParser.OrOr:   op = BinaryExpr.BinaryOp.LOGICAL_OR;  break;
        case MillParser.Assign: op = BinaryExpr.BinaryOp.ASSIGN;      break;
        default: throw new RuntimeException("Unknown binary operator.");
    }
    map.put(ctx, new BinaryExpr(
            op,
            (Expr)map.get(ctx.expression(0)),
            (Expr)map.get(ctx.expression(1)),
            new SourcePosition(ctx.op),
            new SourcePosition(ctx.expression(0)),
            new SourcePosition(ctx.expression(1))
    ));
}
```

# Abstract Syntax Tree

# AST Hierarchy




- (nothing special)
- Build via ANTLR4 `ParseTreeListener`
- Access via `IASTVisitor`

# Semantic Check

- Done in 3 passes (since need to look afterward)
- `StructSymbolScanner`
  - Scan class name
- `StructFunctionDeclarator`
  - Scan function signature
- `SemanticChecker`
  - Build symbol table
  - Semantic Check

# Symbol Table

# Symbol Table

- A good design to refer to
  - Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*.
    - Chapter 6
    - Chapter 7

# Intermediate Representation

# What's IR?

```
define i32 @arith(i32 %x, i32 %y, i32 %z) #0 {
entry:
  %add = add nsw i32 %y, %x
  %mul = mul nsw i32 %z, 48
  %and = and i32 %add, 65535
  %mul1 = mul nsw i32 %mul, %and
  ret i32 %mul1
}
```

Is this IR?

```
class BinaryOperation { … }
class Load { … }
…
```

Is this IR?

```
myprog.bc
0101001010101010100010010000101011О…
```

Is this IR?

# What's IR?

```
define i32 @arith(i32 %x, i32 %y, i32 %z) #0 {
entry:
  %add = add nsw i32 %y, %x
  %mul = mul nsw i32 %z, 48
  %and = and i32 %add, 65535
  %mul1 = mul nsw i32 %mul, %and
  ret i32 %mul1
}
```

Text Form

```
class BinaryOperation { … }
class Load { … }
…
```

Data Structures in Memory

```
myprog.bc
0101001010101010100010010000101 0110…
```

Binary Form

# What's IR?

- Data structure in memory (i.e. Java classes)
  - Of course we need to focus on this
- Text form
  - For debugging…
  - Needed if IR flows through several different program (for example, run on a virtual machine)
- Binary form
- (Though "What is IR?" seems a stupid question, it bothered me a lot at first)

# Where to Start?

- IR design is closely related to
  - Source language
  - Target machine
  - Transforms / Analysis
- Though low coupling, IR designing deserves carefully thinking
- I (not my compiler) was much influenced by LLVM

# Type System

- Should IR contain type information? To what extent?
- LLVM
  - `int: i1, i8, i32, i65536, …, iN`
  - `pointer: <type> *`
  - `array: [3 x [4 x i32]]`
  - `structure: { float, i32 (i32) * }`
  - `getelementptr`

# Type System

- LLVM: almost keep everything!

```c
struct RT {
  char A;
  int B[10][20];
  char C;
};
struct ST {
  int X;
  double Y;
  struct RT Z;
};

int *foo(struct ST *s) {
  return &s[1].Z.B[5][13];
}
```

```llvm
%struct.RT = type { i8, [10 x [20 x i32]], i8 }
%struct.ST = type { i32, double, %struct.RT }

define i32* @foo(%struct.ST* %s) {
entry:
  %arrayidx = getelementptr inbounds %struct.ST,
%struct.ST* %s, i64 1, i32 2, i32 1, i64 5, i64 13
  ret i32* %arrayidx
}
```

# Type System

- LLVM
  - almost keep everything
- However, EAC, the Dragon Book and the Tiger Book
  - They pretend that "Type System? WTF are you talking about?"
  - It seems that everything is of a general register's size.
- What's wrong here?

# Multiple Level

- A compiler can use more than one IR, and of course, there are more than one level.

- HIR/MIR: Carry more information. May have type system similar to the source language. Higher level analysis & transforms can be performed on. (Alias analysis works better with type knowledge)
  - `point1.x => (LoadField point1 “x”)`

- LIR: Closer to the target machine. Don’t have much type information (General/FP Reg). Focus on code generation.
  - `point1.x => (LoadMem (Mem baseAddr 4))`

# Multiple Level

- A compiler can use more than one IR, and of course, there are more than one level.
- LLVM: Low Level Virtual Machine
- Actually, its level is not that low.
- And it happens that LLVM use a single representation.

# Design: Explicit Variable?

```
class Quad {
    OpCode   op;
    Variable src1;
    Variable src2;
    Variable dest;
}
```

vs

```
class Quad {
    OpCode op;
    Quad   src1;
    Quad   src2;
}
```

- Explicit variable:
  - Simple, Straightforward
  - Data dependence information lost
- Implicit variable:
  - Quad itself stands for value
  - Data dependence is explicit
  - Optimizing Compilers for Modern Architectures, Rand Allen and Ken

Ref: https://www.zhihu.com/question/33518780/answer/56731699 (for the following several design question)

# Design: Explicit Variable?

```
class Quad {
    OpCode   op;
    Variable src1;
    Variable src2;
    Variable dest;
}
```

vs

```
class Quad {
    OpCode op;
    Quad   src1;
    Quad   src2;
}
```

- I chose the implicit one.
- After I finished `IRBuilder`, I found it too heavy.
- In addition, I didn't think I would dig into data dependence based optimization.
- I rewrote using the explicit one.

Ref: https://www.zhihu.com/question/23261395/answer/42426957

# Design: Handwritten vs List<T>

- List<T>
  - No need to write by yourself
  - Hard to insert / remove / iterate
  - Cannot modify while iterating
- Handwritten linked list
  - Should not be a problem for a former OIer / present CS student
- Transforms make modifications to IR so often! You'll regret it if you are lazy when building IR.

# Design: Structure

- Tree (the Tiger Book)
- Linear (the Dragon Book)
- Control Flow Graph
  - Nodes are basic blocks
  - Linear inside each node (or DAG if explicit data dependence)

# Design: Explicit CFG?

- Implicit CFG:
  - Build up linear IR, with `class Label { String name; … }`
  - Jump target is a label
  - Scan IR and build CFG
- Explicit CFG:
  - Construct CFG while building IR
  - Jump target is `class BasicBlock { Quad head; … }`
  - Fall-through is not permitted
- I preferred the second one. It seemed more clear for me. (I can't understand what the Tiger Book says.) I'm also influenced by the LLVM Kaleidoscope.

Ref: http://llvm.org/docs/tutorial/LangImpl5.html

# Design: Explicit CFG?

```
Value *IfExprAST::codegen() {
  Value *CondV = Cond->codegen();
  if (!CondV)
    return nullptr;

  // Convert condition to a bool by comparing equal to 0.0.
  CondV = Builder.CreateFCmpONE(
      CondV, ConstantFP::get(TheContext, APFloat(0.0)), "ifcond");

  Function *TheFunction = Builder.GetInsertBlock()->getParent();

  // Create blocks for the then and else cases.
  // Insert the 'then' block at the end of the function.
  BasicBlock *ThenBB = BasicBlock::Create(TheContext, "then", TheFunction);
  BasicBlock *ElseBB = BasicBlock::Create(TheContext, "else");
  BasicBlock *MergeBB = BasicBlock::Create(TheContext, "ifcont");

  Builder.CreateCondBr(CondV, ThenBB, ElseBB);

  // Emit then value.
  Builder.SetInsertPoint(ThenBB);
  Value *ThenV = Then->codegen();
  if (!ThenV)
    return nullptr;
  // ...
}
```

LLVM Kaleidoscope

# Design: Explicit CFG?

```
@Override
public void visit(IfStmt node) {
    BasicBlock BBTrue = new BasicBlock(curFunction, "if_true");
    BasicBlock BBFalse = node.otherwise != null ? new BasicBlock(curFunction, "if_false") : null;
    BasicBlock BBMerge = new BasicBlock(curFunction, "if_merge");

    // branch instruction should be added by logical expression
    node.cond.ifTrue = BBTrue;
    node.cond.ifFalse = node.otherwise != null ? BBFalse : BBMerge;
    visit(node.cond);

    // generate then
    curBB = BBTrue;
    visit(node.then);
    if (!curBB.isEnded()) curBB.end(new Jump(curBB, BBMerge));

    // generate else
    if (node.otherwise != null) {
        curBB = BBFalse;
        visit(node.otherwise);
    }
    if (BBFalse != null && !curBB.isEnded()) curBB.end(new Jump(curBB, BBMerge));

    // merge
    curBB = BBMerge;
}
```

My Compiler

# Design: Explicit CFG?

```java
@Override
public void visit(IfStmt node) {
    BasicBlock BBTrue = new BasicBlock(curFunction, "if_true");
    BasicBlock BBFalse = node.otherwise != null ? new BasicBlock(curFunction, "if_false") : null;
    BasicBlock BBMerge = new BasicBlock(curFunction, "if_merge");

    // branch instruction should be added by logical expression
    node.cond.ifTrue = BBTrue;
    node.cond.ifFalse = node.otherwise != null ? BBFalse : BBMerge;
    visit(node.cond);

    // generate then
    curBB = BBTrue;
    visit(node.then);
    if (!curBB.isEnded()) curBB.end(new Jump(curBB, BBMerge));

    // generate else
    if (node.otherwise != null) {
        curBB = BBFalse;
        visit(node.otherwise);
    }
    if (BBFalse != null && !curBB.isEnded()) curBB.end(new Jump(curBB, BBMerge));

    // merge
    curBB = BBMerge;
}
```

I believe this is much more easy to understand than the Tiger Book's solution

# Short-Circuit Evaluation

- I thought about it for quite a long time
  - Control vs. Value; Various Situation;
  - `if    (!((a != 0 && b/a == 2) || !(b == 0 || c/b == 4) && x)))`
  - `for (; !((a != 0 && b/a == 2) || !(b == 0 || c/b == 4) && x)); )`
  - `t =    !((a != 0 && b/a == 2) || !(b == 0 || c/b == 4) && x))`
- I got inspired when I was scanning the Dragon Book

# Short-Circuit Evaluation

```
B -> B1 || B2      B1.ifTrue  = B.ifTrue
                   B1.ifFalse = new BasicBlock(“lhsFalse”)
                   B2.ifTrue  = B.ifTrue
                   B2.ifFalse = B.ifFalse

B -> !B1           B1.ifTrue  = B.ifFalse
                   B1.ifFalse = B.ifTrue

S -> if (B) S1     B.ifTrue   = new BasicBlock(“ifTrue”)
     else   S2     B.ifFalse  = new BasicBlock(“ifFalse”)
```

- Push down information!
- If implemented correctly, this should be efficient!
- I never struggle with *Testcase expr*

# Design: Memory Model?

- Memory-to-Memory
- Register-to-Register:
  - Unlimited virtual register
  - Easy to understand
  - The target machine is MIPS, why not?

# Design: Stack, Heap?

- Since unlimited virtual register, keep most of things in virtual register.
- For stack space, use `class StackSlot { … }`, which will be replaced by `$sp` offset in `TargetIRTransformer`.
- For heap space, use `class HeapAllocate { int size; … }`, which will be replaced by MIPS system call `sbrk` in `RegisterInformationInjector`.
- No explicit `class FrameManager { … }` or something

# Design: Function?

- Should the "function" and "function call" concept present in IR?
- I strongly support it
  - Simplify things
  - Function call doesn't split basic block
  - In optimization's language, "global" means inside a function, not the whole program.
  - (Influenced by LLVM)

# Debugging

- I printed my IR in LLVM’s format and run
  - Painful!
  - No direct memory arithmetic!
- I wrote my own interpreter
  - https://github.com/abcdabcd987/LLIRInterpreter
  - Life is much more easier!

# LLIRInterpreter

```
func main {
%main.entry:
    $n.1  = move 10
    $f0.1 = move 0
    $f1.1 = move 1
    $i.1  = move 1
    jump %for_cond

%for_cond:
    $f2.1 = phi %for_step $f2.2  %main.entry undef
    $f1.2 = phi %for_step $f1.3  %main.entry $f1.1
    $i.2  = phi %for_step $i.3   %main.entry $i.1
    $f0.2 = phi %for_step $f0.3  %main.entry $f0.1
    $t.1  = slt $i.2 $n.1
    br $t.1 %for_loop %for_after

%for_loop:
    $t_2.1 = add $f0.2 $f1.2
    $f2.2  = move $t_2.1
    $f0.3  = move $f1.2
    $f1.3  = move $f2.2
    jump %for_step

%for_step:
    $i.3   = add $i.2 1
    jump %for_cond

%for_after:
    ret $f2.2
}
```

Calculate Fibonacci[n]

# Global Variable

# Global Variable

- Global variables are escaped variable
- They must be kept in memory

# Initialization

- Int / String:
  - store in .data segment
- Struct / Array:
  - store pointer in .data segment
  - do initialization before entering main

# Read

- Don't load every time needed.

```
int n;
void foo() {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            // … do some stuffs
}
```

- Load all needed to virtual register at the entry of the function.

```
%foo.entry:
    $n = load word @n
    // …
```

# Write

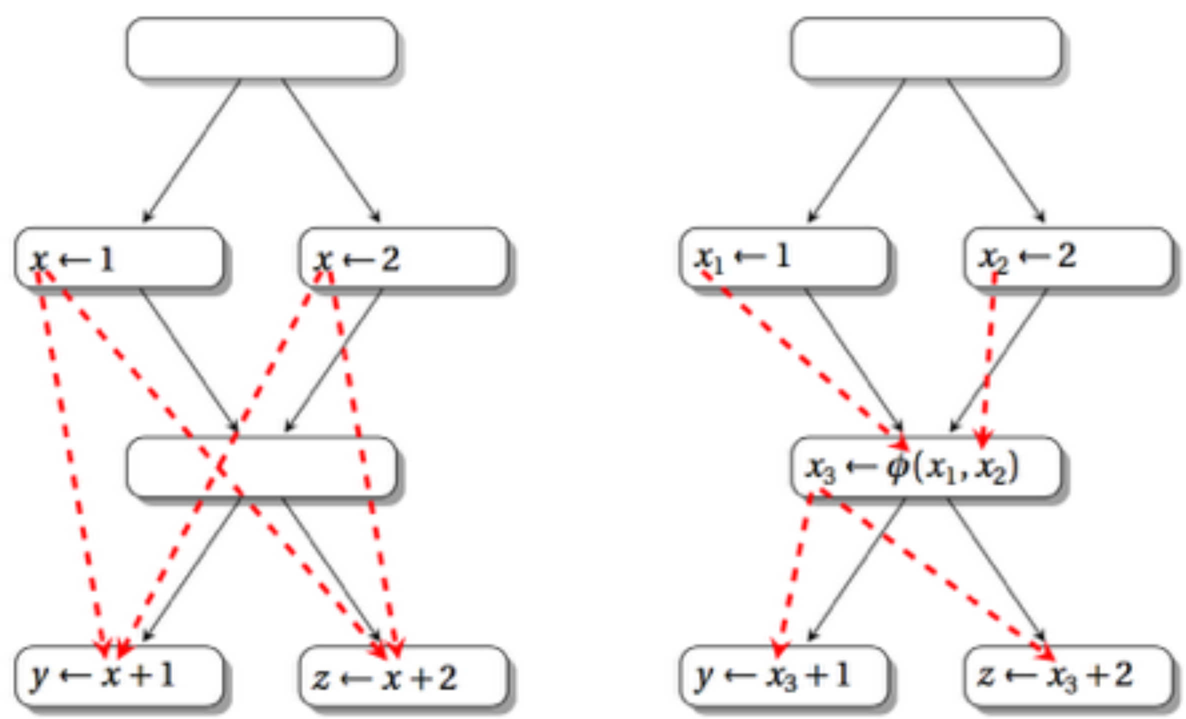
- Postpone until
  - function call
  - function exit

```
%foo.entry:
    $n = load word @n
    $n = add $n $n
    store word @n $n
    call bar
    $n = add $n 1
    $n = mul $n 2
    store word @n $n
    ret
```

# Single Static Assignment Form

# Why SSA?

- Single assignment: clearer def-use chain
- Simplify and strengthen analysis and transforms




**(a)** non-SSA **(b)** SSA form

**Fig. 2.1** Def-use chains (dashed) for non-SSA form and its corresponding SSA form program.

Ref: Chapter 2.1, The SSABook

# LLVM: alloca

```c
int main() {
    int x = 10;
    int y = 20;
    int z;
    if (x < y) z = x;
    else z = y;
    return z;
}
```

```llvm
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %x = alloca i32, align 4
  %y = alloca i32, align 4
  %z = alloca i32, align 4
  store i32 0, i32* %1
  store i32 10, i32* %x, align 4
  store i32 20, i32* %y, align 4
  %2 = load i32* %x, align 4
  %3 = load i32* %y, align 4
  %4 = icmp slt i32 %2, %3
  br i1 %4, label %5, label %7

; <label>:5                ; preds = %0
  %6 = load i32* %x, align 4
  store i32 %6, i32* %z, align 4
  br label %9

; <label>:7                ; preds = %0
  %8 = load i32* %y, align 4
  store i32 %8, i32* %z, align 4
  br label %9

; <label>:9                ; preds = %7, %5
  %10 = load i32* %z, align 4
  ret i32 %10
}
```

# LLVM: alloca

- alloca: get space from stack, return pointer
- store: initial value
- load: load to virtual reg
- This is not actually in SSA form.

```
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %x = alloca i32, align 4
  %y = alloca i32, align 4
  %z = alloca i32, align 4
  store i32 0, i32* %1
  store i32 10, i32* %x, align 4
  store i32 20, i32* %y, align 4
  %2 = load i32* %x, align 4
  %3 = load i32* %y, align 4
  %4 = icmp slt i32 %2, %3
  br i1 %4, label %5, label %7

; <label>:5                    ; preds = %0
  %6 = load i32* %x, align 4
  store i32 %6, i32* %z, align 4
  br label %9

; <label>:7                    ; preds = %0
  %8 = load i32* %y, align 4
  store i32 %8, i32* %z, align 4
  br label %9

; <label>:9                    ; preds = %7, %5
  %10 = load i32* %z, align 4
  ret i32 %10
}
```

# LLVM: mem2reg Pass

- LLVM goes into SSA after mem2reg pass
  - alloca/load/store are replaced

```
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %x = alloca i32, align 4
  %y = alloca i32, align 4
  %z = alloca i32, align 4
  store i32 0, i32* %1
  store i32 10, i32* %x, align 4
  store i32 20, i32* %y, align 4
  %2 = load i32* %x, align 4
  %3 = load i32* %y, align 4
  %4 = icmp slt i32 %2, %3
  br i1 %4, label %5, label %7

; <label>:5                ; preds = %0
  %6 = load i32* %x, align 4
  store i32 %6, i32* %z, align 4
  br label %9

; <label>:7                ; preds = %0
  %8 = load i32* %y, align 4
  store i32 %8, i32* %z, align 4
  br label %9

; <label>:9                ; preds = %7, %5
  %10 = load i32* %z, align 4
  ret i32 %10
}
```

```
define i32 @main() #0 {
  %1 = icmp slt i32 10, 20
  br i1 %1, label %2, label %3

; <label>:2    ; preds = %0
  br label %4

; <label>:3    ; preds = %0
  br label %4

; <label>:4    ; preds = %3, %2
  %z.0 = phi i32 [ 10, %2 ], [ 20, %3 ]
  ret i32 %z.0
}
```

# LLVM: mem2reg Pass

- alloca/load/store pattern: good for front-end author.
  - They don't need to generate SSA IR
  - Variable usage wraps by load/store is quite easy to generate.

- See source code:
  - http://llvm.org/docs/doxygen/html/Mem2Reg_8cpp_source.html
  - http://llvm.org/docs/doxygen/html/PromoteMemoryToRegister_8cpp_source.html

- Play with LLVM:
  - `clang -emit-llvm -S tmp.c`
  - `opt -mem2reg -S tmp.ll`

Ref: https://www.zhihu.com/question/41999500

# LLVM: mem2reg Pass

- I followed LLVM’s alloca/load/store pattern at first
- Few documents & references
- I gave it up and made virtual register writeable
- Then I went on the common road

# Dominance Tree/Frontier

- Thomas Lengauer and Robert Endre Tarjan. *A fast algorithm for finding dominators in a flowgraph*
  - $O(m\alpha(m,n))$
- EAC gives pseudocode. But I don't see how to get IDom from Dom set??
- My savior: K.D. Cooper, T. J. Harvey and K.Kennedy. *A simple, fast dominance algorithm*
  - O(n^2) but usually not that bad
  - Run faster than Lengauer-Tarjan's algorithm (the authors claimed)
  - Give pseudocode to calculate IDom set directly
- Iteration ends faster if run in reverse-post-order.

# SSA Construction

- EAC gives good explanation and also pseudocode.
- Algorithm 9.9
- Algorithm 9.12

# SSA Destruction

- The SSA Book gives good explanation and also pseudocode.
  - phi => parallel copy
- Algorithm 3.5: *Critical Edge Splitting Algorithm for making non-conventional SSA form conventional.*
- Algorithm 22.6: *Parallel copy sequentialization algorithm*.
- Also see: Benoit Boissinot, Alain Darte, Fabrice Rastello, Benoît Dupont de Dinechin, Christophe Guillon. *Revisiting Out-of-SSA Translation for Correctness, Code Quality, and Efficiency.*

# Some SSA Optimizations

- Strong ones:
  - Sparse Conditional Constant Propagation
  - GVN & GCM
  - …
- However, I only did trivial ones:
  - Naive Dead Code Elimination
  - Simple Constant Propagation
- To learn from big projects:
  - LLVM Scalar Transforms
  - QTDeclarative Compiler
  - Android Dalvik SSA Transforms

# Graph Coloring Reg Alloc

- Interference graph of live-ranges in SSA is chordal graph
  - Perfect graph coloring in O(|V|+|E|)
  - What about destruction phase?
    - Naive trick: keep a register not allocated to break parallel copy cycle
    - Other…

Ref: Chapter 2.4, The SSABook

# Linear Scan Reg Alloc

- M. Poletto and V. Sarkar. *Linear scan register allocation*.
  - Simple, straightforward, fast, not so satisfactory result
  - Based on life interval
- C. Wimmer and H. Mössenböck. *Optimized interval splitting in a linear scan register allocator*.
  - Add holes to interval, much better result (~75% graph coloring)
  - No concrete tutorial for interval building and resolving phase
- C. Wimmer and M. Franz. *Linear Scan Register Allocation on SSA Form*.
  - Pseudocode for interval building on SSA and SSA resolving
  - The order of basic block still have huge influence on result

PS: I saw the last paper on http://c9x.me/compile/bib/, and then found the other two from its references

# Linear Scan Reg Alloc

- I was following these papers at first (`regalloc_on_ssa` branch)
- Then, I found that block order matters.
  - Basic Block Reschedule
  - Loop Detection
  - …
- So, I gave up again. Back to graph coloring in non-SSA
- BTW, QTDeclarative implemented these papers
  - https://github.com/qtproject/qtdeclarative/blob/dev/src/qml/compiler/qv4ssa.cpp

PS: I searched "TryAllocateFreeReg" on Github and found QTDeclarative

# Optimization
# (Analysis & Transforms)

# SSA

- Without strong optimization, the result is quite disappointing.
- Negative result sometimes shows up.
- -7% ~ +16%

| | bulgarian | heapsort | horse | horse2 | horse3 | magic |
|---|---|---|---|---|---|---|
| **Limit** | 1500000 | 20000000 | 25000000 | 15000000 | 25000000 | 2000000 |
| **Original** | 793272 | 8710389 | 13373269 | 9470959 | 15216222 | 1549065 |
| **Original %** | 52.88% | 43.55% | 53.49% | 63.14% | 60.86% | 77.45% |
| **SSA** | 818151 | 8541923 | 12217366 | 7203394 | 14099845 | 1549394 |
| **+DCE** | 775052 | 8756811 | 11726688 | 7114878 | 14586323 | 1531605 |
| **+SCP+DCE** | 737866 | 8237036 | 11897680 | 7122020 | 13973512 | 1473229 |
| **Optimized %** | 49.19% | 41.19% | 47.59% | 47.48% | 55.89% | 73.66% |

# Function Inlining

- Highly Effective! up to ~ 40% improvement
- Can be extended to recursion unrolling.

| | hashmap | horse | horse3 |
|---|---|---|---|
| Limit | 550000 | 25000000 | 25000000 |
| Original | 248263 | 11911209 | 14027651 |
| Original % | 45.14% | 47.64% | 56.11% |
| Optimized | 171953 | 3914219 | 4642421 |
| Optimized % | 31.26% | 15.66% | 18.57% |

# Optimization
# (Code Generation)

# Self Move

- move $reg, $reg
- cooperate with register allocation
- help to remove some moves caused by SSA destruction
- 1% ~ 2% overall improvement

# Print / Println

- Reasonable and highly effective! up to ~50% improvement
  - `print(str1 + str2)    => print(str1); print(str2);`
  - `print(toString(ival)) => printInt(ival);`
  - Rewrite recursively when building IR

| | bulgarian | hanoi | hashmap | spill2 |
|---|---|---|---|---|
| Limit | 1500000 | 450000 | 550000 | 100000 |
| Original | 1403790 | 401672 | 630450 | Can't Load |
| Original % | 93.59% | 89.26% | 114.63% | |
| Optimized | 798373 | 73196 | 306164 | 18966 |
| Optimized % | 53.22% | 16.27% | 55.67% | 18.97% |

# Saving Register

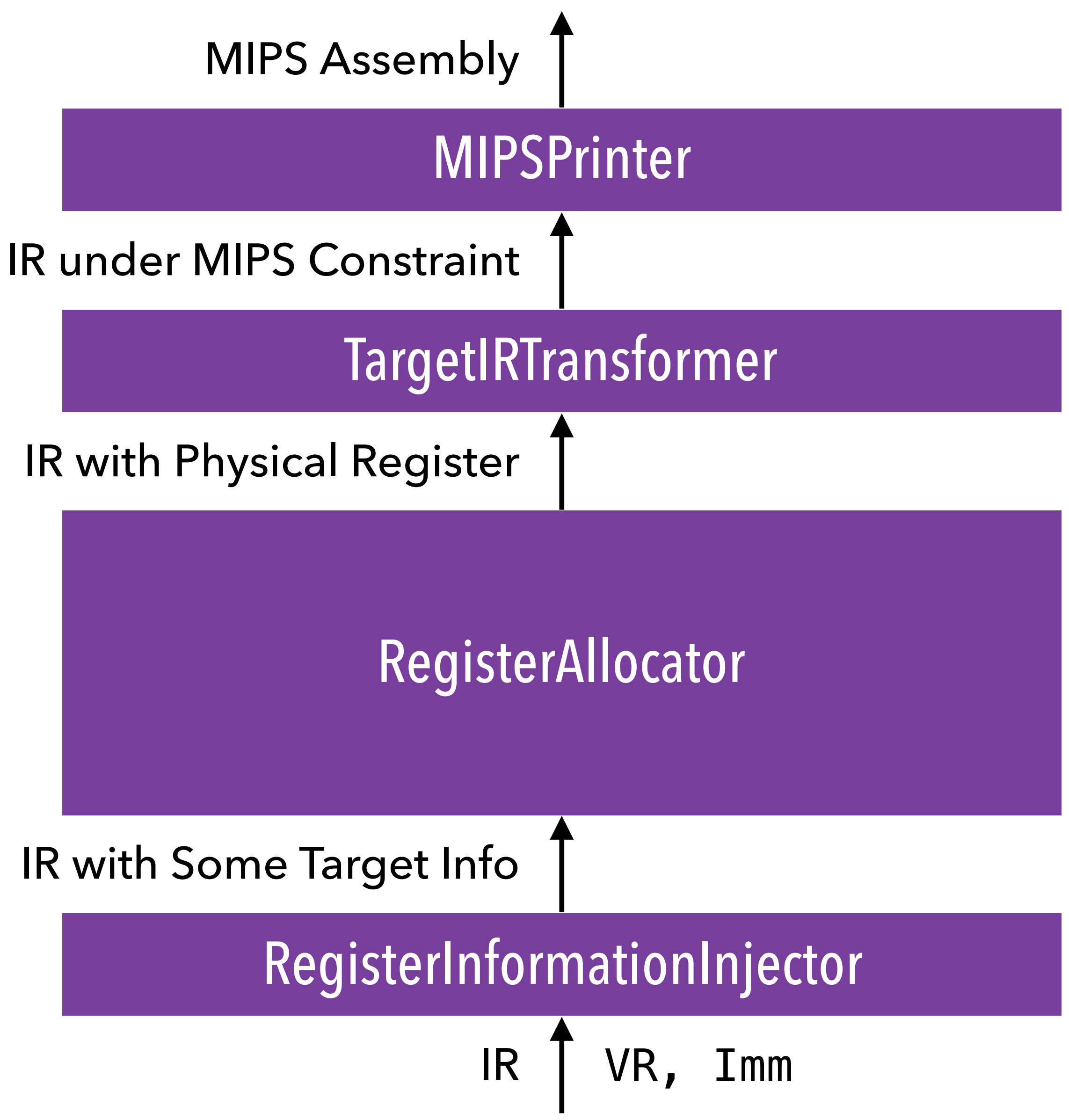
- Don’t save those caller-save registers who are not used by callee (and callee’s callee)
- Need to calculate call graph
- ~10% improvement for program with frequent function calls

| | hashmap | horse2 | spill2 |
|---|---|---|---|
| Limit | 550000 | 15000000 | 100000 |
| Original | 295263 | 9470959 | 18181 |
| Original % | 53.68% | 63.14% | 18.18% |
| Optimized | 240663 | 6711852 | 13065 |
| Optimized % | 43.76% | 44.75% | 13.07% |

# Code Generation

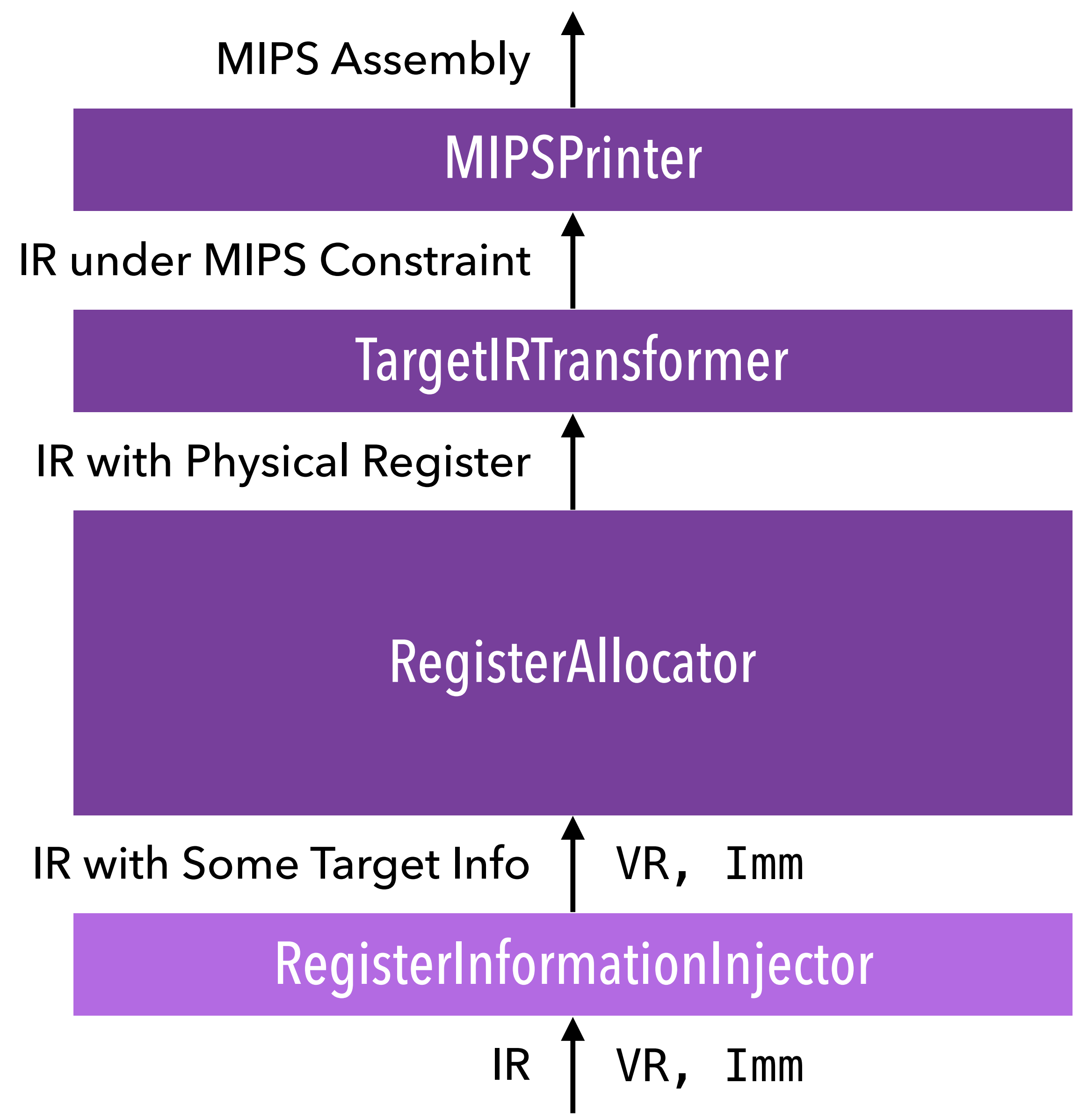
# Code Generation

- No another representation for code generation phase
- Make some modification and attach some target related information to IR instead
- IR before code generation, all operands are:
    - `VirtualRegister`
    - `IntImmediate`



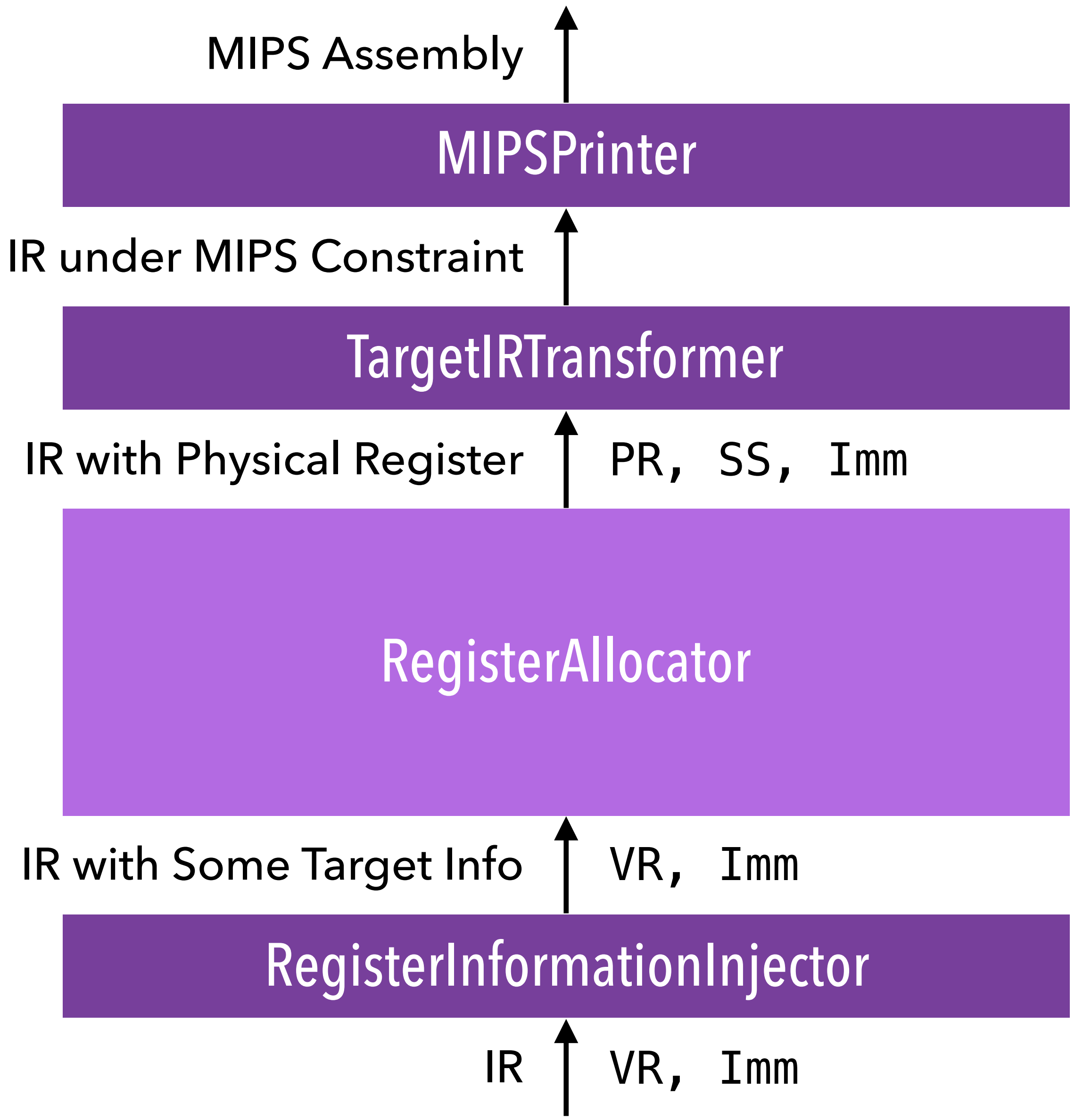
# Code Generation

- RegisterInformationInjector:
- Force first 4 args to be in \$a0, …
- Create StackSlot for each arg
- Replace immediate number (MIPS lhs cannot be an immediate)
  - `%x = shl 1 %y` becomes:
  - `%imm = move 1`
  - `%x = shl %imm %y`
- Replace system call (print, sbrk, …)



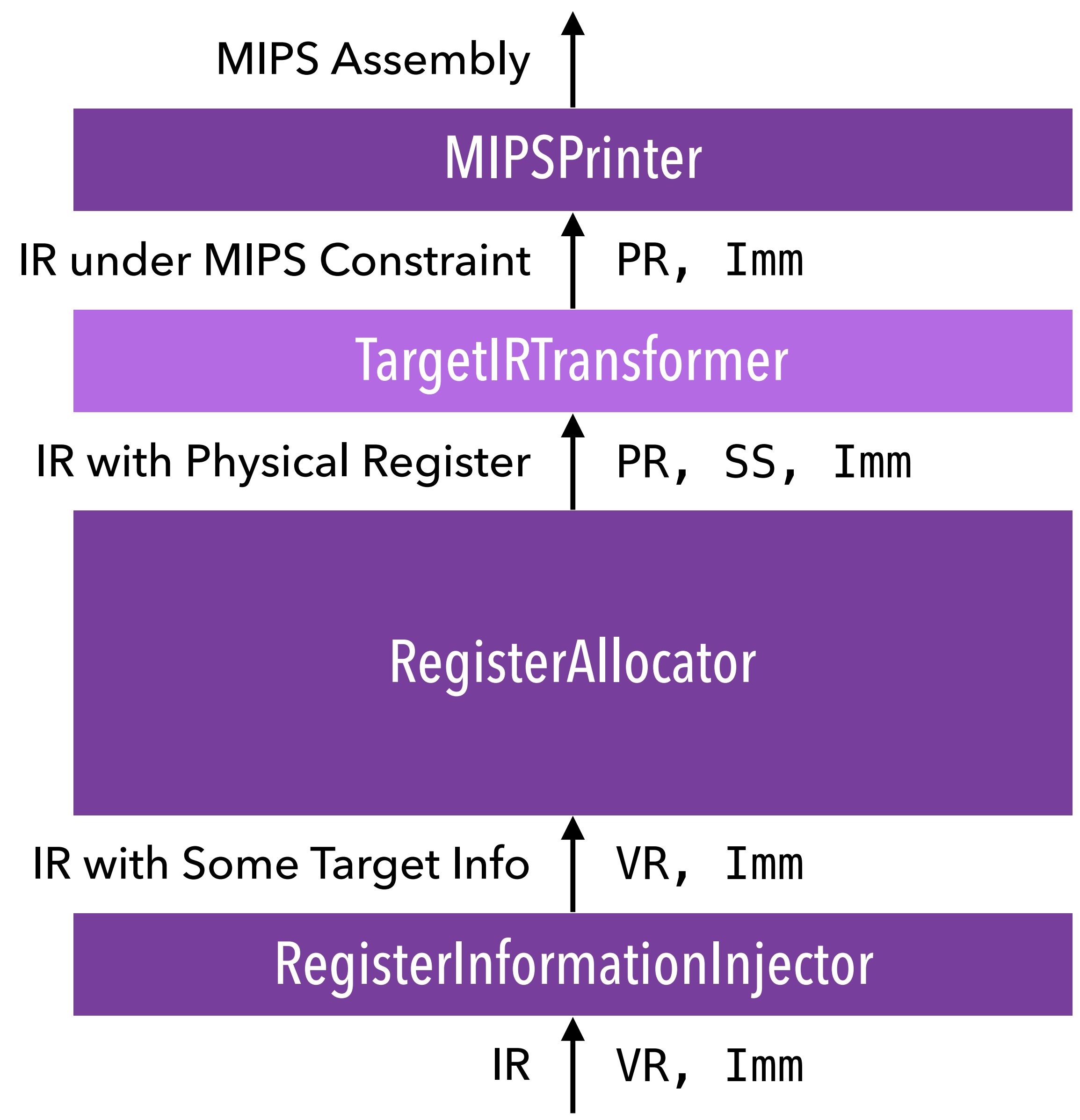
# Code Generation

- RegisterAllocator:
  - `StupidAllocator`
    - Don’t allocate at all (used to ensure correctness)
  - `LocalBottomUpAllocator`
  - `GraphColoringAllocator`
- Rewrite VirtualRegister to either
  - `PhysicalRegister` for lucky ones
  - `StackSlot` for spilled ones



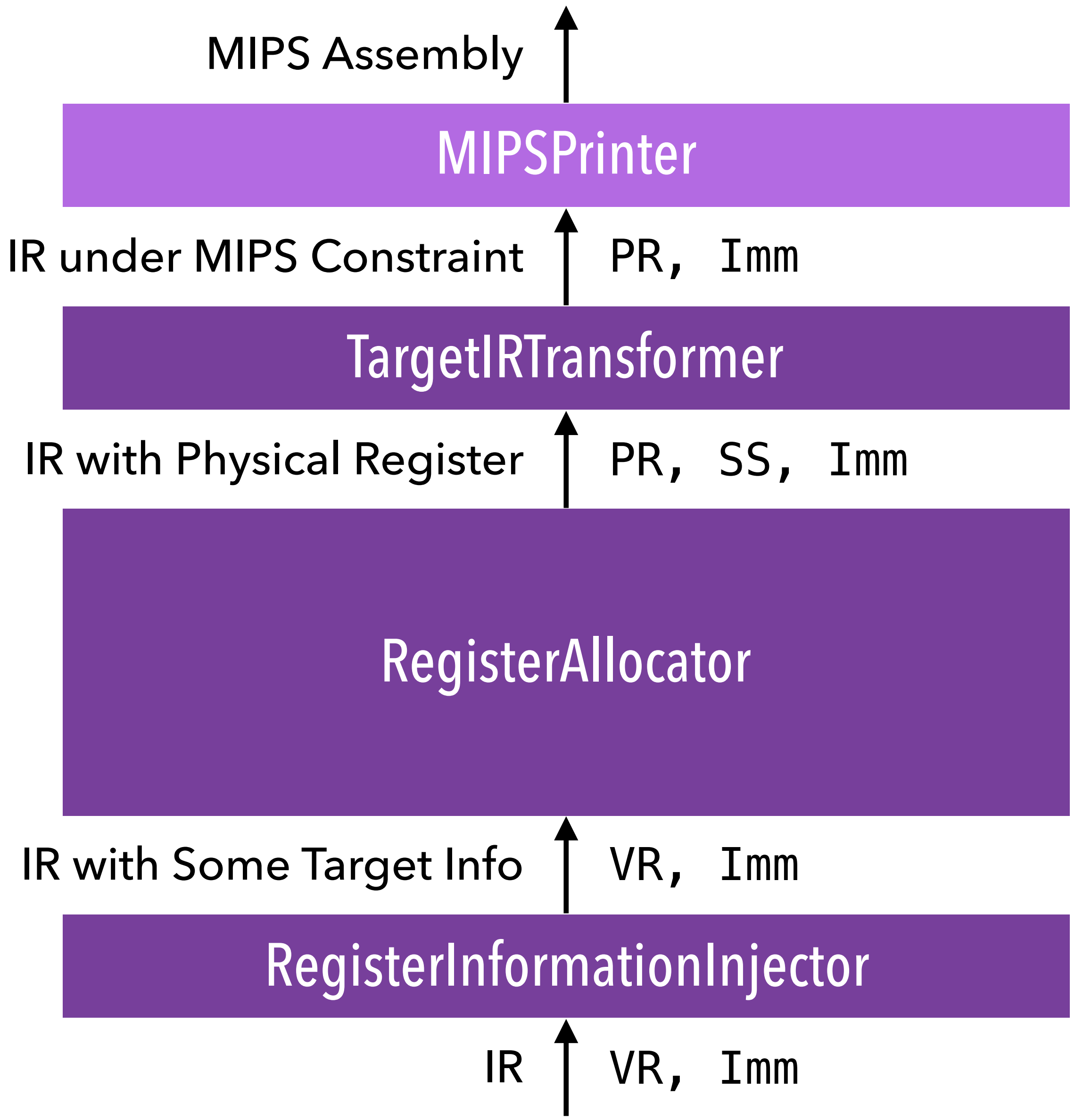
# Code Generation

- TargetIRTransformer:
- Calculate stack frame
- Backup/Restore registers when function entry and exit
- Replace function call
- Replace StackSlot ($sp + offset)
- Remove self move



# Code Generation

- MIPSPrinter:
- Print IR in MIPS assembly format
- Nothing special



# Graph Coloring Reg Alloc

- G. J. Chaitin. *Register Allocation & Spilling via Graph Coloring*
- P. Briggs. *Register Allocation via Graph Coloring*
- Allow forced allocation ($a0, …)
- Allow preferred allocation (`%x = move %y`)
- Don't want to rebuild again after spilling?
  - Trick: Keep 2 registers not allocated to load StackSlot

# Builtin Function

# Builtin Function

- There are two levels of builtin functions, I suppose
  - Source Language Level (strcmp, …)
    - Can be written in the source language
    - Can also be written in the target assembly
  - Target Machine Level (sbrk, print, syscall, …)
    - Can only be written in the target assembly
- I have no good idea to handle builtin functions elegantly.

# Unit Test

# Unit Test

- Unit test is necessary for a project
- If your code quality is good, you'll be able to write tests easily and deeply
- If your project is protected by much tests, you'll be able to debug / refactor / write new code / etc. without fear
- I recommended classmates to do unit test in several forum posts

# LLVM: “`use`” Data Structure

# LLVM: Pass Manager

# Thanks to

# Books

- EAC: Engineering a Compiler, 2nd Edition
  - This is really an awesome one! You'll need it when writing every phase!
- Tiger: Modern Compiler Implementation in Java
- Dragon: Compilers: Principles, Techniques, and Tools
- LIP: Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages
- ANTLR4: The Definitive ANTLR 4 Reference

# And…

- @RednaxelaFX
  - who wrote many answers and articles about compilers/VM on ZhiHu and his blog
  - and answered my questions patiently
- LLVM
- QBE: http://c9x.me/compile/
  - a modern compiler backend of small size
  - maybe bad coding habit (no doc, too short variable name)
  - but worth looking at what it've done, its ideas and bibliography