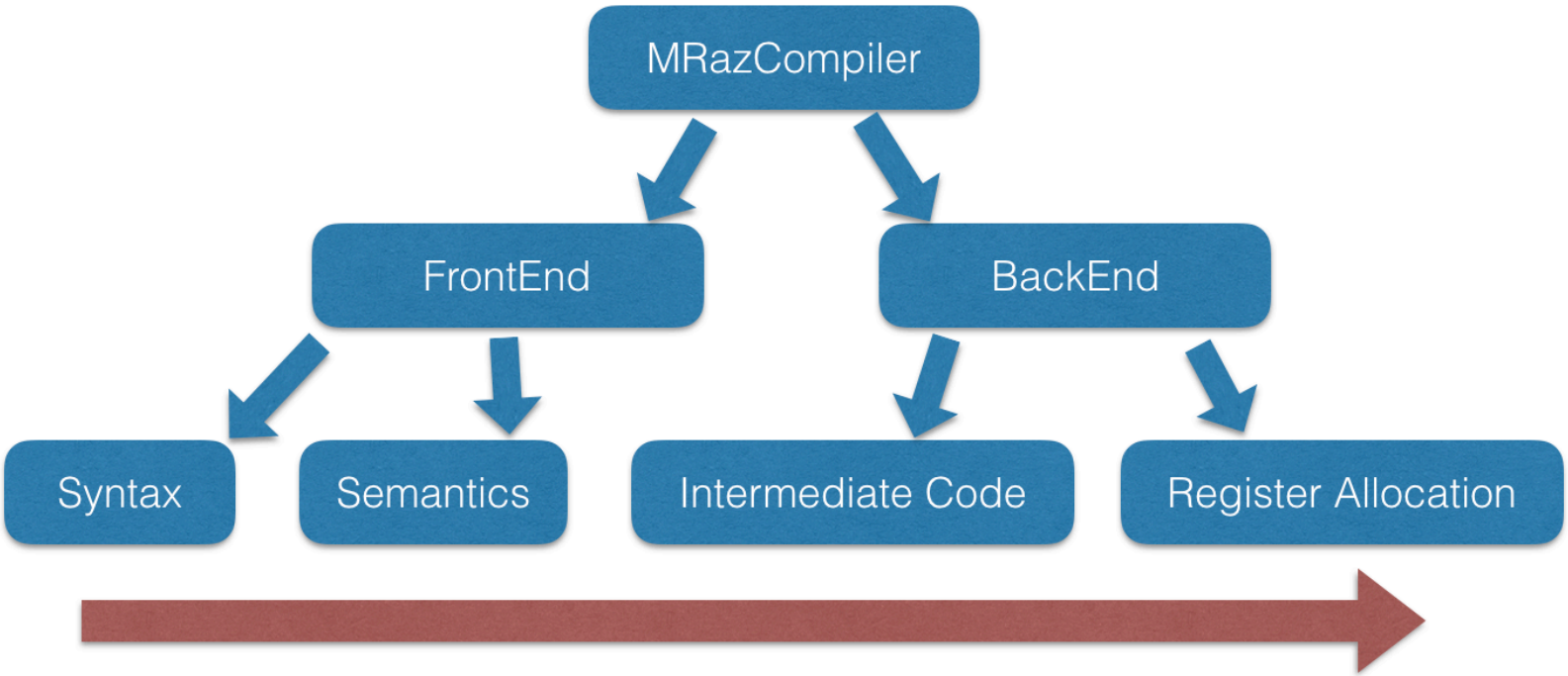
Mr.A-Z

Runzhe Yang

2014 ACM Honoured Class

May 12, 2016

General Structure

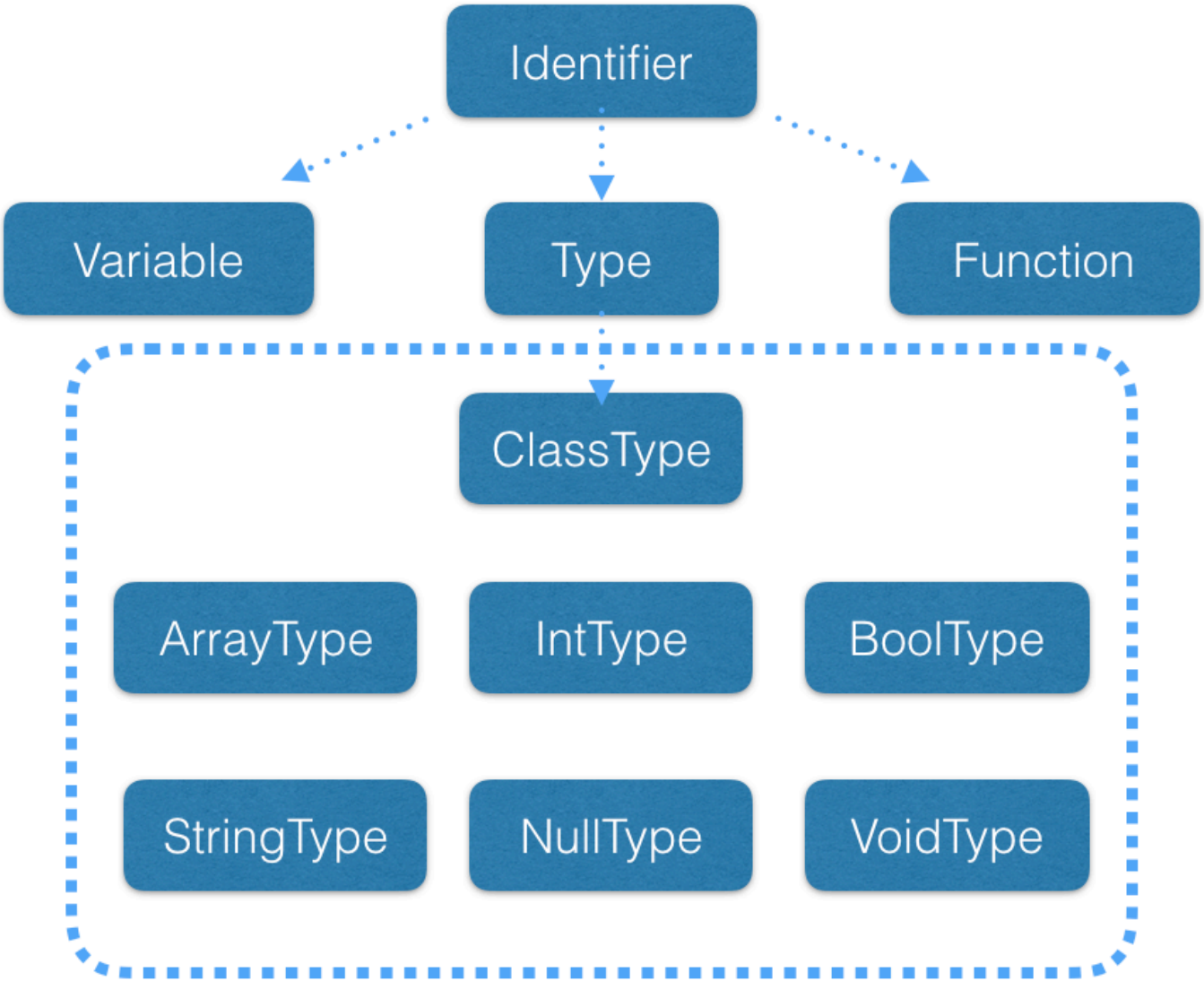


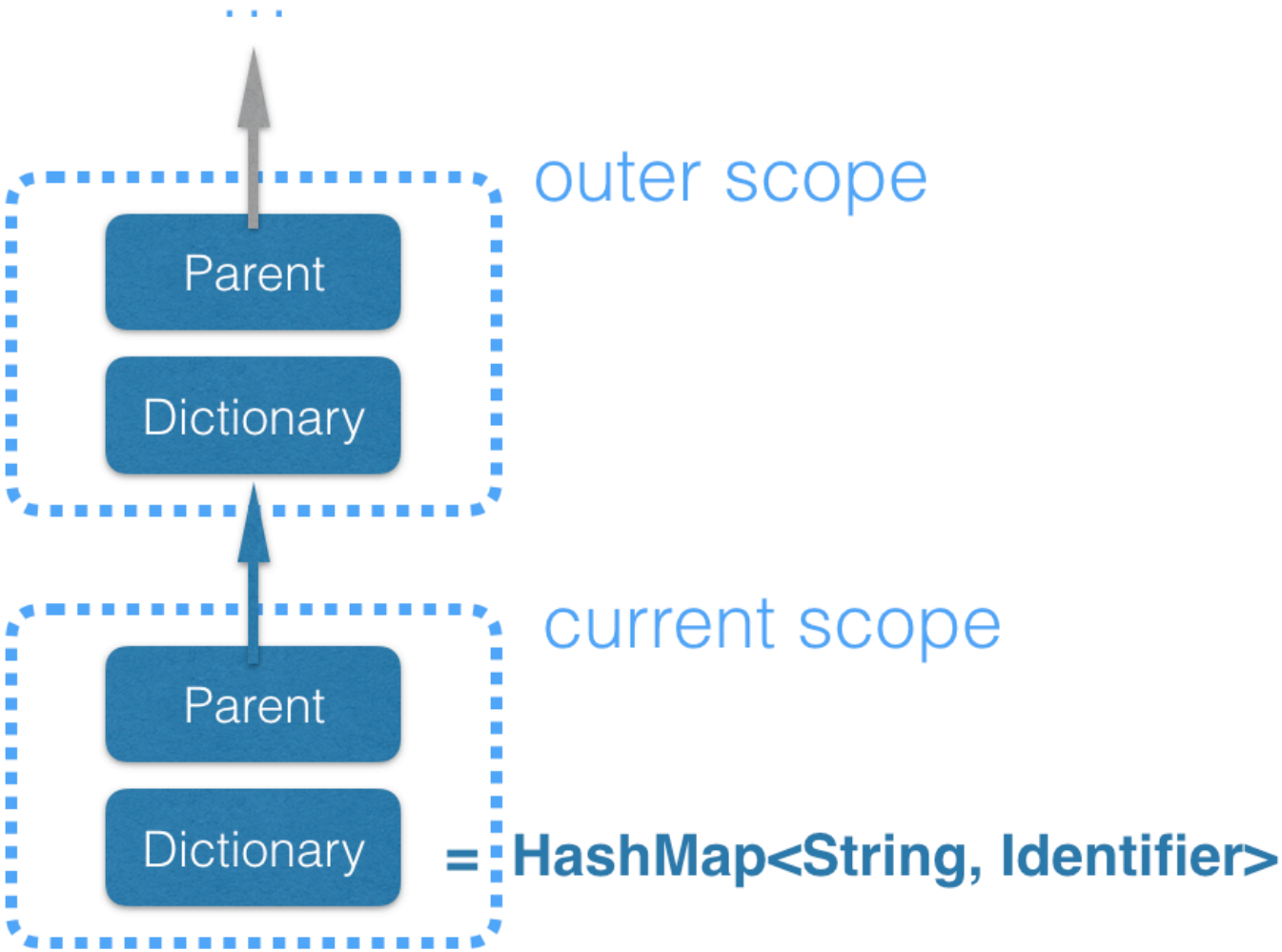
- clear thinking
- concise front end & not too complicated back end
- good performance without re-run optimization
- ummm...actually it doesn't have any.

Rz.g4 $\Rightarrow$

$$\begin{cases} \text{RzLexer: recognize and convert program into token stream} \\ \text{RzParser: generate a parse tree (concrete syntax tree)} \end{cases}$$

- using RzBaseVisitor to travel on CST
- using SymbolTable to record identifier
- using TypeAnalyser to extract the **result type** and **identifier type** of expression
- the whole semantic check contains three rounds


Identifier
Variable
Type
Function
ClassType
ArrayType
IntType
BoolType
StringType
NullType
VoidType


. . .
outer scope
Parent
Dictionary
current scope
Parent
Dictionary
= HashMap<String, Identifier>

Type Analyzer

- Downcast Pattern
- Result type: type of return values or variables
- Identifier type: a variable, a function or a resulted type of expression.
- type checking for expressions

3 Rounds for Semantic Check

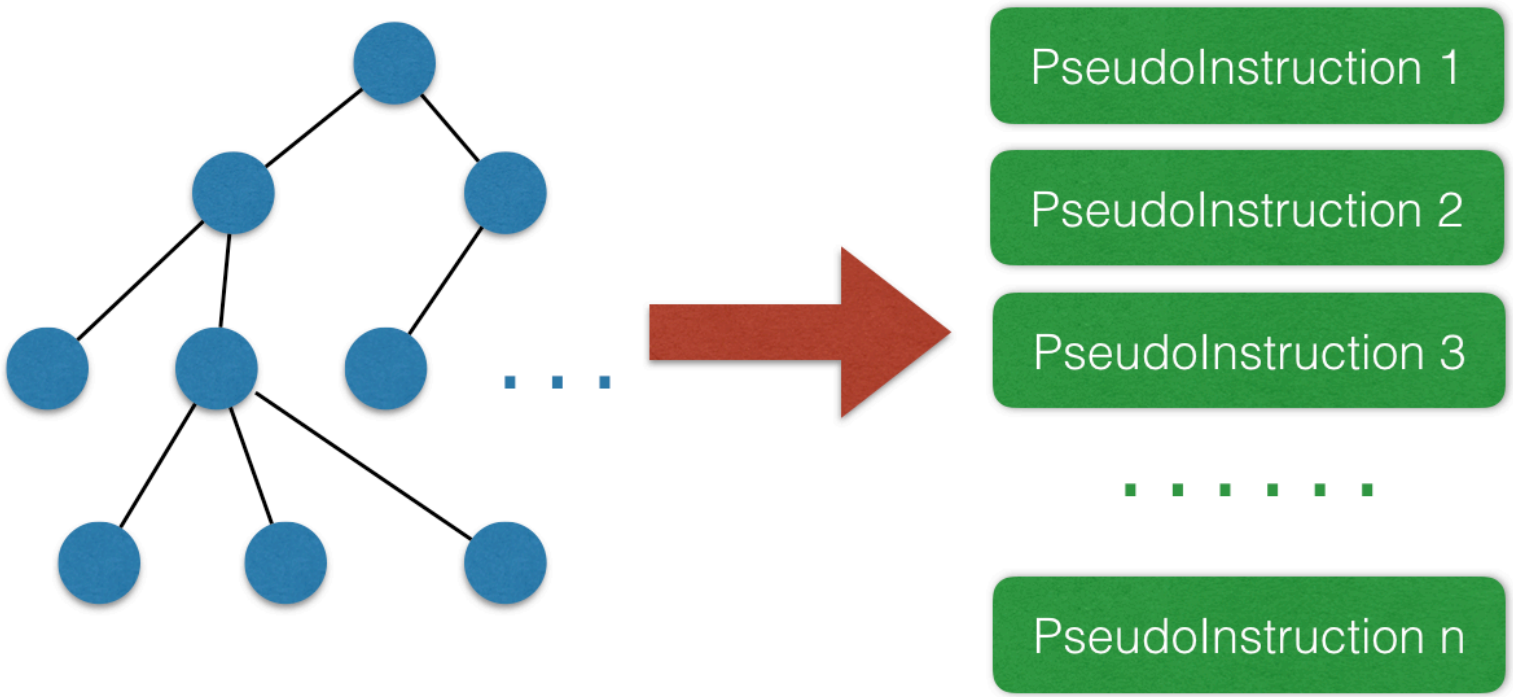
- Round 1: GetClass
  - add class name and set build-in functions to symbol table.
- Round 2: GetFuncAndClassMem
  - get parameter list for each function
  - get members for each class
  - add functions and re-add class name to symbol table
- Round 3: SemanticCheck
  - check the program whether have a "main" function
  - check the variable declaration and **update the symbol table**
  - refresh the outer symbol table of the function
  - link or recover the symbol table when enter or exit scope
  - check the selection and iteration statement
  - check the continue, break and return statement

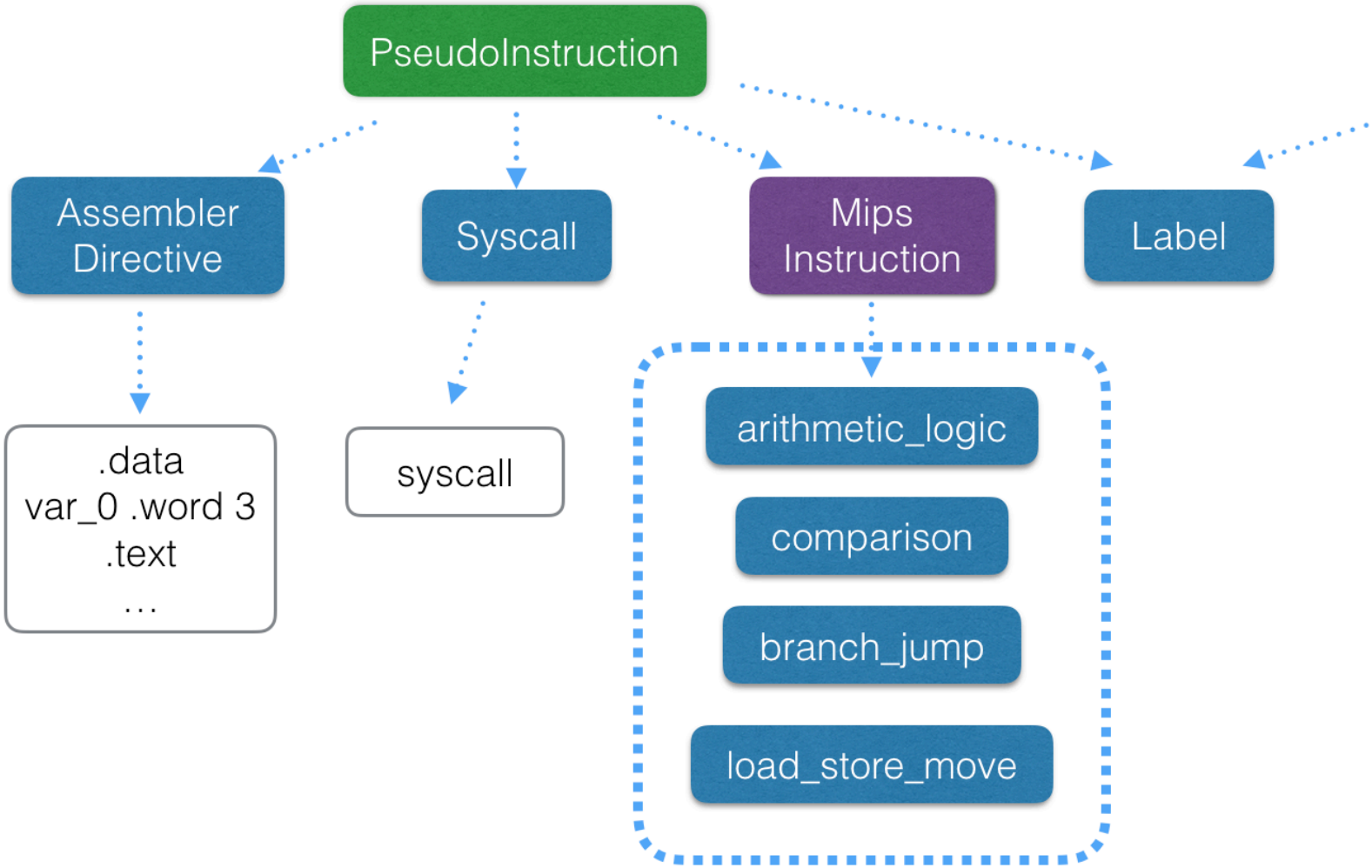
Pretty Print

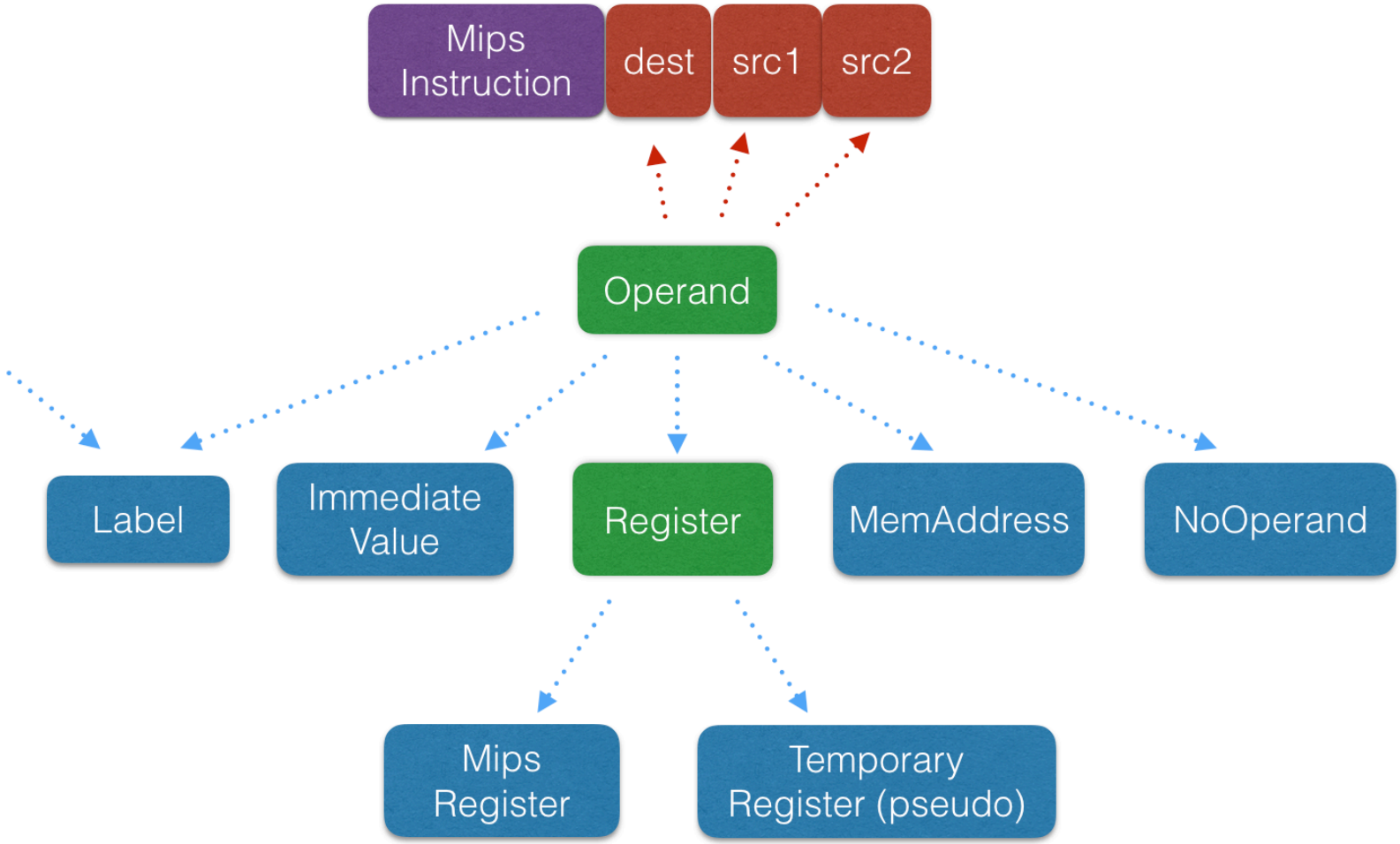
- *frontendText(parser, program, false, true);*

```
int N = 8;int[] row = new int[8];int[]
col = new int[8];int[][] d = new int[2][
];void printBoard() {    int i;    int
j;    for (i = 0; i < N; i++) {
for (j = 0; j < N; j++) {            if
(col[i] == j)                print(" O")
;            else                print(
" .");        }        println("");
}    println("");}void search(int c)
{    if (c == N) {        printBoard()
;    }    else {        int r;
for (r = 0; r < N; r++) {            if
(row[r] == 0 && d[0][r+c] == 0 && d[1][
r+N-1-c] == 0) {                row[r]
= d[0][r+c] = d[1][r+N-1-c] =
1;                col[c] =
r;                search(c+1)
;                row[r] = d[0][r+c] = d[
1][r+N-1-c] = 0;            }
}    }}int main() {    int i;  for (i =
0; i < 2; i ++)        d[i] = new int[8
+ 8 - 1];    search(0);    return 0;}
```

```
//******* PRETTY PRINT *******//

//******* CLASS DECLARATION *******//


//******* GLOBAL VARIABLE *******//

int N = 8;
int[] row = new int[8];
int[] col = new int[8];
int[][] d = new int[2][];

//******* FUNCTION DECLARATION *******//

void printBoard() {
    int i;
    int j;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            if (col[i] == j) print(" O");
            else print(" .");
        }
        println("");
    }
    println("");
}

void search(int c) {
    if (c == N) {
        printBoard();
    }
    else {
        int r;
        for (r = 0; r < N; r++) {
            if (row[r] == 0 && d[0][r + c] == 0 && d[1][r + N - 1 - c] == 0) {
                row[r] = d[0][r + c] = d[1][r + N - 1 - c] = 1;
                col[c] = r;
                search(c + 1);
                row[r] = d[0][r + c] = d[1][r + N - 1 - c] = 0;
            }
        }
    }
}
```


PseudoInstruction 1
PseudoInstruction 2
PseudoInstruction 3
PseudoInstruction n


PseudoInstruction
Assembler Directive
Syscall
Mips Instruction
Label
.data
var_0 .word 3
.text
...
syscall
arithmetic_logic
comparison
branch_jump
load_store_move

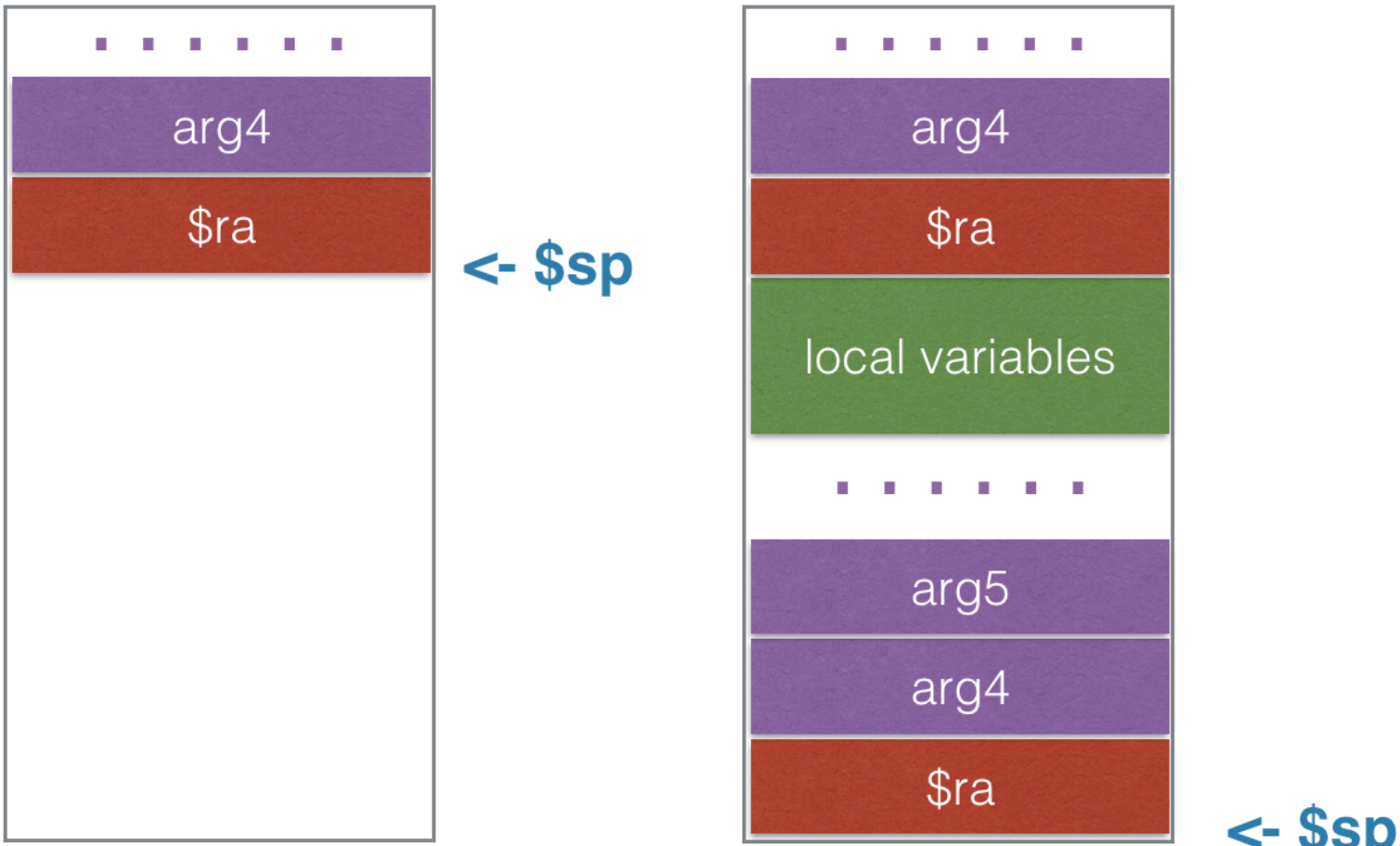

Mips Instruction
dest
src1
src2
Operand
Label
Immediate Value
Register
MemAddress
NoOperand
Mips Register
Temporary Register (pseudo)

Three Minor Things

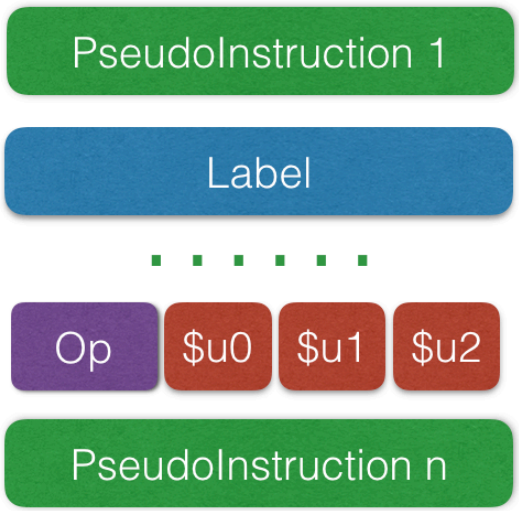
- StringConstGetter
  - extract string constants & add them to data section
  - make a dictionary for looking up corresponding label of each string const
- PreIntermediateCodeTranslator
  - extract global constants & initialize them in data section
  - if the initialization is complicated, then move those instructions to the beginning of main function.
- SeparateIntermediateCodeTranslator
  - translate each function to list of pseudo-instructions

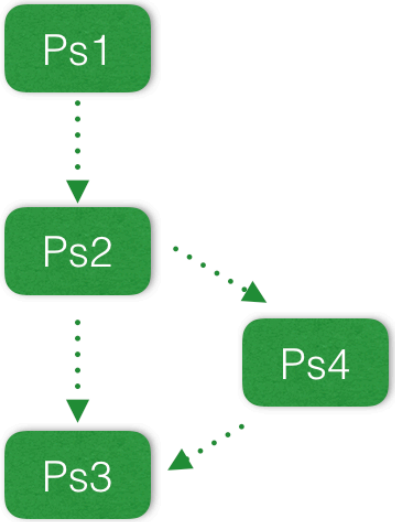
IntermediateCodeTranslator

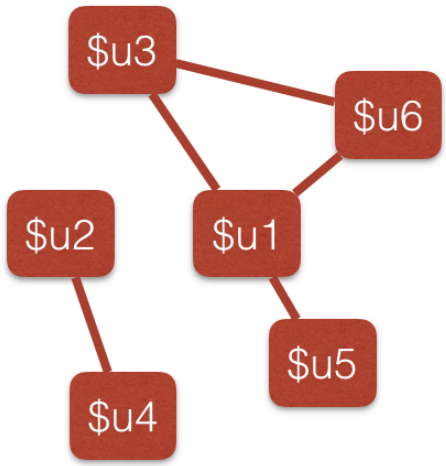
- Visitor Pattern
- TemporaryRegisterGenerator produces unique temporary register (pseudo) for every variable and intermediate result
- synchronize the value of global variables, class type variables and array type variables between register and memory
- constant folding: i = 320 * 200 * 32;
- calculate the minimal offset for $sp



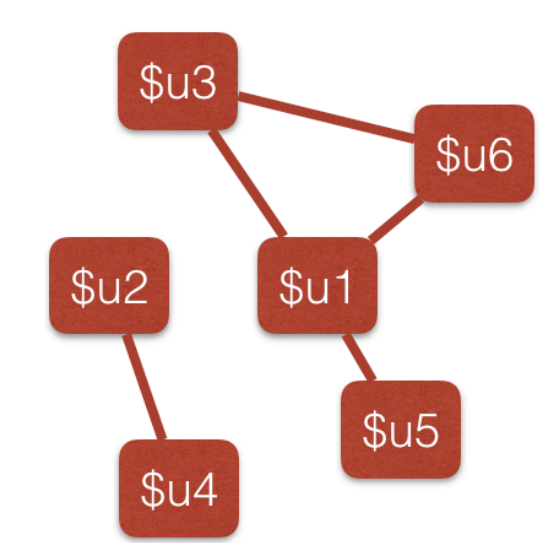
arg0, …,arg3 are in $a0, …$a3

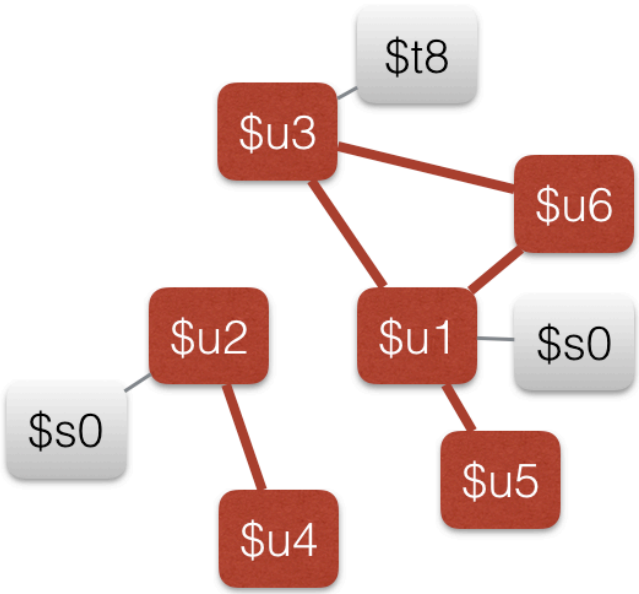

PseudoInstruction 1
Label
Op
$u0
$u1
$u2
PseudoInstruction n

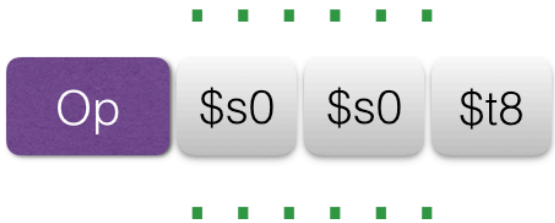

Ps1
Ps2
Ps4
Ps3


$u3
$u6
$u2
$u1
$u5
$u4

Register Allocation







Control Flow Graph

- each node represents a non-label pseudo-instruction
  - DefinedRegisterGetter, UsedRegisterGetter
- backward visit the pseudo-instruction list
- make a dictionary for label and its next non-label instruction
- make a CFG node when meeting a non-label instruction
- pull an edge from current node to previous node
- adapt branch and jump instruction with dictionary

Interference Graph

- calculate liveIn and liveOut for each node
  - liveIn $=$ *Uses* $\cup$ (liveOut $-$ *Defs*)
  - liveOut $= \bigcup_{succ}$ liveIn
- do iteration until they are unchanged
- **tag the instruction if its def temp reg in not in liveOut**
- connect two register if they are in the same liveOut.

FrameManager

- Cast to Memory
  - allocate a memory address to real register
  - update the size of stack frame
- Back from Memory
  - re-load the data from memory to history register
- record register & count minimal offset

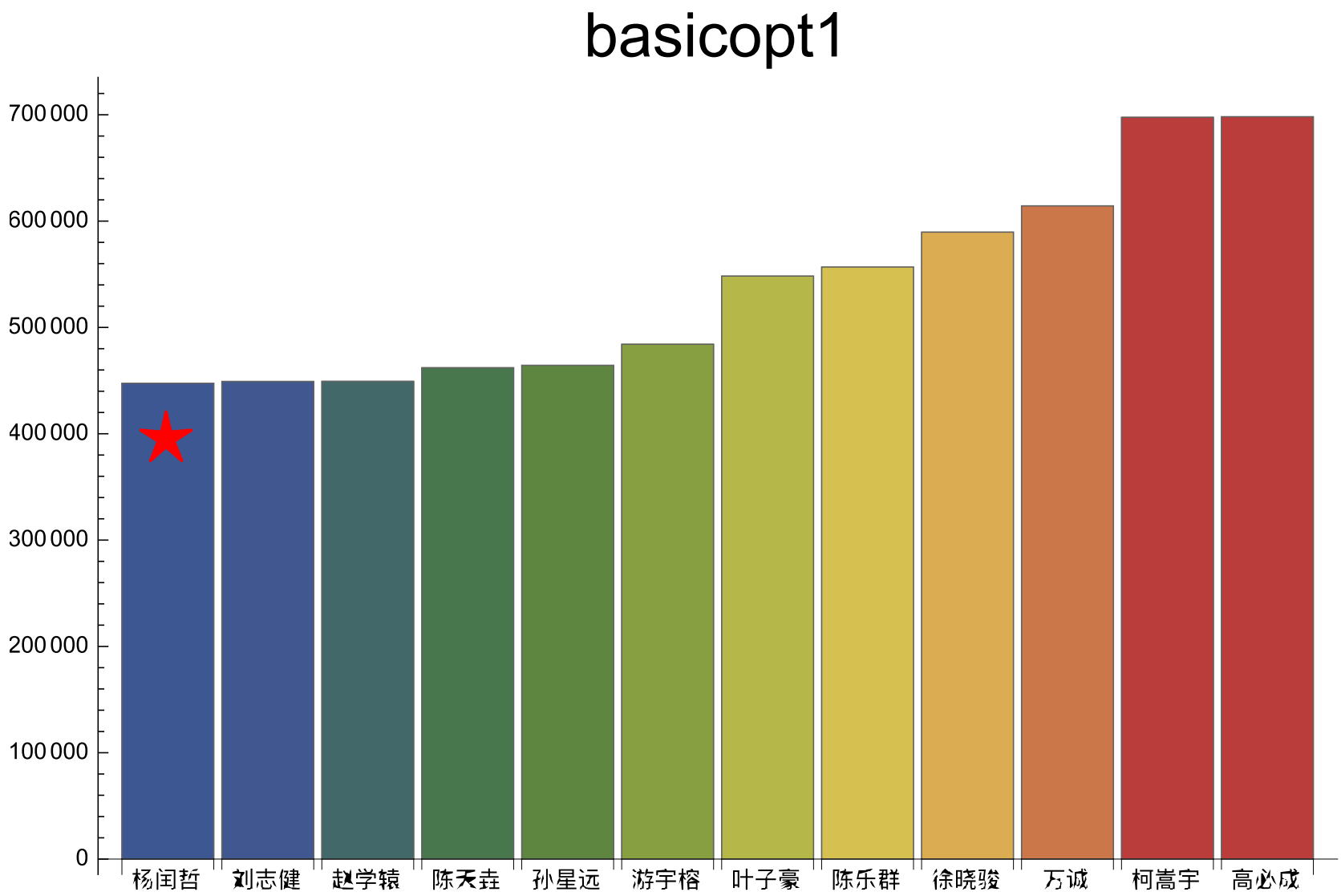
Linear scan the untagged pseudo-instruction in list, for each operand:

- if the temporary register has no history
  - available = {$t0,...,$t9,$s0,...$s7} - $\bigcup_{\text{adjacent}}$ realRegister
  - if available = $\emptyset$, cast one adjacent temp reg to memory, set real register as this one's
  - else set real register as some one in available
- if the temporary register has a history register
  - if the history register $\notin \bigcup_{\text{adjacent}}$ realRegister, then set the real register as the history
  - else cast the adjacent one to memory, and set the real register as the history
- real register, immediate value and label, just keep them

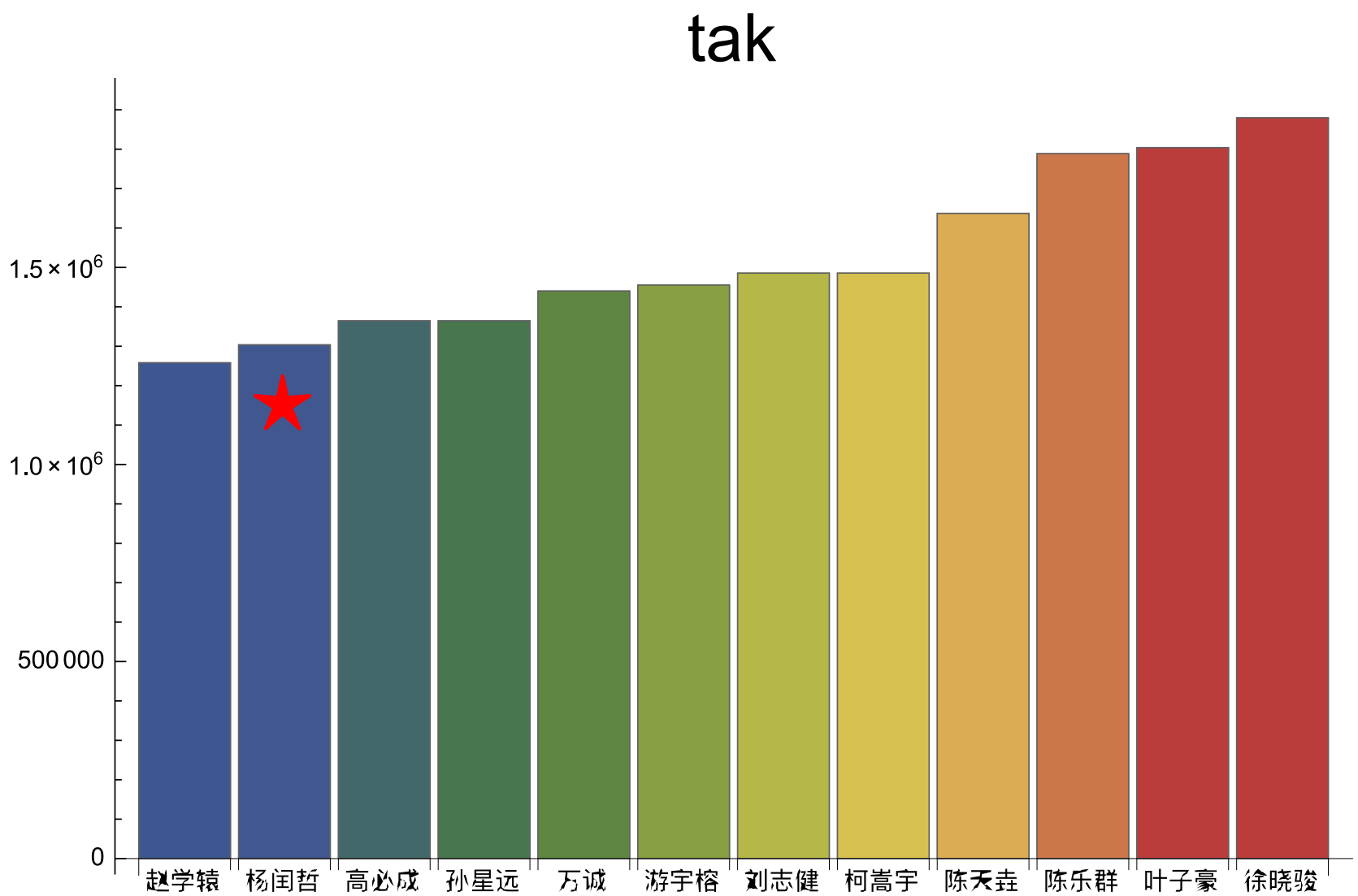
- When meeting "jal" instruction, cast all temporary register registers in its liveIn to memory before calling, and pull them back after calling.
- Correct $sp movement and clean up history every time after register allocation.
- use InstructionPrinter to print the mips assemblers.

Special Case

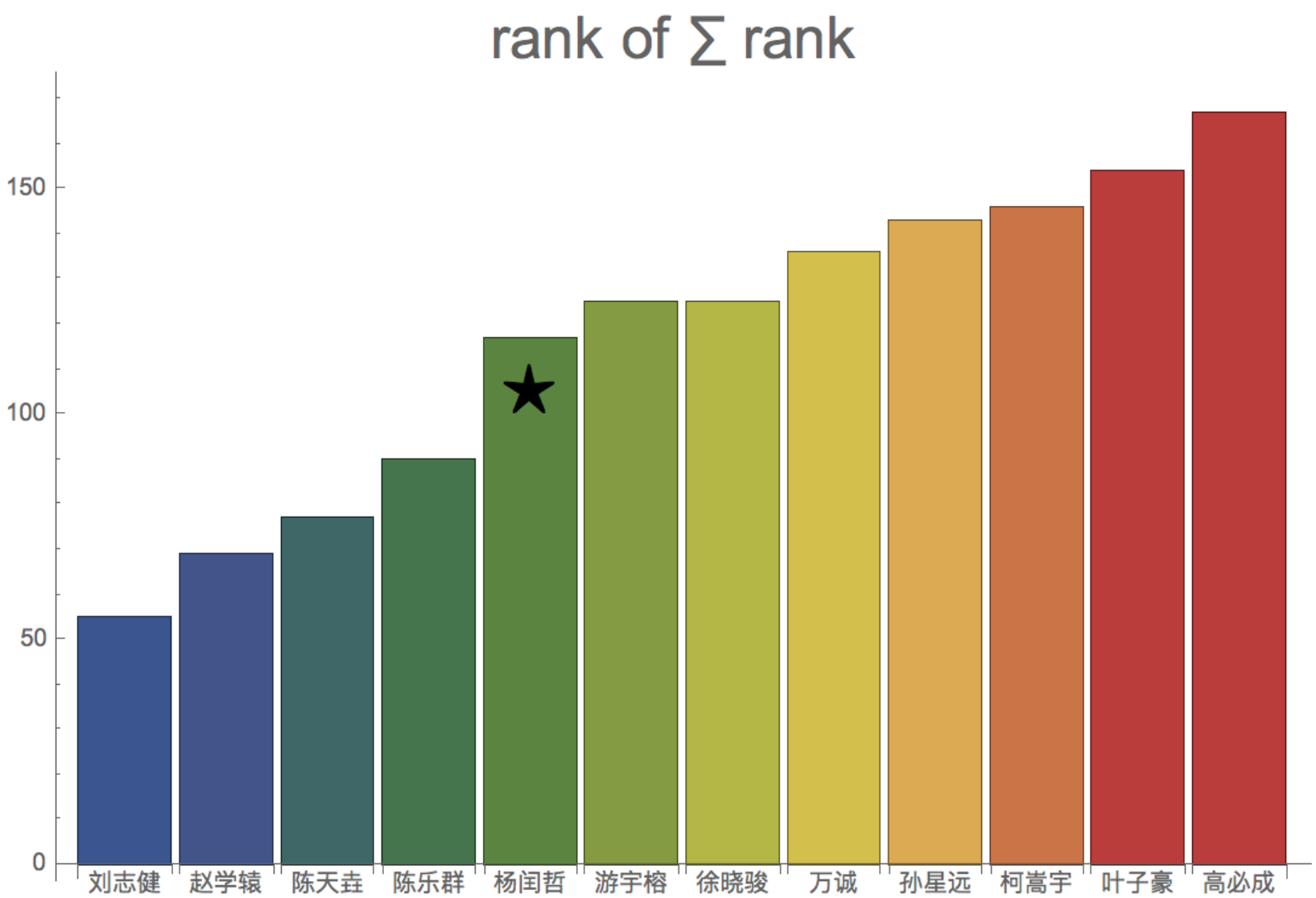
- Spill2: CiscRegisterAllocator
  - Too many CFGNodes & Too dense IG
  - Why not use only three registers, then load, load, store.


basicopt1
700 000
600 000
500 000
400 000
300 000
200 000
100 000
0
杨闰哲
刘志健
赵学辕
陈天垚
孙星远
游宇榕
叶子豪
陈乐群
徐晓骏
万诚
柯嵩宇
高必成

Summary


tak
1.5 × 10^6
1.0 × 10^6
500 000
0
赵学辕
杨闰哲
高必成
孙星远
万诚
游宇榕
刘志健
柯嵩宇
陈天垚
陈乐群
叶子豪
徐晓骏

Summary


rank of ∑ rank
150
100
50
0
刘志健
赵学辕
陈天垚
陈乐群
杨闰哲
游宇榕
徐晓骏
万诚
孙星远
柯嵩宇
叶子豪
高必成

- More precise def-use relationships
  - constant propagation
  - dead code elimination
- Make short functions inline & opt print(toString(a) + " ")
- Loop optimizations
  - loop unrolling
  - loop fusion
  - software pipelining
- Delete redundant memory operations before and after a function call

Mr. A-Z