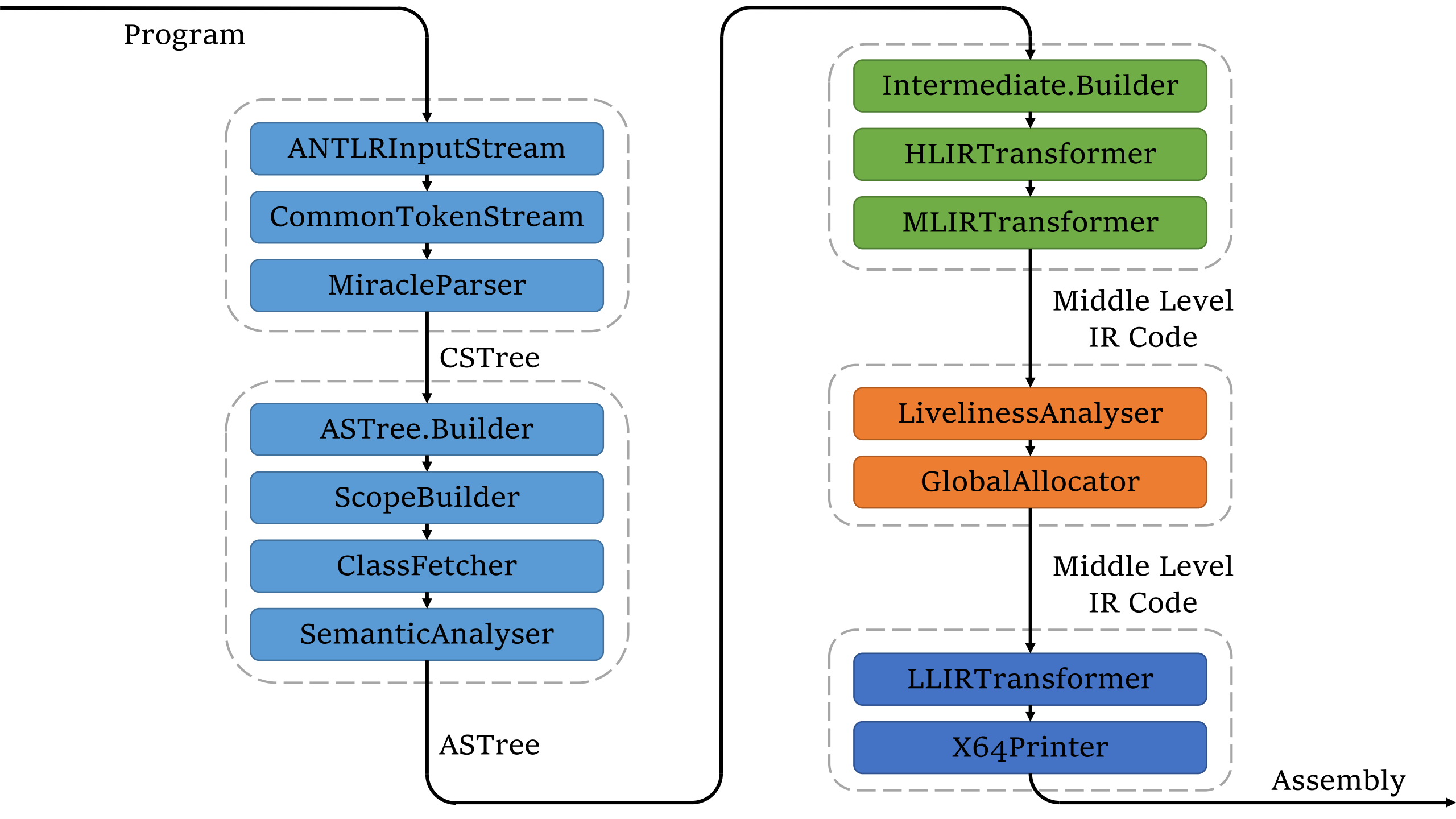
Compiler 2017

Kipsora Lawrence

Overview

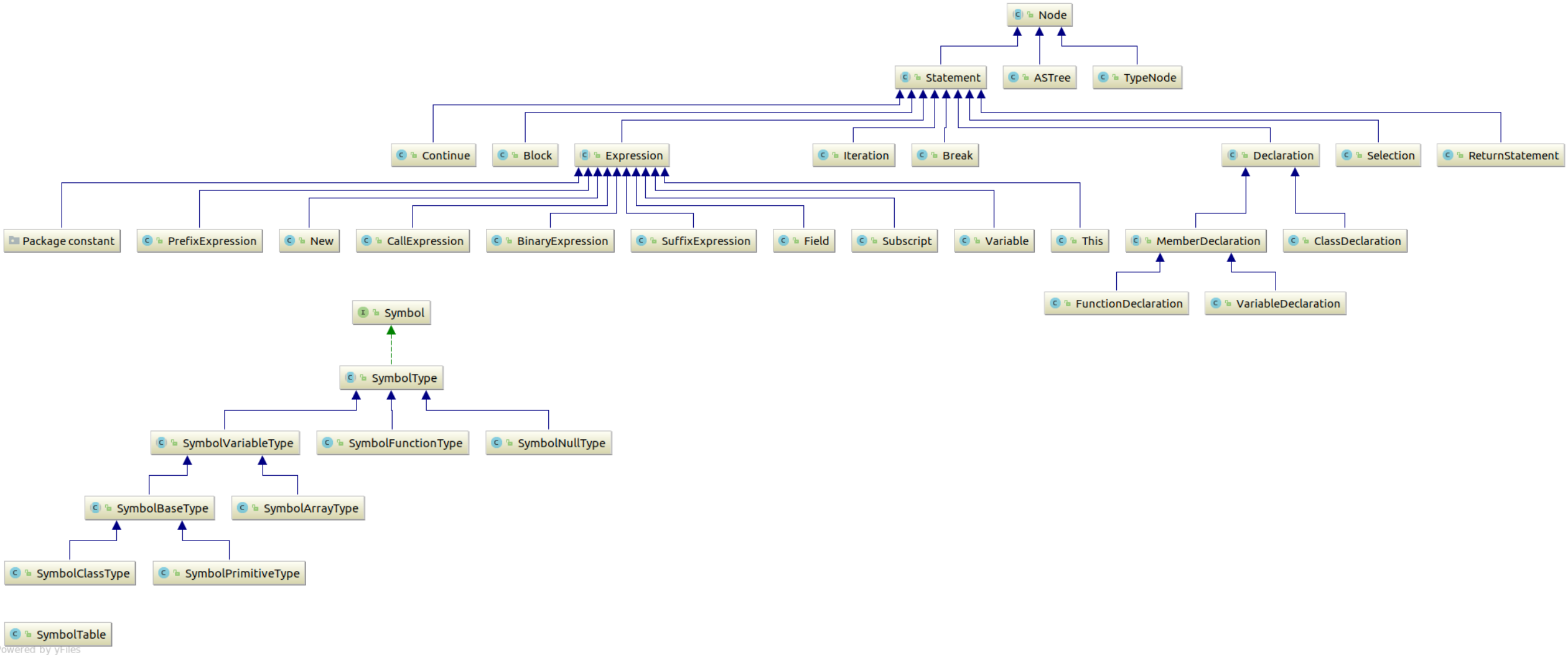
- Rank 7^{th} regardless of the penalty of overdue
- User-friendly semantic checker
- Command parameters
- SSA Constructor and SSA Destructor
- Desperately implemented Liveliness analyzer and global allocator


Program
ANTLRInputStream
CommonTokenStream
MiracleParser
CSTree
ASTree.Builder
ScopeBuilder
ClassFetcher
SemanticAnalyser
ASTree
Intermediate.Builder
HLIRTransformer
MLIRTransformer
Middle Level IR Code
LivelinessAnalyser
GlobalAllocator
Middle Level IR Code
LLIRTransformer
X64Printer
Assembly

Semantic Checker

ANTLR ~~are~~ a very useful tool.
is

Extending Graph



The 1st Version of Semantic Analyzer

- Considering the principle that simple is fast, reduced

  ANTLR G4 file is used:

```
expression: constant                                          #constantExpression
    | 'this'                                                  #thisExpression
    ......
    | expression operator='.' IDENTIFIER                      #fieldExpression
    | expression operator=('*' | '/' | '%') expression        #binaryExpression
    | expression operator=('+' | '-') expression              #binaryExpression
    | expression operator=('<<' | '>>') expression            #binaryExpression
    ......
    | expression operator=('==' | '!=') expression            #binaryExpression
    ......
```

- Construct abstract syntax tree with semantic checker first

The 2nd Version of Semantic Analyzer

- Feeling WTF after a few days when implementing linear intermediate representation
- Constructing the whole abstract syntax tree first, then applying class fetcher and other checkers
- Checkers becoming relatively easy to implement
- Omitting some redundant details

Semantic Analyzer

- User-friendly error reports:

```
Miracle:2:5: error: cannot declare a variable of type `void`
  void a;
  ^
Miracle:3:9: error: cannot find identifier named "a"
  return a;
         ^
Miracle:3:9: error: identifier "a" is not declared as a valid
expression
  return a;
         ^
Miracle:5:8: error: the rawval of left operand is not mutable
  true = false;
  ^
```

Symbols

- Function are treated as a special type of variables
- Every type is virtually treated as a class with member variables and member methods
- `NULL` is a special type with only one instance `null`.
- Built-in methods and type comparator should be carefully implemented due to the abstractions above

Symbol Table

- Tried O(1) symbol table first
- Complicated implementations
- Re-write with O(n) symbol table
- Remember parent scope in each scope
- Performs well in practice

Tricks in Symbol Table

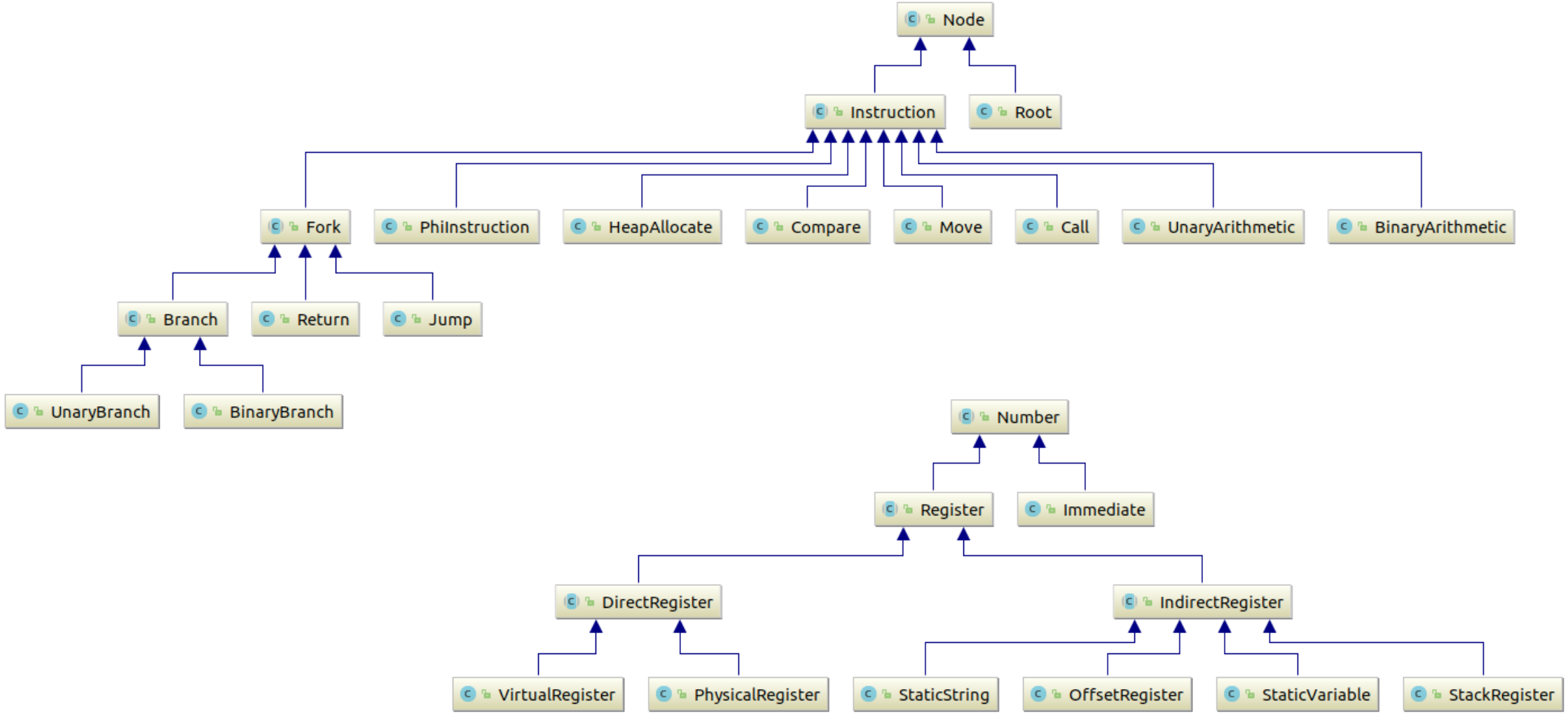
- Normal variables are not able to be accessed before definition
  - But member variables can be accessed anywhere
  - Checking order is needed
- Class identifiers, function identifiers and variable identifiers are stored in one map
  - Resolving process should be carefully implemented

Intermediate Representation

The cat ran at the rat

call cat.run target=rat

Extending Graph



LLVM IR

- Confused by the conception of intermediate representation
- Look for a standard IR
- LLVM IR with SSA Form!
- Having a view of “standard IR”

```
define i32 @main() nounwind {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval
  %0 = load i32* %a, align 4
  %1 = load i32* %b, align 4
  %d = add nsw i32 %0, %1
  ret i32 %d
}
```

https://www.zhihu.com/question/40032931

Single Static Assignment Form

- Fascinated by LLVM, I decide to use SSA form directly after the construction of abstract syntax tree
- Some paper discovered some simple methods to construct SSA form from abstract syntax tree
- Time limited, I decided to finish basic test first and then transform the code to SSA form

https://www.zhihu.com/question/40032931
https://pp.info.uni-karlsruhe.de/uploads/publikationen/braun13cc.pdf

Single Static Assignment Form

- I think I should use Tarjan's methods to construct dominator tree first and then construct SSA Form
- I have looked up many materials of SSA form to find a efficient implementation
- Find a simple method in Lequn Chen's report, which suggest a book named Engineering a Compiler(EAC) and also SSA Books

http://ssabook.gforge.inria.fr/latest/book.pdf
https://www.cs.cmu.edu/~fp/courses/15411-f08/lectures/09-ssa.pdf

Calculating Dominator Frontier

- Performed well in practice

```
for all nodes, b /* initialize the dominators array */
    doms[b] ← Undefined
doms[start_node] ← start_node
Changed ← true
while (Changed)
    Changed ← false
    for all nodes, b, in reverse postorder (except start_node)
        new_idom ← first (processed) predecessor of b /* (pick one) */
        for all other predecessors, p, of b
            if doms[p] ≠ Undefined /* i.e., if doms[p] already calculated */
                new_idom ← intersect(p, new_idom)
        if doms[b] ≠ new_idom
            doms[b] ← new_idom
            Changed ← true
```

```
function intersect(b1, b2) returns node
    finger1 ← b1
    finger2 ← b2
    while (finger1 ≠ finger2)
        while (finger1 < finger2)
            finger1 = doms[finger1]
        while (finger2 < finger1)
            finger2 = doms[finger2]
    return finger1
```

Number of Nodes	*Iterative Algorithm* Dominance DOM	DF	Postdominance DOM	DF	*Lengauer-Tarjan/Cytron et al.* Dominance DOM	DF	Postdominance DOM	DF
> 400	3148	1446	2753	1416	7332	2241	6845	1921
201–400	1551	716	1486	674	3315	1043	3108	883
101–200	711	309	600	295	1486	446	1392	388
51–100	289	160	297	151	744	219	700	191
26–50	156	86	165	94	418	119	412	99
<= 25	49	26	52	25	140	32	134	26

http://www.cs.rice.edu/~keith/EMBED/dom.pdf

Constructing SSA Form

```
Globals ← ∅
Initialize all the Blocks sets to ∅
for each block b
    VarKill ← ∅
    for each operation i in b, in order
        assume that op_i is "x ← y op z"
        if y ∉ VarKill then
            Globals ← Globals ∪ {y}
        if z ∉ VarKill then
            Globals ← Globals ∪ {z}
        VarKill ← VarKill ∪ {x}
        Blocks(x) ← Blocks(x) ∪ {b}
```

(a) Finding Global Names

```
for each name x ∈ Globals
    WorkList ← Blocks(x)
    for each block b ∈ WorkList
        for each block d in DF(b)
            if d has no φ-function for x then
                insert a φ-function for x in d
                WorkList ← WorkList ∪ {d}
```

(b) Rewriting the Code

http://bank.engzenon.com/download/560e72f1-0a74-4507-8385-12aec0feb99b/Engineering_a_Compiler_2nd_edition_by_Cooper_and_Tor

Destructing SSA Form

Algorithm 22.1: Algorithm making non-conventional SSA form conventional by isolating ϕ-nodes

```
1  begin
2      foreach B: basic block of the CFG do
3          insert an empty parallel copy at the beginning of B;
4          insert an empty parallel copy at the end of B;
5      foreach B_0: basic block of the CFG do
6          foreach φ-function at the entry of B_0 of the form a_0 = φ(B_1 : a_1, ..., B_n : a_n) do
7              foreach a_i (argument of the φ-function corresponding to B_i) do
8                  let PC_i be the parallel-copy at the end of B_i;
10
11                 let a'_i be a freshly created variable;
12                 add copy a'_i ← a_i to PC_i;
13                 replace a_i by a'_i in the φ-function;
14             begin
15                 let PC_0 be the parallel-copy at the beginning of B_0;
17
18                 let a'_0 be a freshly created variable;
19                 add copy a_0 ← a'_0 to PC_0;
20                 replace a_0 by a'_0 in the φ-function;
               /* all a'_i can be coalesced and the φ-function removed     */
```

http://ssabook.gforge.inria.fr/latest/book.pdf

Registers in IR

- Virtual register is a ideal conception that registers are infinite
- Offset-registers is another conception used to represent subscription and member accessing operations
- A naivë approach translating IR into assembly is to map every virtual register to memory

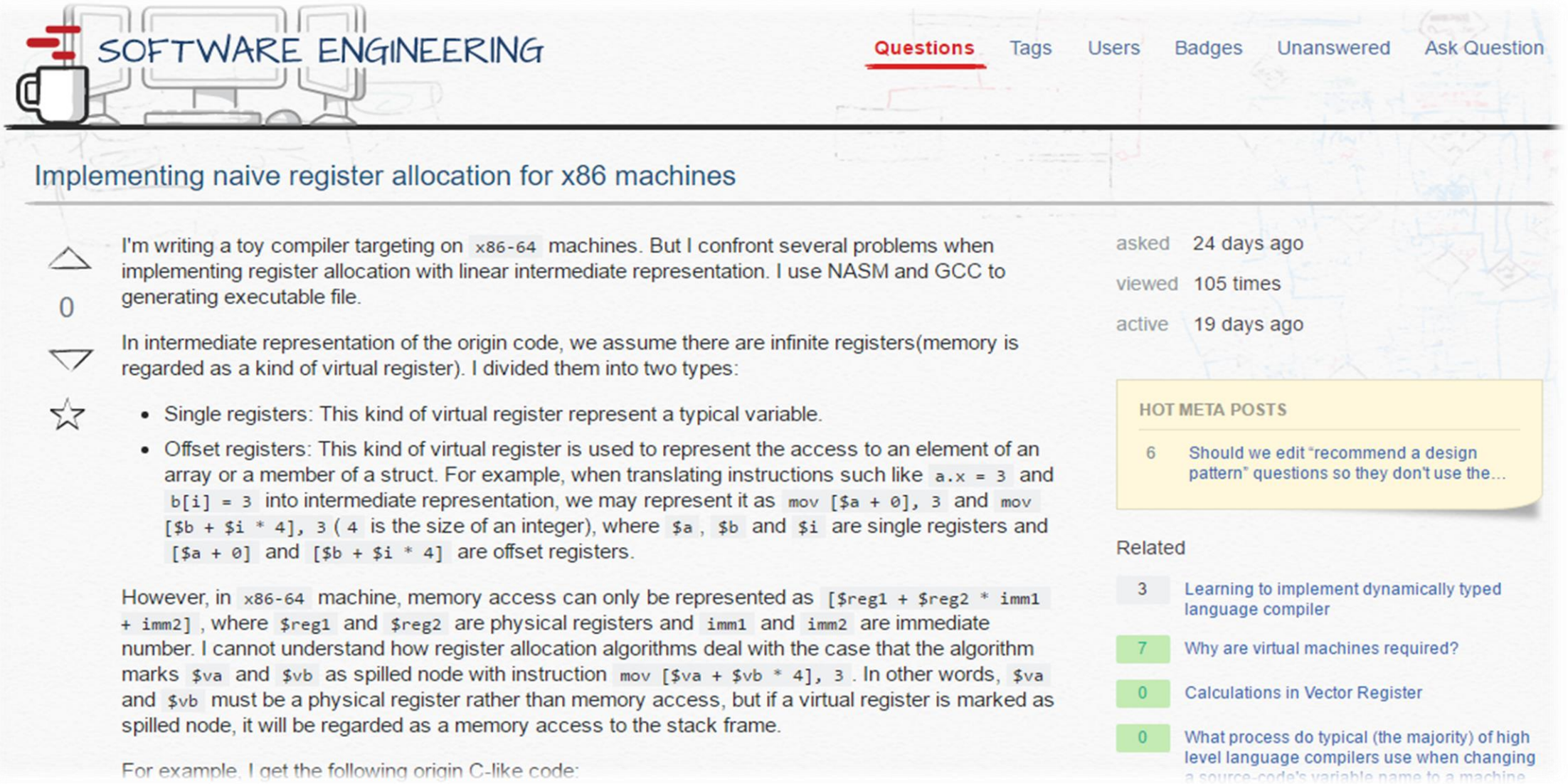
https://www.zhihu.com/question/29355187

Troubles

- I feel tremendously difficult to translating IR into assembly considering the following case:

```
index_a = 3                    mov dword [$rbp - 4], 3
index_b = 4                ⟶   mov dword [$rbp - 8], 4
a[index_a] = a[index_b]        mov dword [$rax + dword [$rbp - 4]],
                                   dword [$rax + dword [$rbp - 8]]
```

- I raise a question on StackExchange and finally I surrender to selecting some special registers for compromise



https://softwareengineering.stackexchange.com/questions/349684/implementing-naive-register-allocation-for-x86-machines

GCC's Solution

- GCC uses several passes to deal with this case
- In the first pass, GCC use Graph Coloring to color each virtual register
- If a node is marked *spilled* and this node cannot be in memory, then make a copy and re-allocate all virtual registers
- The new copied virtual registers have few degrees, hence few passes can solve this problem

ftp://gcc.gnu.org/pub/gcc/summit/2003/Graph%20Coloring%20Register%20Allocation.pdf

Optimization & Code Generation

That what you say is false is true, which is false actually

What you say is true

mov $[what your say], true

Register Allocation

- Delayed by the troubles above, I desperately implemented register allocation in a really short time with overdue penalty finally
- I use 9 registers for allocation, and RAX, R14, R15, RBP, RSP, RCX are reserved
- Code really become faster after register allocation
- Allocator is naivë, but the performance is very well
- Debugging and handling tricks are tremendously troublesome

Optimization in Code Generation

- Usually we uses `push` and `pop` to pass spilled calling parameters, which needs a subtraction of `RSP` register
- I calculate the size of all the spilled parameters first and then directly use `sub $RSP, SIZE` and then use `mov` instruction to pass the calling parameters
- And the `bool` type is only one bit which make calculation related with `bool` faster.

What I Have Learned in This Project

- I have learned how to extract useful information from a technical manual and from a paper
- I have learned how to implement a tremendously complicated project in practice
- I have learned how to struggle with my classmates who helped me a lot when I face some difficulties
- I have learned how to keep calm under the overwhelming pressure

References

- Engineering a Compiler
- Tiger Books
- ふつうのコンパイラをつくろう
- NASM Tutorial
- Lequn Chen’s Project
- Zhijian Liu’s Project
- ……

Thanks for Listening