CSE110A: Compilers

May 13, 2022

Topics:

- *Finish intro to optimizations*
- *Basic blocks*
- *Local value numbering*

Announcements

- Pending grades
  - HW 2 (expect by Monday)
  - Midterm (expect by next Friday)

- HW 3 is released
  - Due in two weeks from release date
  - Get started early; you have all the material you need!
  - Packet updated, but nothing major

Quick homework demo

Quiz

Quiz

Write a simple grammar to parse functions like follows:

int main() { return 1 + 2; }

void foo(int a, double b) { a = a + 1; b = b + 1; return; }

You may use the format from your HW2 (like part 1.1), don't worry about left recursions, just the simple grammar. **You don't need to write the entire grammar**, just fill in the blanks.

You may use the same Tokens from HW2, and here are some new tokens

- RETURN   - keyword 'return'
- VOID   - keyword 'void'

Obviously, a function should have a **return type**, a **function name**, followed by a **list of arguments** enclosed by **parens.** Then followed by a **block of statements**. And for our case, assume the list of arguments is "0 or more declaration statements". You should extend the 'statement' to contain a **return_statement,** which may return an expression or may not.

Here is a template, **fill in the blanks:**

... <assume you have the reset of your grammar from HW2>...

function_decl := return_type ID LPARAN arg_list RPARAN block_stmt

arg_list := ___________

return_type := ___________

block_stmt := ___________

statements := assign_stmt | var_decl_stmt | if_else_stmt | for_stmt | block_stmt | return_stmt

return_stmt := ________

Quiz

Write a simple grammar to parse functions like follows:

int main() { return 1 + 2; }

void foo(int a, double b) { a = a + 1; b = b + 1; return; }

Here is a template, **fill in the blanks:**

... <assume you have the reset of your grammar from HW2>...

function_decl := return_type ID LPARAN arg_list RPARAN block_stmt

arg_list := ___________

return_type := ___________

block_stmt := ___________

statements := assign_stmt | var_decl_stmt | if_else_stmt | for_stmt | block_stmt | return_stmt

return_stmt := ________

Let's do the exercise

Quiz

We know loop unrolling will increase code size, but if we don't care about code sizes, we should unroll all loops so a program can execute faster.

- ○ True
- ○ False

Discussion

- Why might it not be a good idea?

Discussion

- Why might it not be a good idea?
  - Instruction cache
  - branch predictors
- In practice, compilers rarely unroll by more than 4 or 8.

Quiz

Inline functions are inlined after parsing AST and before emitting the final IR during the optimization phase.

- ○ True
- ○ False

Discussion

Quiz

It is the last lecture of Module 3; please let me know any feedback you might have about the module: e.g. what you enjoyed or what you think could be improved.

As always, thanks for your feedback!

Extra quiz question

- What would we need to do to extend our C-simple parse to handle if/else if/else statements?

Review

We started talking about compiler optimizations.

There's still much more to say, so let's pick up there.

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?

```
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

- 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

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

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

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;
            }
}
```

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

is this transform legal?

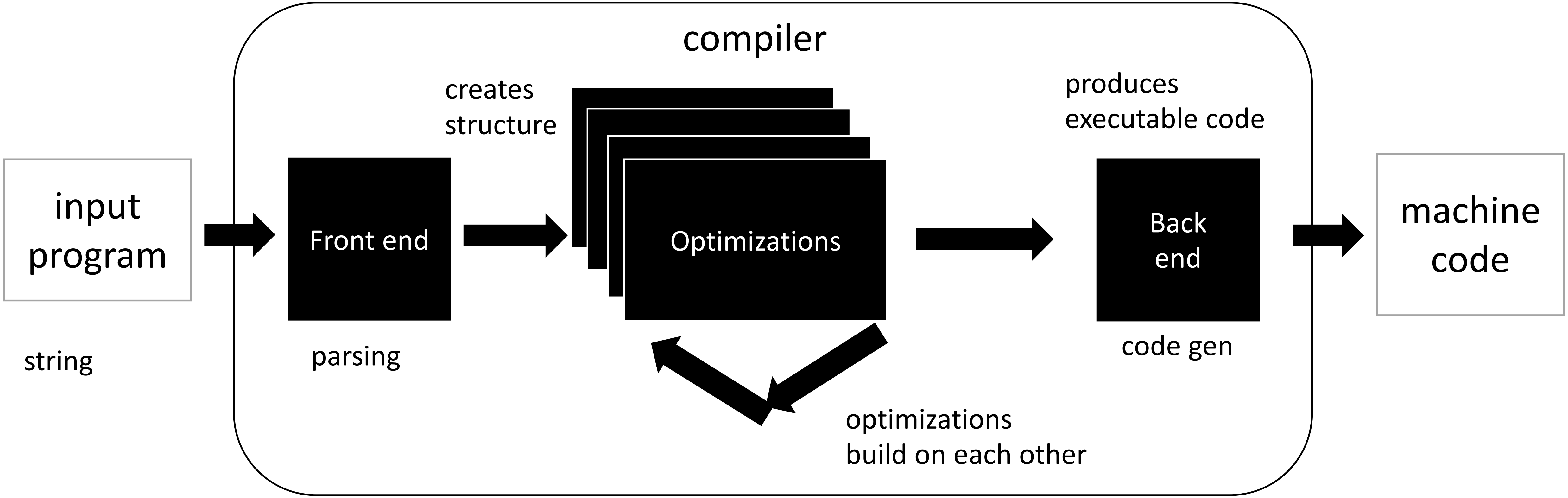
```
int p(int arr[], int start, int end)
{

    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

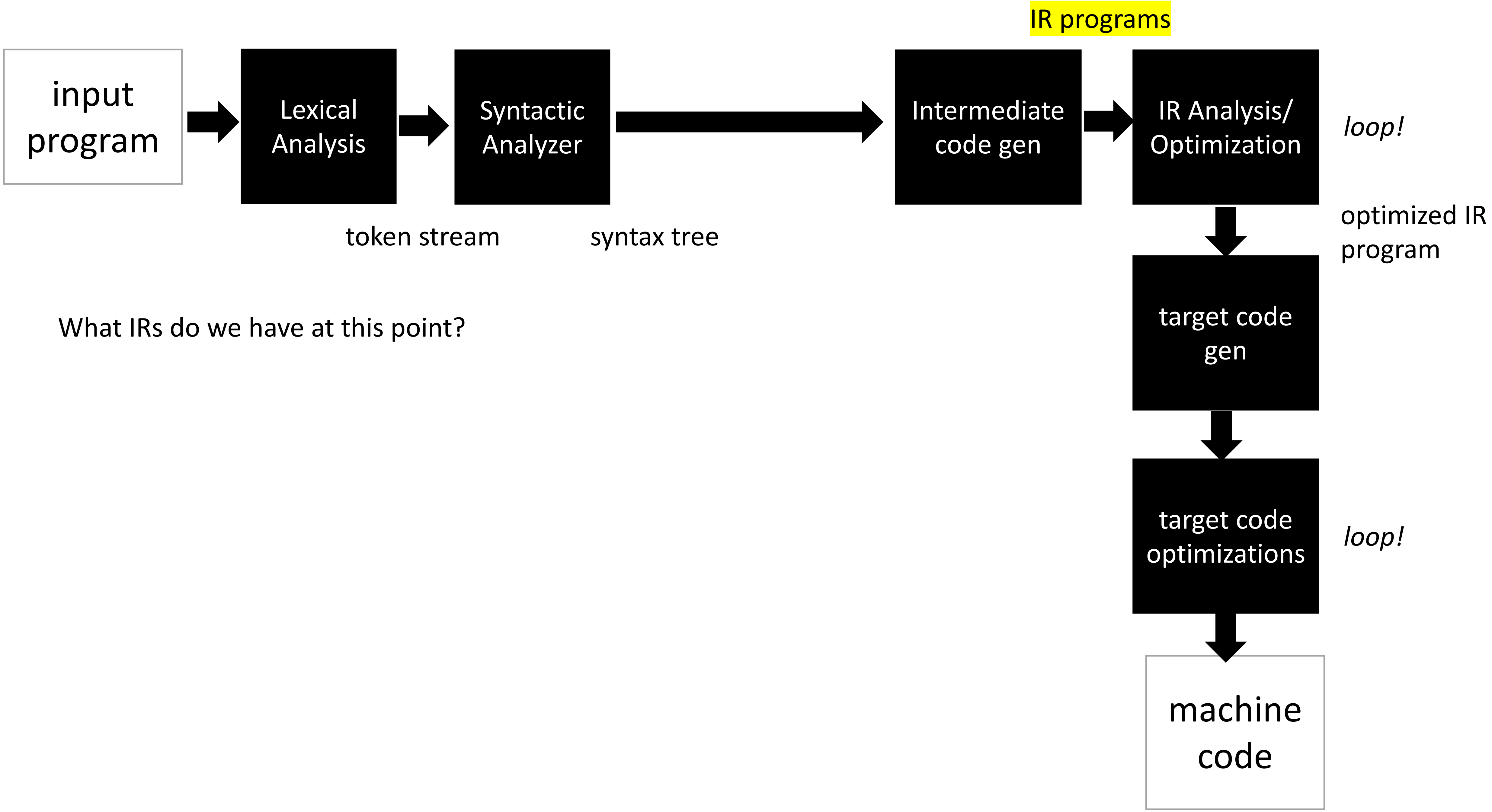
code from https://www.geeksforgeeks.org/

Moving on

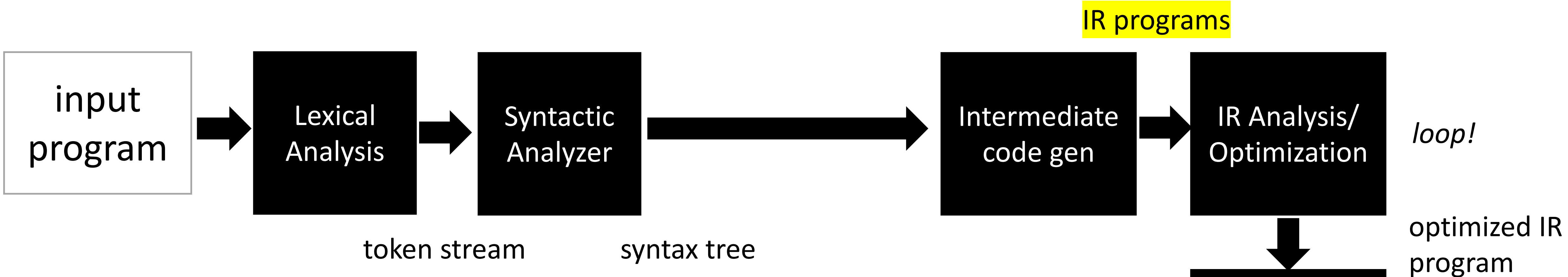
Zooming out again: Compiler Architecture

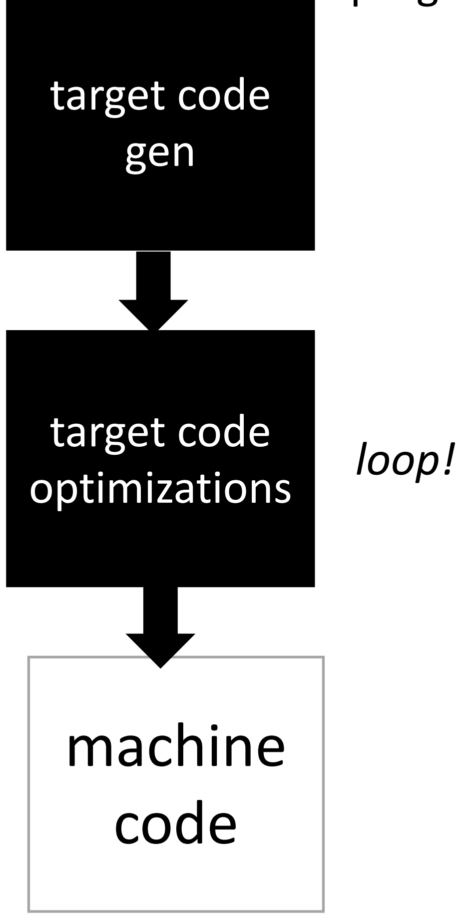


IRs and type inference type inference are at the boundary of parsing and optimizations



What IRs do we have at this point?

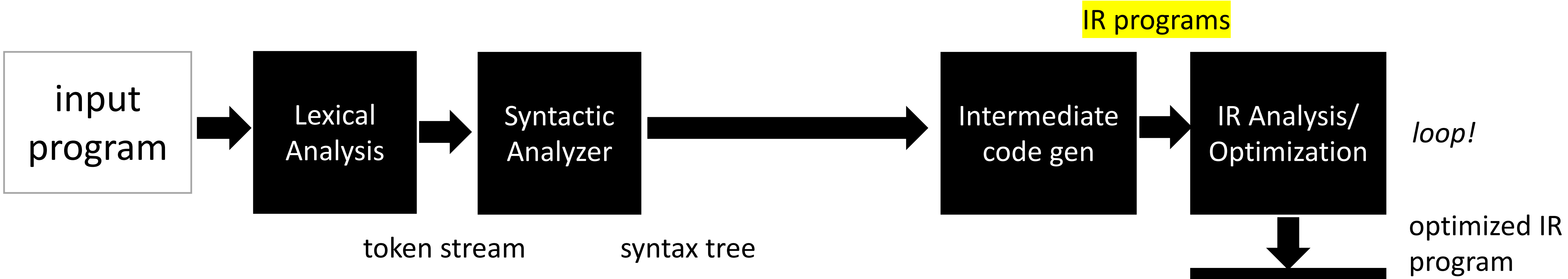


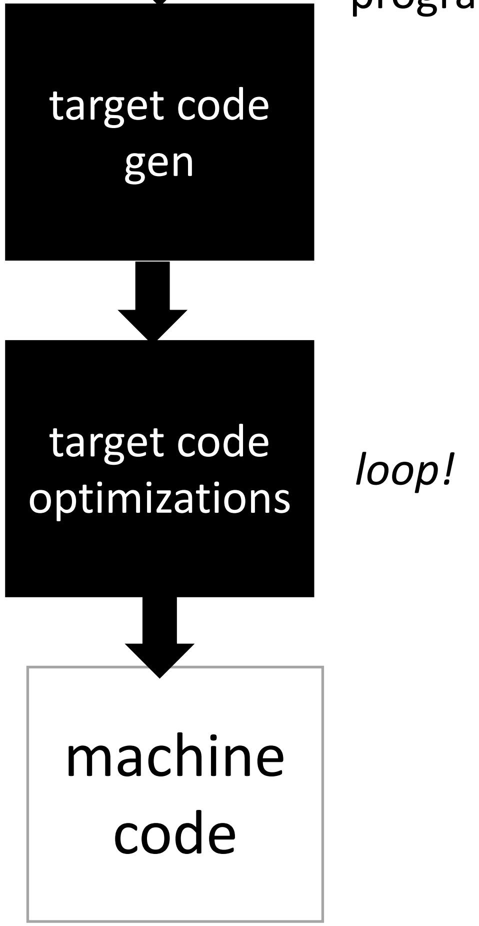


What IRs do we have at this point?

3 address code

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

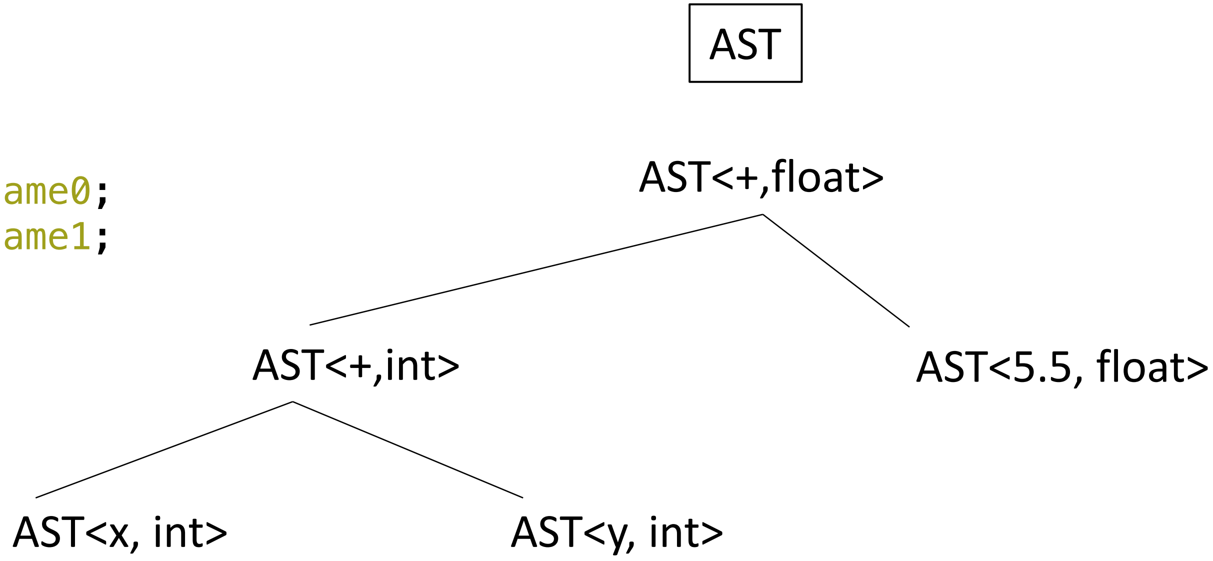




What IRs do we have at this point?

3 address code

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



Implicit parse tree

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

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

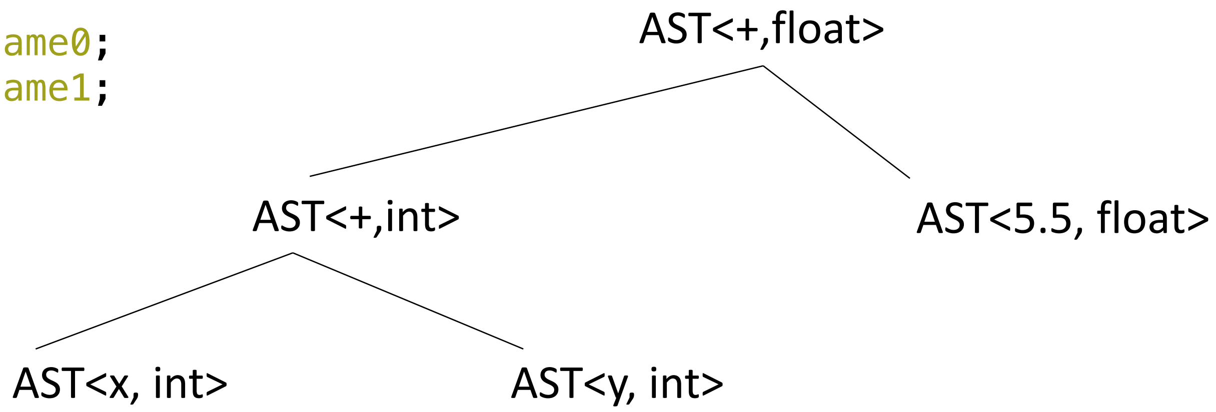
We have several structures to utilize to analyze and optimize programs!

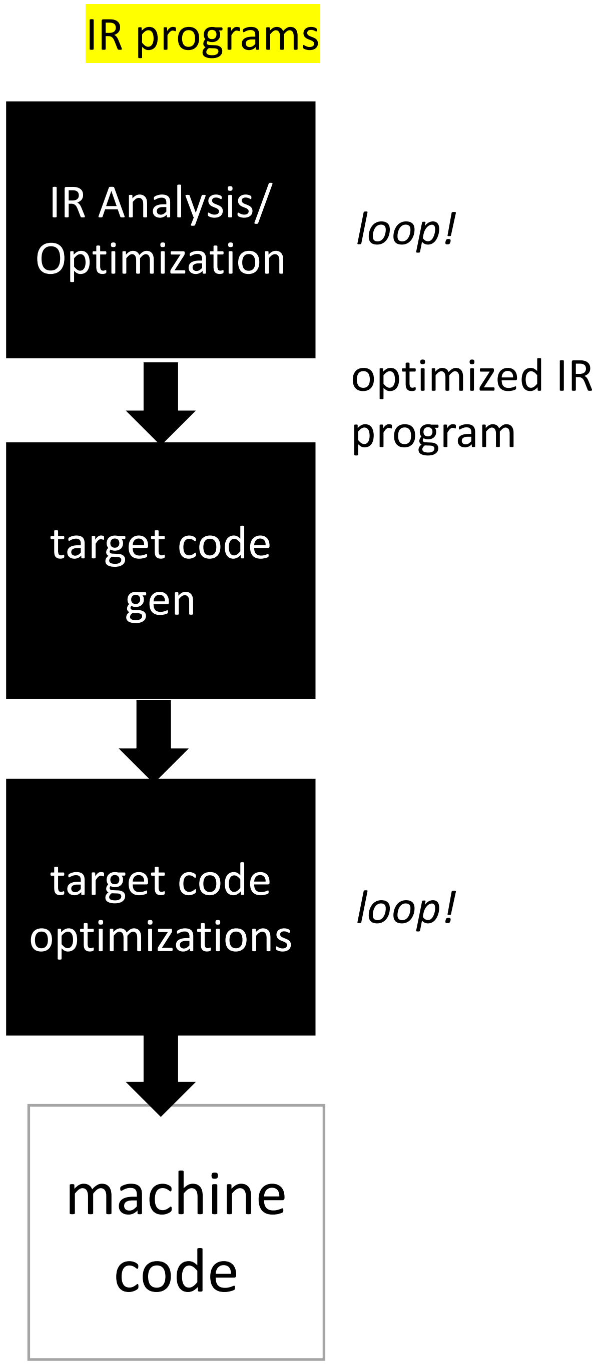
What IRs do we have at this point?

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

Basic blocks

IR Program structure

- A sequence of 3 address instructions

- Programs can be split into **Basic Blocks:**
  - A sequence of 3 address instructions such that:
  - There is a single entry, single exit

- *Important property*: an instruction in a basic block can assume that all preceding instructions will execute

Single Basic Block

```
Label_x:
op1;
op2;
op3;
br label_z;
```

IR Program structure

- A sequence of 3 address instructions

- Programs can be split into **Basic Blocks:**
  - A sequence of 3 address instructions such that:
  - There is a single entry, single exit

- *Important property*: an instruction in a basic block can assume that all preceding instructions will execute

Single Basic Block

```
Label_x:
op1;
op2;
op3;
br label_z;
```

Two Basic Blocks

```
Label_x:
op1;
op2;
op3;

Label_y:
op4;
op5;
```

IR Program structure

How might they appear in a high-level language? What are some examples?

- A sequence of 3 address instructions
- Programs can be split into **Basic Blocks:**
  - A sequence of 3 address instructions such that:
  - There is a single entry, single exit
- *Important property*: an instruction in a basic block can assume that all preceding instructions will execute

Single Basic Block

```
Label_x:
op1;
op2;
op3;
br label_z;
```

Two Basic Blocks

```
Label_x:
op1;
op2;
op3;

Label_y:
op4;
op5;
```

IR Program structure

- A sequence of 3 address instructions

- Programs can be split into **Basic Blocks:**
  - A sequence of 3 address instructions such that:
  - There is a single entry, single exit

- *Important property*: an instruction in a basic block can assume that all preceding instructions will execute

How might they appear in a high-level language?

How many basic blocks?

```
…
if (x) {
    …
}
else {
   …
}
…
```

Single Basic Block

```
Label_x:
op1;
op2;
op3;
br label_z;
```

Two Basic Blocks

```
Label_x:
op1;
op2;
op3;

Label_y:
op4;
op5;
```

Converting 3 address code into basic blocks

- Let’s try an example: test 4 in HW 3:

Converting 3 address code into basic blocks

- Simple algorithm:
  - keep a list of basic blocks
  - a basic block is a list of instructions

  - Iterate over the 3 address instructions
  - if you see a branch or a label, finalize the current basic block and start a new one.

Converting 3 address code into basic blocks

pseudo code

```
basic_blocks = []
bb = []
for instr in program:
    if instr type is in [branch, label]:
        bb.append(instr)
        basic_blocks.append[bb]
        bb = []
    else:
        bb.append(instr)
```

Optimization levels

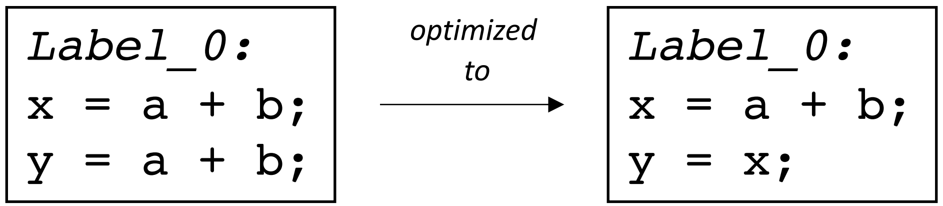
- **Local optimizations:**
  - Optimizes an individual basic block
- **Regional optimizations:**
  - Combines several basic blocks
- **Global optimizations:**
  - operates across an entire procedure
  - what about across procedures?

Optimization levels

```
Label_0:
x = a + b;
y = a + b;
```

- **Local optimizations:**
  - Optimizes an individual basic block
- **Regional optimizations:**
  - Combines several basic blocks
- **Global optimizations:**
  - operates across an entire procedure
  - what about across procedures?

Optimization levels



- **Local optimizations:**
  - Optimizes an individual basic block
- **Regional optimizations:**
  - Combines several basic blocks
- **Global optimizations:**
  - operates across an entire procedure
  - what about across procedures?

Optimization levels

- **Local optimizations:**
  - Optimizes an individual basic block
- **Regional optimizations:**
  - Combines several basic blocks
- **Global optimizations:**
  - operates across an entire procedure
  - what about across procedures?

```
Label_0:
x = a + b;
y = a + b;
```

optimized to →

```
Label_0:
x = a + b;
y = x;
```

```
Label_0:
x = a + b;

Label_1:
y = a + b;
```

CANNOT always optimized to →

```
Label_0:
x = a + b;

Label_1:
y = x;
```

Optimization levels

- **Local optimizations:**
  - Optimizes an individual basic block
- **Regional optimizations:**
  - Combines several basic blocks
- **Global optimizations:**
  - operates across an entire procedure
  - what about across procedures?

```
Label_0:
x = a + b;
y = a + b;
```

optimized to →

```
Label_0:
x = a + b;
y = x;
```

```
Label_0:
x = a + b;

Label_1:
y = a + b;
```

CANNOT always optimized to →

```
Label_0:
x = a + b;

Label_1:
y = x;
```

code could skip Label_0, leaving x undefined!

```
br Label_1;

Label_0:
x = a + b;

Label_1:
y = a + b;
```

Regional Optimization

```
…
if (x) {
   …
}
else {
  x = a + b;
}
y = a + b;
…
```

we cannot replace:
y = a + b.
with
y = x;

Regional Optimization

```
…
if (x) {
   …
}
else {
  x = a + b;
}
y = a + b;
…
```

we cannot replace:
y = a + b.
with
y = x;

```
x = a + b;
if (x) {
   …
}
else {
  …
}
y = a + b;
…
```

But in this case, we can check if a and b are not redefined, then
y = a + b;
can be replaced with
y = x;

This requires regional analysis and optimizations

See everyone on Monday

- A concrete optimization: local value numbering