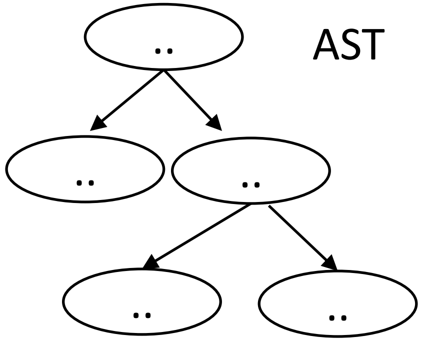
CSE110A: Compilers

May 11, 2022

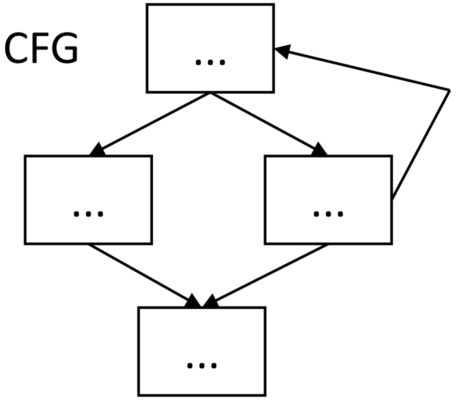
AST



Topics:

- *Finishing up scopes for 3 address code*
- *Homework review*
- *Start of Module 4*

CFG



3 address code

```
store i32 0, ptr %2
%3 = load i32, ptr %1
%4 = add nsw i32 %3, 1,
store i32 %4, ptr %1
%5 = load i32, ptr %2
```

Announcements

- Pending grades
  - HW 2 (expect by Monday)
  - Midterm (expect by next Friday)
- HW 3 is released
  - Due in two weeks from release date
  - We will go over some of it during class today
  - Get started early; you have all the material you need!

Review

- Converting statements into ClassIeR

Let’s do another one

```
statement := declaration_statement
           |  assignment_statement
           |  if_else_statement
           |  block_statement
           |  for_loop_statement
```

```
if_else_statement := IF LPAR expr RPAR statement ELSE statement

{
    eat("IF");
    eat("LPAR");
    expr_ast = parse_expr()
    ...
    program0 = # type safe and linearized ast
    eat("RPAR");
    program1 = parse_statement()
    eat("ELSE")
    program2 = parse_statement()
    ...
}
```

```
if (program0) {
   program1
}
else {
   program2
}
```

We need to convert this to 3 address code

```
if_else_statement := IF LPAR expr RPAR statement ELSE statement

{
    eat("IF");
    eat("LPAR");
    expr_ast = parse_expr()
    ...
    program0 = # type safe and linearized ast
    eat("RPAR");
    program1 = parse_statement()
    eat("ELSE")
    program2 = parse_statement()
    ...
}
```

```
if (program0) {
   program1
}
else {
   program2
}
```

We need to convert this to 3 address code

```
  program0;
  vrX = int2vr(0)
  beq(expr_ast.vr, vrX, else_label);
  program1
  branch(end_label);
else_label:
  program2
end_label:
```

```
if_else_statement := IF LPAR expr RPAR statement ELSE statement

{
   ...
   # get resources
   end_label  = mk_new_label()
   else_label = mk_new_label()
   vrX        = mk_new_vr()

   # make instructions
   ins0 = “%s = int2vr(0)” % vrX
   ins1 = “beq(%s, %s, %s);” %
          (expr_ast.vr, vrX, else_label)
   ins2 = “branch(%s)” % end_label

   # concatenate all programs
   return program0 + [ins0, ins1] + program1
          + [ins2, label_code(else_label)]
          + program2 + [label_code(end_label)]
}
```

```
if (program0) {
  program1
}
else {
  program2
}
```

We need to convert this to 3 address code

```
  program0;
  vrX = int2vr(0)
  beq(expr_ast.vr, vrX, else_label);
  program1
  branch(end_label);
else_label:
  program2
end_label:
```

```
if_else_statement := IF LPAR expr RPAR statement ELSE statement

{
   ...
   # get resources
   end_label  = mk_new_label()
   else_label = mk_new_label()
   vrX        = mk_new_vr()

   # make instructions
   ins0 = “%s = int2vr(0)” % vrX
   ins1 = “beq(%s, %s, %s);” %
          (expr_ast.vr, vrX, else_label)
   ins2 = “branch(%s)” % end_label

   # concatenate all programs
   return program0 + [ins0, ins1] + program1
          + [ins2, label_code(else_label)]
          + program2 + [label_code(end_label)]
}
```

```
class VRAllocator():
    def __init__(self):
        self.count = 0


    def get_new_register(self):
        vr = "vr" + str(self.count)
        self.count += 1
        return vr
```

```
if_else_statement := IF LPAR expr RPAR statement ELSE statement

{
   ...
   # get resources
   end_label  = mk_new_label()
   else_label = mk_new_label()
   vrX        = mk_new_vr()

   # make instructions
   ins0 = “%s = int2vr(0)” % vrX
   ins1 = “beq(%s, %s, %s);” %
          (expr_ast.vr, vrX, else_label)
   ins2 = “branch(%s)” % end_label

   # concatenate all programs
   return program0 + [ins0, ins1] + program1
          + [ins2, label_code(else_label)]
          + program2 + [label_code(end_label)]
}
```

```
class LabelAllocator():
    def __init__(self):
        self.count = 0


    def get_new_register(self):
        lb = "label" + str(self.count)
        self.count += 1
        return lb
```

Compiling Scopes

Scopes

```
int x;
int y;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

How can we get rid of the {}'s?

What do x and y hold at the end of the program?

Scopes

Let's walk through it with a symbol table

```
int x;
int y;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

Scopes

Let’s walk through it with a symbol table

```
int x;
int y;
x = 5;
{
   int x;
   x = 6;
   y = x;
}
```

HT0

symbol table hash table stack

Scopes

rename

Let's walk through it with a symbol table

```
int x_0;
int y;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

make a new unique name for x

HT0

x: (INT, VAR, "x_0")

symbol table hash table stack

Scopes

Let's walk through it with a symbol table

```
int x_0;
int y;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

HT0

```
x: (INT, VAR, "x_0")
```

symbol table hash table stack

Scopes

rename

Let’s walk through it with a symbol table

```
int x_0;
int y_0;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

make a new unique name for y

HT0

```
x: (INT, VAR, “x_0”)
y: (INT, VAR, “y_0”)
```

symbol table hash table stack

Scopes

search

Let’s walk through it with a symbol table

```
int x_0;
int y_0;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

HT0

```
x: (INT, VAR, "x_0")
y: (INT, VAR, "y_0")
```

symbol table hash table stack

Scopes

replace
with
new name

Let's walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
  int x;
  x = 6;
  y = x;
}
```

HT0

```
x: (INT, VAR, "x_0")
y: (INT, VAR, "y_0")
```

symbol table hash table stack

Scopes

Let’s walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
  int x;
  x = 6;
  y = x;
}
```

new scope. Add x with a new name

HT1	x: (INT, VAR, “x_1”)
HT0	x: (INT, VAR, “x_0”) y: (INT, VAR, “y_0”)

symbol table hash table stack

Scopes

Let’s walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
  int x_1;
  x = 6;
  y = x;
}
```

new scope. Add x with a new name

HT1	x: (INT, VAR, “x_1”)
HT0	x: (INT, VAR, “x_0”) y: (INT, VAR, “y_0”)

symbol table hash table stack

Scopes

Let’s walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
  int x_1;
  x = 6;
  y = x;
}
```

lookup

new scope. Add x with a new name

HT1	x: (INT, VAR, “x_1”)
HT0	x: (INT, VAR, “x_0”) y: (INT, VAR, “y_0”)

symbol table hash table stack

Scopes

Let's walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
  int x_1;
  x_1 = 6;
  y = x;
}
```

lookup

new scope. Add x with a new name

HT1

```
x: (INT, VAR, "x_1")
```

HT0

```
x: (INT, VAR, "x_0")
y: (INT, VAR, "y_0")
```

symbol table hash table stack

Scopes

Let’s walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
  int x_1;
  x_1 = 6;
  y = x;
}
```

lookup

new scope. Add x with a new name

HT1

x: (INT, VAR, “x_1”)

HT0

x: (INT, VAR, “x_0”)
y: (INT, VAR, “y_0”)

symbol table hash table stack

Scopes

Let's walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
  int x_1;
  x_1 = 6;
  y_0 = x_1;
}
```

lookup

new scope. Add x with a new name

HT1

```
x: (INT, VAR, "x_1")
```

HT0

```
x: (INT, VAR, "x_0")
y: (INT, VAR, "y_0")
```

symbol table hash table stack

Scopes

Let's walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
{
  int x_1;
  x_1 = 6;
  y_0 = x_1;
}
```

new scope. Add x with a new name

No more need for {}

HT1	x: (INT, VAR, "x_1")
HT0	x: (INT, VAR, "x_0") y: (INT, VAR, "y_0")

symbol table hash table stack

Scopes

Let’s walk through it with a symbol table

```
int x_0;
int y_0;
x_0 = 5;
int x_1;
x_1 = 6;
y_0 = x_1;
```

new scope. Add x with a new name

HT1	x: (INT, VAR, “x_1”)
HT0	x: (INT, VAR, “x_0”) y: (INT, VAR, “y_0”)

symbol table hash table stack

No more need for {}

How do you implement this?

- It is not a “search and replace” preprocess
- You do it during parsing
- Only required for program variables, not IO variables

Class-IR

Remind ourselves what we are compiling

```
void test4(float &x) {
  int i;
  for (i = 0; i < 100; i = i + 1) {
    x = x + i;
  }
}
```

We only need new names for program variables, not for IO variables

Scopes

Get the new name put in the symbol table when the declaration is parsed

```
int x;
int y;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

make a new unique name for x

HT0

x: (INT, VAR, "x_0")

symbol table hash table stack

Scopes

```
int x;
int y;
x = 5;
{
  int x;
  x = 6;
  y = x;
}
```

Use the new names in:

- *the lhs side of an assignment statement*
- *unit nodes in expressions*

HT1	x: (INT, VAR, “x_1”)
HT0	x: (INT, VAR, “x_0”) y: (INT, VAR, “y_0”)

symbol table hash table stack

building an expression AST, we parse a unit at the base

```
unit := ID
     |  ...
```

How do we know whether to make an IO node or a Var node?

```
{
   id_name = self.to_match[1]
   id_data = # get id_data from the symbol table
   eat("ID")
   if (id_data.id_type == IO)
       return ASTIOIDNode(id_name, id_data.data_type)
   else
       return ASTVarIDNode(id_data.new_name, id_data.data_type)
}
```

id_data should contain:

***id_type**: IO or Var*

***data_type**: int or float*

***new_name**: new unique name*

Homework review

End of Module 3

- We went from an implicit parse tree to an explicit AST
- We transformed typed expressions into equivalent untyped expressions
- We defined a simple 3-address code and compiled expressions and statements to that 3-address code
- By the end of the homework, you will have a functioning IR compiler!
  - ClassIeR is pretty close to an assembly ISA!

Start of module 4: optimizations

Discussion

- What are compiler optimizations?

- Why do we want compiler optimizations?

Discussion

- What are compiler optimizations?
  - automated program transforms designed to make code more optimal
  - optimal can mean different things
    - code optimized for one system might be different for code optimized for a different system
    - we can optimize for speed, for energy efficiency, or for code size. What else?

- Why do we want the compiler to help us optimize?
  - So we can write more maintainable/portable code
  - So we don’t have to worry about learning nuanced details about every possible system

Discussion

- What are some compiler optimizations you know about?

Discussion

- What are some compiler optimizations you know about?

```
for (int i = 0; i < 10; i++) {
  x = x + 1;
}
```

loop unrolling

```
for (int i = 0; i < 10; i++) {
  x = x + 1;
  i++;
  x = x + 1;
}
```

Discussion

- What are some compiler optimizations you know about?

```
for (int i = 0; i < 10; i++) {
  x = x + 1;
}
```

loop unrolling

```
for (int i = 0; i < 10; i++) {
  x = x + 1;
  i++;
  x = x + 1;
}
```

```
int foo() {
  int i,j,k;
  i = 10;
  j = i;
  k = j;
  return k;
}
```

constant propagation

```
int foo() {
  int i,j,k;
  return 10;
}
```

Discussion

- What are some compiler optimizations you know about?

```
for (int i = 0; i < 10; i++) {
  x = x + 1;
}
```

loop unrolling

```
for (int i = 0; i < 10; i++) {
  x = x + 1;
  i++;
  x = x + 1;
}
```

What does this save us?

Discussion

- What are some compiler optimizations you know about?

```
for (int i = 0; i < 10; i++) {
  x = x + 1;
}
```

loop unrolling

```
for (int i = 0; i < 10; i++) {
  x = x + 1;
  i++;
  x = x + 1;
}
```

What does this save us?

optimizations at one stage can enable optimizations at another stage:

```
for (int i = 0; i < 10; i+=2) {
  x = x + 2;
}
```

provides a bigger window for local analysis

Discussion

- What are some compiler optimizations you know about?

let's do a few more

Function inlining

```
int add(int x, int y) {
  return x + y;
}

int foo(int x, int y, int z) {
  return add(x,y);
}
```

```
int foo(int x, int y, int z) {
  return x + y;
}
```

What does this save us?
code size? speed? the ability to debug? local regions to optimize more?

Discussion

- What are some compiler optimizations you know about?

There are many more! This is an active area of research and development

For a rough metric:

`git effort` shows activities on different files and directories

clang C++/C parser: 3.5K commits
clang AST: 8.7K commits
LLVM transforms/optimizations: 30K commits

The transformation part of the code base has the most activity by far

Discussion

- How do you enable compiler optimizations?

Discussion

- How do you enable compiler optimizations?

Discussion

- How do you enable compiler optimizations?

- most C/++ compilers
  - optimizing for speed
    - -O0, -O1, -O2, -O3
    - what about O4?
  - optimizing for size
    - -Os, -Oz
  - relax some constraints (especially around floating point):
    - -Ofast
    - Godbolt example

https://stackoverflow.com/questions/15548023/clang-optimization-levels

Discussion

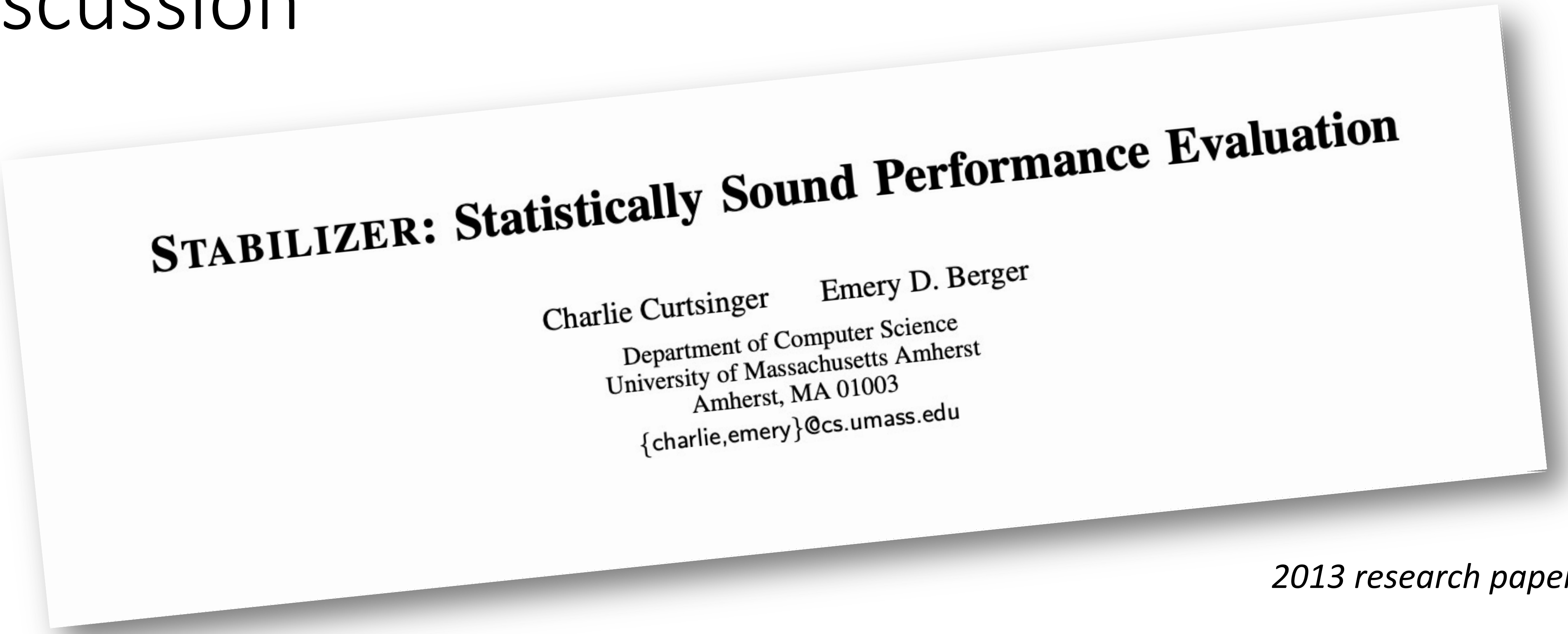
- How do you enable compiler optimizations?

- most C/++ compilers
  - optimizing for speed
    - -O0, -O1, -O2, -O3
    - what about O4?
  - optimizing for size
    - -Os, -Oz
  - relax some constraints (especially around floating point):
    - -Ofast
    - Godbolt example

Does -O3 actually make a difference?

https://stackoverflow.com/questions/15548023/clang-optimization-levels

Discussion



STABILIZER: Statistically Sound Performance Evaluation

Charlie Curtsinger   Emery D. Berger

Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
{charlie,emery}@cs.umass.edu

2013 research paper

"the performance impact of -O3 over -O2 optimizations is indistinguishable from random noise."

Discussion

- What are some of the biggest improvements you’ve seen from compiler optimizations?

Discussion

- What are some of the biggest improvements you’ve seen from compiler optimizations?

- compiler optimizations are great at well-structured, regular loops and arrays

- Example: adding together two matrices

Discussion

- What kind of transforms on your code is the compiler allowed to do?

- many_add example

Discussion

- What kind of transforms on your code is the compiler allowed to do?

- many_add example

- Why did we get such a dramatic increase?

Discussion

- What kind of transforms on your code is the compiler allowed to do?

- many_add example

- Why did we get such a dramatic increase?
  - Programs must maintain their input/output behavior
  - Hard to precisely define (and still being discussed in C++ groups)
  - input/output can be files, volatile memory, console log, etc.

Discussion

- Extreme example

```
void foo(int * arr, int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++)
         for (j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1]) {
                tmp = arr[j];
                arr[j] = arr[j + 1]);
                arr[j + 1] = tmp;
            }
}
```

```
int p(int arr[], int start, int end)
{

    int pivot = arr[start];

    int count = 0;
    for (int i = start + 1; i <= end; i++) {
        if (arr[i] <= pivot)
            count++;
    }

    int pivotIndex = start + count;
    swap(arr[pivotIndex], arr[start]);

    int i = start, j = end;
    while (i < pivotIndex && j > pivotIndex) {

        while (arr[i] <= pivot) {
            i++;
        }

        while (arr[j] > pivot) {
            j--;
        }

        if (i < pivotIndex && j > pivotIndex) {
            swap(arr[i++], arr[j--]);
        }
    }

    return pivotIndex;
}

void foo(int *arr, int n)
{

    if (start >= end)
        return;

    int p = p(arr, m, n);

    foo(arr, start, p - 1);

    foo(arr, p + 1, end);
}
```

is this transform legal?

code from https://www.geeksforgeeks.org/

Discussion

- Extreme example

Yes this transform would be legal!

Could any compiler figure it out? currently unlikely..

This is a technique called “super optimizing” and it is getting more and more interest

bubble sort

```
void foo(int * arr, int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++)
         for (j = 0; j < n - i - 1; j++)
             if (arr[j] > arr[j + 1]) {
                 tmp = arr[j];
                 arr[j] = arr[j + 1]);
                 arr[j + 1] = tmp;
             }
}
```

```
int p(int arr[], int start, int end)
{

    int pivot = arr[start];

    int count = 0;
    for (int i = start + 1; i <= end; i++) {
        if (arr[i] <= pivot)
            count++;
    }

    int pivotIndex = start + count;
    swap(arr[pivotIndex], arr[start]);

    int i = start, j = end;
    while (i < pivotIndex && j > pivotIndex) {

        while (arr[i] <= pivot) {
            i++;
        }

        while (arr[j] > pivot) {
            j--;
        }

        if (i < pivotIndex && j > pivotIndex) {
            swap(arr[i++], arr[j--]);
        }
    }

    return pivotIndex;
}

void foo(int *arr, int n)
{

    if (start >= end)
        return;

    int p = p(arr, m, n);

    foo(arr, start, p - 1);

    foo(arr, p + 1, end);
}
```

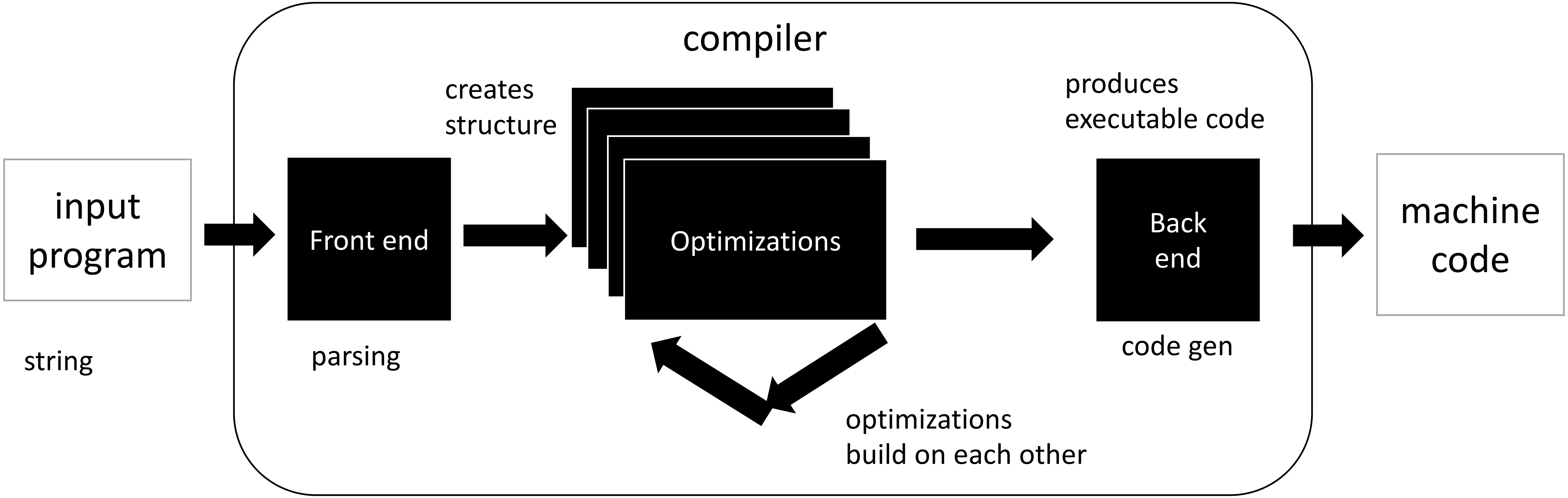
quick sort

is this transform legal?

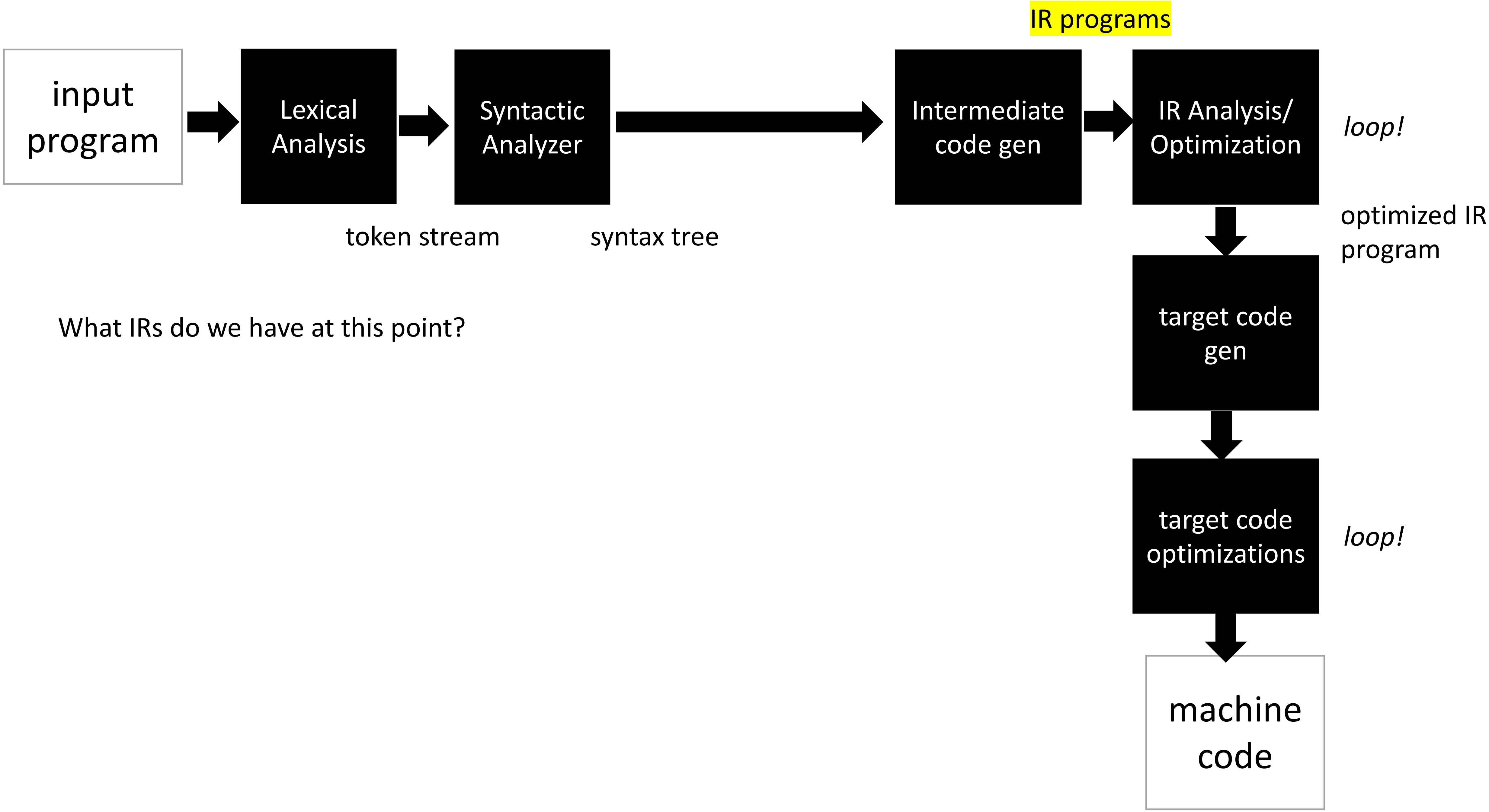
code from https://www.geeksforgeeks.org/

Moving on

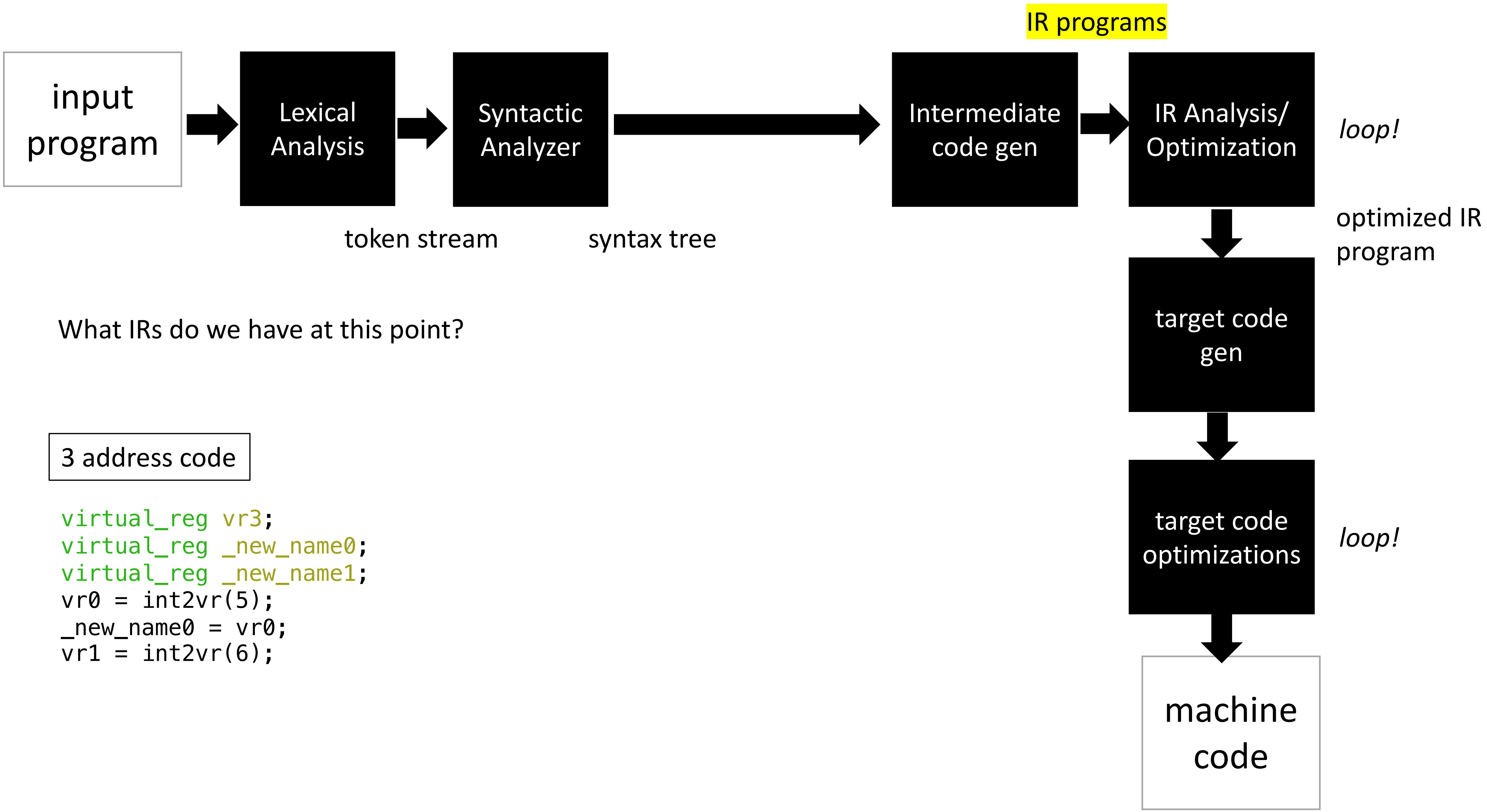
Zooming out again: Compiler Architecture



IRs and type inference type inference are at the boundary of parsing and optimizations



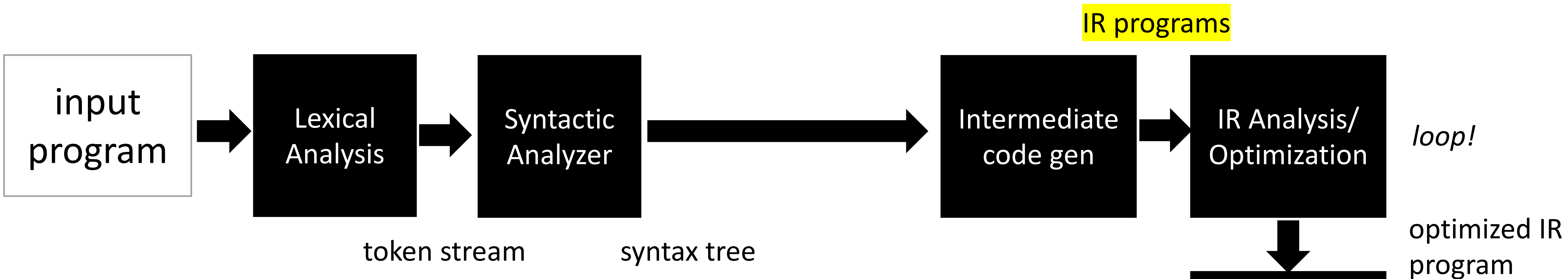
What IRs do we have at this point?

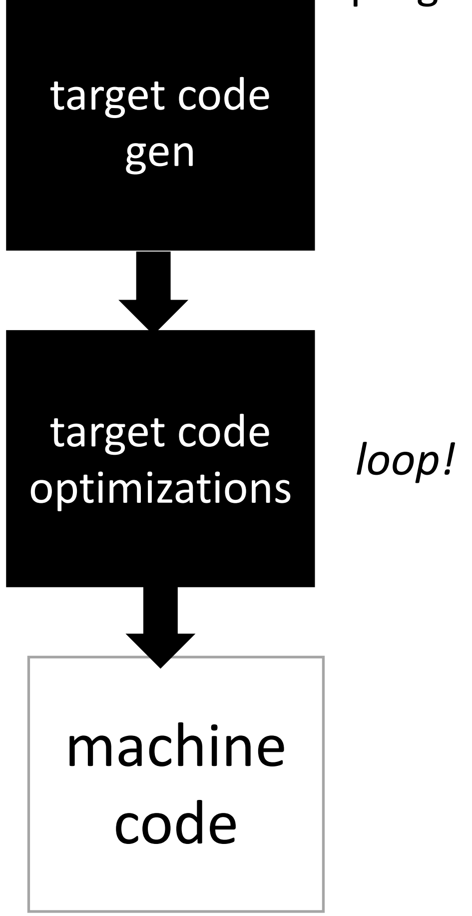


What IRs do we have at this point?

3 address code

```
virtual_reg vr3;
virtual_reg _new_name0;
virtual_reg _new_name1;
vr0 = int2vr(5);
_new_name0 = vr0;
vr1 = int2vr(6);
```



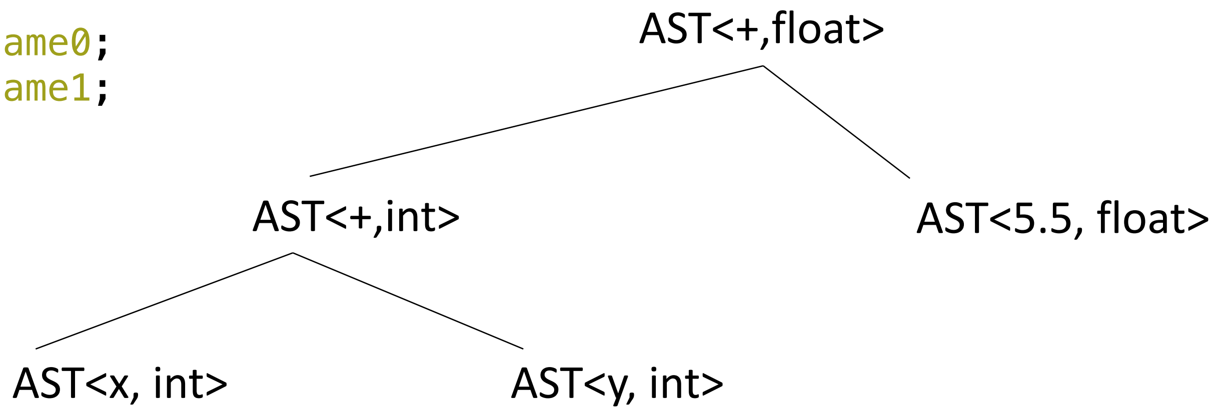


What IRs do we have at this point?

3 address code

```
virtual_reg vr3;
virtual_reg _new_name0;
virtual_reg _new_name1;
vr0 = int2vr(5);
_new_name0 = vr0;
vr1 = int2vr(6);
```

AST



Implicit parse tree

```
if_else_statement := IF LPAR expr RPAR statement ELSE statement
```

```
if (program0) {
  program1
}
else {
  program2
}
```

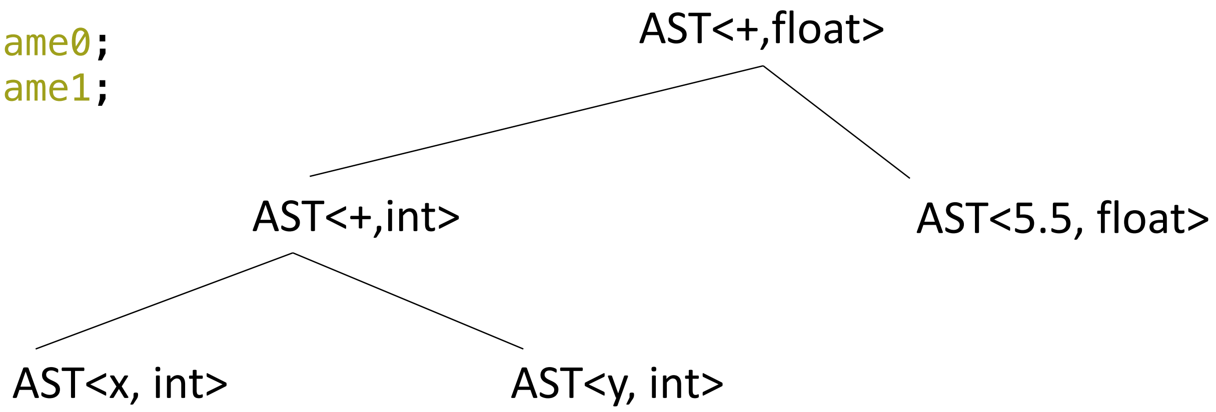
We have several structures to utilize to analyze and optimize programs!

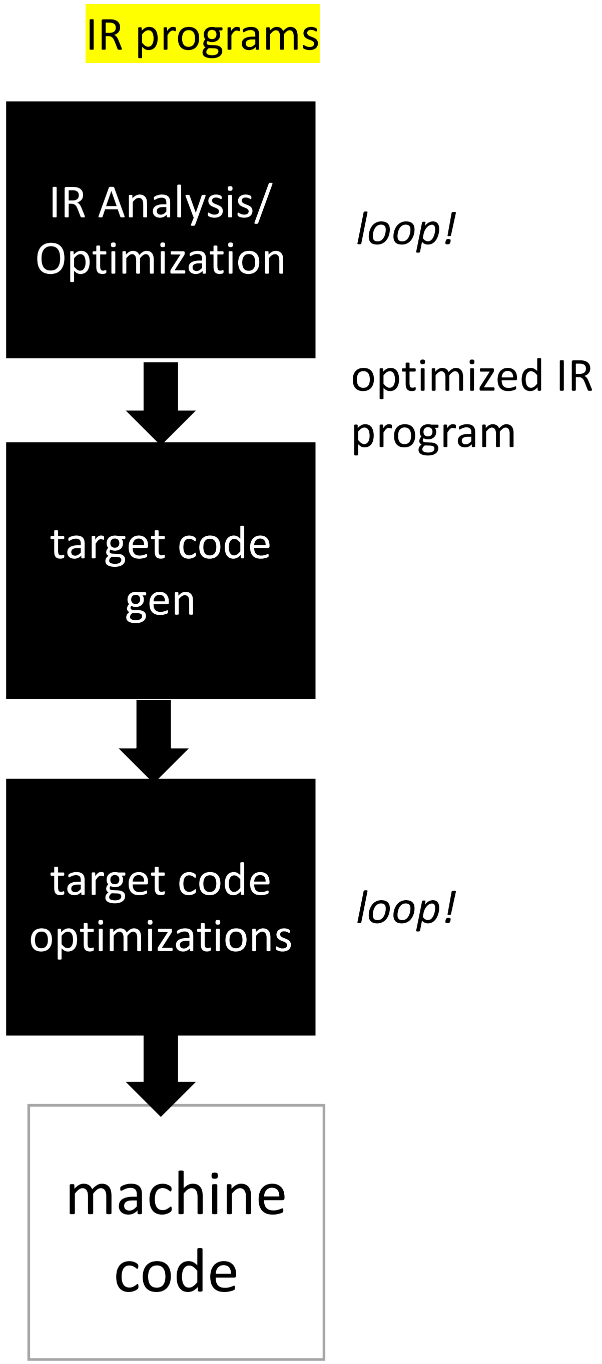
What IRs do we have at this point?

3 address code

```
virtual_reg vr3;
virtual_reg _new_name0;
virtual_reg _new_name1;
vr0 = int2vr(5);
_new_name0 = vr0;
vr1 = int2vr(6);
```

AST





Optimization categories

- Machine-independent - these optimizations should work well across many different systems
  - Examples?

- Machine dependent - these optimizations start to optimize the code for a given system
  - Examples?

Optimization categories

- Machine-independent - these optimizations should work well across many different systems
  - Examples?
  - All the examples we looked at before seem like they will help across many systems

- Machine dependent - these optimizations start to optimize the code for a given system
  - Examples?
  - loop chunking for cache line size and vectorization.
  - instruction re-orderings to take advantage of processor pipelines.
  - fused multiply-and-add instructions

Optimization categories

- **Machine-independent** - these optimizations should work well across many different systems
  - Examples?
  - All the examples we looked at before seem like they will help across many systems

- In this module we will be looking at machine-independent optimizations. Module 5 might start to look at others

- What are the pros of machine-independent optimizations?

Optimization categories

Next category level is how much code we need to reason about for the optimization.

- **local optimizations**: examine a ”basic block”, i.e. a small region of code with no control flow.
  - Examples?
- **Regional optimizations**: several basic blocks with simple control flow.
  - Examples?
- **Global optimization:** optimizes across an entire function

Optimization categories

- **local optimizations**: examine a ”basic block”, i.e. a small region of code with no control flow.
- **Regional optimizations**: several basic blocks with simple control flow
- **Global optimization:** optimizes across an entire function

Discussion:

- What are the pros and cons of each?
- Why don’t we go further than functions?

Optimization categories

- **local optimizations**: examine a ”basic block”, i.e. a small region of code with no control flow.
- **Regional optimizations**: several basic blocks with simple control flow
- **Global optimization:** optimizes across an entire function

For this module:

- We will look at two optimizations in detail:
- A local optimization: Local value numbering
- A regional optimization: Loop unrolling
- We will implement both as homework
- We will discuss several other optimizations and analysis

See everyone on Friday

- More about optimizations!