The Middle-End

15-411/15-611 Compiler Design

Seth Copen Goldstein

September 22, 2020

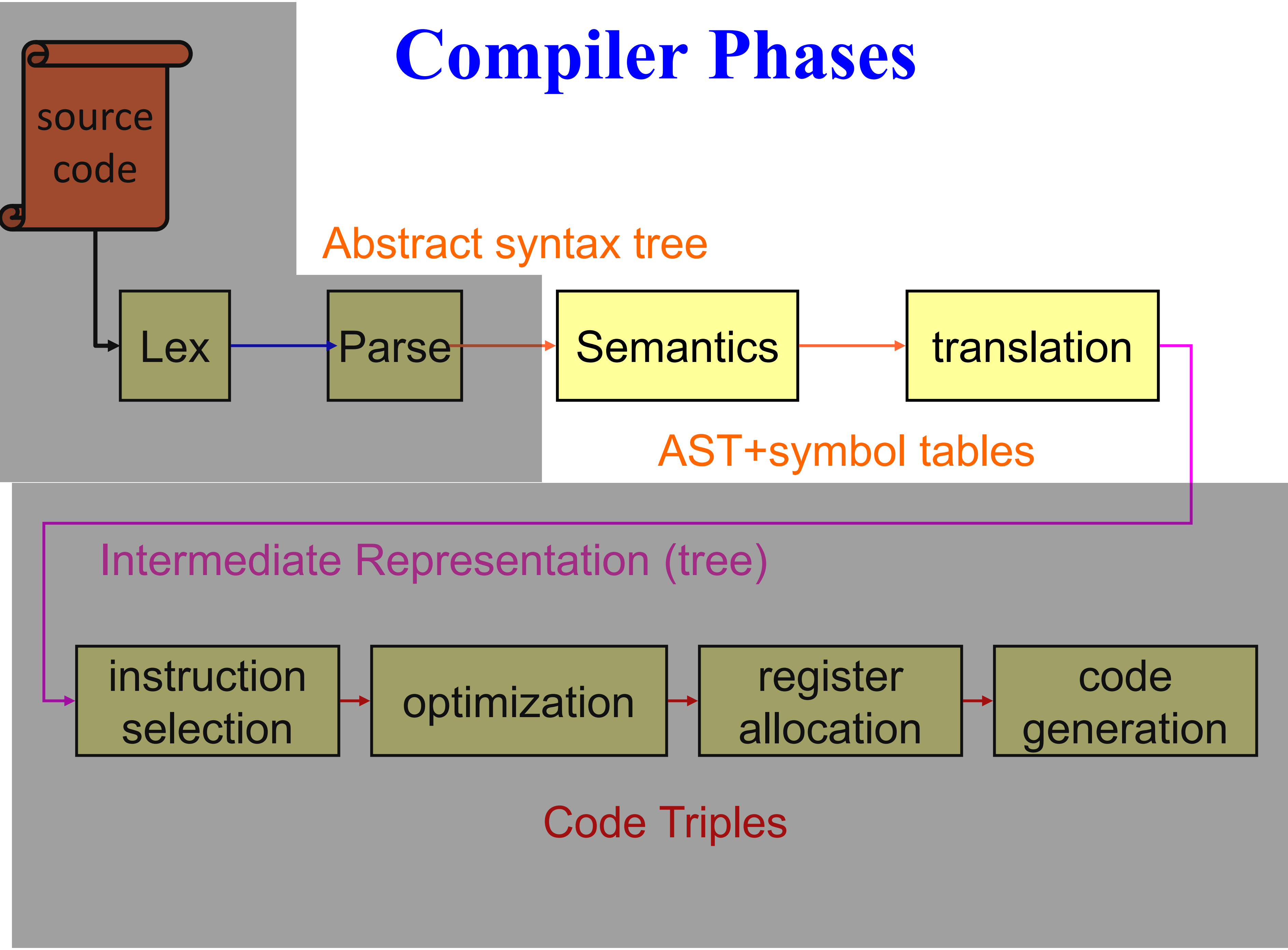
Today

- lab2
- Elaboration
- Static Semantics
- Translating to IR

L2

```
⟨program⟩   ::=  int main () ⟨block⟩
⟨block⟩     ::=  { ⟨stmts⟩ }
⟨type⟩      ::=  int | bool
⟨decl⟩      ::=  ⟨type⟩ ident | ⟨type⟩ ident = ⟨exp⟩
⟨stmts⟩     ::=  ϵ | ⟨stmt⟩ ⟨stmts⟩
⟨stmt⟩      ::=  ⟨simp⟩ ; | ⟨control⟩ | ⟨block⟩
⟨simp⟩      ::=  ⟨lvalue⟩ ⟨asop⟩ ⟨exp⟩ | ⟨lvalue⟩ ⟨postop⟩ | ⟨decl⟩ | ⟨exp⟩
⟨simpopt⟩   ::=  ϵ | ⟨simp⟩
⟨lvalue⟩    ::=  ident | ( ⟨lvalue⟩ )
⟨elseopt⟩   ::=  ϵ | else ⟨stmt⟩
⟨control⟩   ::=  if ( ⟨exp⟩ ) ⟨stmt⟩ ⟨elseopt⟩
              |  while ( ⟨exp⟩ ) ⟨stmt⟩
              |  for ( ⟨simpopt⟩ ; ⟨exp⟩ ; ⟨simpopt⟩ ) ⟨stmt⟩
              |  return ⟨exp⟩ ;
⟨exp⟩       ::=  ( ⟨exp⟩ ) | ⟨intconst⟩ | true | false | ident
              |  ⟨unop⟩ ⟨exp⟩ | ⟨exp⟩ ⟨binop⟩ ⟨exp⟩ | ⟨exp⟩ ? ⟨exp⟩ : ⟨exp⟩
⟨intconst⟩  ::=  num
⟨asop⟩      ::=  = | += | -= | *= | /= | %= | &= | ^= | |= | <<= | >>=
⟨binop⟩     ::=  + | - | * | / | % | < | <= | > | >= | == | !=
              |  && | || | & | ^ | | | << | >>
⟨unop⟩      ::=  ! | ~ | -
⟨postop⟩    ::=  ++ | --
```

Compiler Phases



Elaboration

- Eliminate syntactic sugar
- Simplify future analysis
- For example:
  - `for (init; test; incr) stmt`
  - `while (test) stmt`
  - `expr && expr`
  - `expr || expr`
  - others?

for loop

```
for (init; test; incr) stmt
 ⇒ {
      init;
      while (test) { stmt; incr; }
   }
```

X && Y

```
exp1 && exp2
```

$\Rightarrow$

```
exp1 || exp2
```

$\Rightarrow$

X && Y

```
exp1 && exp2
 ⇒ exp1 ? exp2 : false
exp1 || exp2
 ⇒ exp1 ? true : exp2
```

When?

- When to do elaboration?
  - While parsing?

```
stmt :=    for ( simpstmt ; expr; simpstmt ) stmt
           {
                $$ = new Block( );
                $$->append($3);
                Block body = new Block();
                body->append($9);
                body->append($7);
                $$->append(new While($5, body));
           }
```

  - As a separate pass, after parsing?

What?

- Absolutely: **`for`**, **`&&`**, **`||`**
- What about: **`int x;`**
  - What would we elaborate it to?
  - Why would this be good? Bad?

- Other things to keep in mind:
  - line numbers
  - errors

Now ready to goto IR?

- Many choices of IR (discussed in lecture 2)
  - I chose tree-IR and Triples
- Before converting to IR: Semantic Analysis

Semantic Analysis

- Semantic analysis is a static analysis of the program to make sure it has a meaning
- It is a context sensitive analysis!
- At this point in the compilation we have an AST of the input program
  i.e., we know it is syntactically correct
- What kinds of checks are needed to ensure a semantically correct program?

Semantic Analysis

- Type checks
  - Is variable **`x`** declared?
  - What is its type?
  - Can an operator operate on a particular type?
  - What is the result type of an operation?
- Control flow checks
  - Is the placement of a **`break`** or **`continue`** legal?
  - Is the placement of a **`return`** legal?

Semantic Analysis

- Uniqueness checks
  - Is a variable declared more than once?
  - Are the labels in a switch unique?
  - Are the labels in a procedure legal?
  - Are the field names in a record unique?
- Matching Name checks
  - E.g., in ada loops can have names at start and end and they must be the same

Semantic Analysis

- Static analysis:
  - Type checks
  - Control flow checks
  - Uniqueness checks
  - Matching Name checks
- As opposed to dynamic analysis:
  - dereferencing a null pointer
  - array bounds checks
  - infinite loops
- Why do we defer the static checks til now?

The easy cases

- Control flow checks
- Matching names
- Uniqueness?

The easy cases

- Control flow checks
  - recursively walk AST keeping track of loop depth.
  - If break or continue encountered, then depth == 0 $\Rightarrow$ error.
- Matching names
- Uniqueness?

The easy cases

- Control flow checks
  - recursively walk AST keeping track of loop depth.
  - If break or continue encountered, then depth == 0 $\Rightarrow$ error.
- Matching names
  - recursive walk of tree keep track of “opening” name and then match to “closing” name.
- Uniqueness?

Uniqueness

- These questions are harder:
  - Is a variable declared more than once?
  - Are the labels in a switch unique?
  - Are the labels in a procedure legal?
  - Are the field names in a record unique?
- When is a variable declared more than once?

```
int foo(int a) {
        int a;
        for (i=0; i<100; i++) {
                int a = i*i;
                …
        }
}
```

- In checking types and declarations we must take scope into account.

Scope

- Declarations associate information with names
  – a variable name to its type, storage, etc.
  – a type name to a particular type
  – a function name to its parameter list, body, etc.
- The scope rules of a language determine the extent that the declaration is valid

or

- They determine which declaration applies to a name at a given place in the program

Different Kinds of Scope Rules

- C like
  - static/lexical scoping
  - global, static, local, block (most closely nested)
- Pascal
  - local, block
  - nested procedures
- Java
  - global, package, file, class, method, block
- Lisp
  - dynamic scope

Example of nesting

```
int f(int b) {
  b = 0;
  { int b = 1; int c = 1;
      {int b = 2; int c = 2;
           …
      }
      {int b = 3; … c …
      }
    …
  }
  …
}
```

Not legal c0!

Dynamic V. Static Scope

```
void weird() {
     int N = 1;
    void  show() {
         print(N); print(" ")   }
    void two() {
        int N = 2;
        show();
     }
    show(); two(); show(); two();
}
```

Static scope: "1 1 1 1 "

Dynamic scope: "1 2 1 2 "

Symbol Tables

- Symbol tables are key data structure for semantic analysis
- A symbol table maps identifiers to attributes
  - its type
  - its location on stack
  - its register name if any
  - storage class
  - offset from base of record
  - etc.
- Structure of symbol table(s) must reflect scope of program
- It must be efficient
- Support multiple name spaces

Symbol Tables

- Two main choices:
  - A Stack of tables:
    - entering a scope: create new table, link to parent
    - leaving a scope: remove table
  - Table of stacks
    - one symbol table
    - A stack for variables pointing to entry in table
    - On leaving scope, remove all variables declared in current scope
- Where do we store information, e.g., type, …

Rewrite AST

- When we insert a new entry, attach attribute information to decl node
- When we lookup a name, point to the decl node to which it maps.
- When we are done with this pass the symbol table is no longer needed!

Semantic Analysis

- Type checks
  - Is variable **x** declared?
  - What is its type?
  - Can an operator operate on a particular type?
  - What is the result type of an operation?
- Control flow checks
- Uniqueness checks
  - Is a variable declared more than once?
  - Are the labels in a switch unique?
  - Are the labels in a procedure legal?
  - Are the field names in a record unique?
- Matching Name checks

Type Checking

- Ensures that type of an expression is valid in the context in which it appears.
- For example:
  - arguments to + are integers
  - index operation is applied to arrays
  - that '.' is applied to records
  - function call has proper number of args (and they are of proper type)
  - casts are legal

What is a Type?

- A type describes a class of values.
- So far in C0
  - int: class of integers
  - bool: true or false
  - More coming soon
- Two kinds of declarations:
  - Type declarations create new types from other types.
  - Variable declarations specify that a variable will always have a particular type.

What does decl of x tell us

- From the type:
  - Know what kinds of values are stored in x
  - Know what kinds of operations are legal
    - +,-,*, …
    - Function call: # of args, return type
  - How big x is
- From the scope:
  - Where it is stored
  - How it is allocated, inited
  - How long it should be kept around

Type Checking

- Build up an environment which maps
  - variables to type
  - values to types
  - expressions to types
- Given an environment and an expression
  - check that it is correct
  - update the environment
- Do this on entire program
- This is a syntax directed analysis, i.e., recursively walk ast checking types as we go.

Approaches to Semantic Analysis

- Ad hoc, e.g., tree-walk to make sure all control-flow paths end in a return
- Attribute grammars: Use a grammar to automatically generate an analysis pass
- Inference rules, judgements and solvers

Using Inference Rules

- Our language:

```
e := n | x | e1+e2 | e1 && e2
s := x←e
  |  if(e,s1,s2)
  |  while(e,s)
  |  return(e)
  |  seq(s1,s2)
  |  decl(x,τ,s)
```

Check for Proper Returns

$$\frac{}{\text{hasret}(\texttt{return(e)})}$$

$$\frac{\text{hasret}(\texttt{s1})}{\text{hasret}(\texttt{seq(s1,s2)})} \qquad \frac{\text{hasret}(\texttt{s2})}{\text{hasret}(\texttt{seq(s1,s2)})}$$

decl?

if?

while?

nop?

assign?

Check for Proper Returns

$$\frac{}{\text{hasret}(\mathtt{return(e)})}$$

$$\frac{\text{hasret}(\mathtt{s1})}{\text{hasret}(\mathtt{seq(s1,s2)})} \qquad \frac{\text{hasret}(\mathtt{s2})}{\text{hasret}(\mathtt{seq(s1,s2)})}$$

$$\frac{\text{hasret}(\mathtt{s})}{\text{hasret}(\mathtt{decl(x,\tau,s)})} \qquad \frac{\text{hasret}(\mathtt{s1}) \quad \text{hasret}(\mathtt{s2})}{\text{hasret}(\mathtt{if(e,s1,s2)})}$$

Iplementation

$$\frac{}{\text{hasret}(\texttt{return(e)})}$$

$$\frac{\text{hasret}(\texttt{s1})}{\text{hasret}(\texttt{seq(s1,s2)})}$$

$$\frac{\text{hasret}(\texttt{s2})}{\text{hasret}(\texttt{seq(s1,s2)})}$$

$$\frac{\text{hasret}(\texttt{s})}{\text{hasret}(\texttt{decl}(\texttt{x},\tau,\texttt{s}))}$$

$$\frac{\text{hasret}(\texttt{s1}) \quad \text{hasret}(\texttt{s2})}{\text{hasret}(\texttt{if(e,s1,s2)})}$$

A recursive treewalk using judgements as cases.

hasret(**return(e)**) = true
hasret(**seq(s1,s2)**) = hasret(**s1**) || hasret(**s2**)
hasret(**decl(x,τ,s)**) = hasret(**s**)
hasret(**if(e,s1,s2))** = hasret(**s1**)&&hasret(**s2**)
hasret(**while(e,s)**) = false
....

Initialization Checking

- How do we make sure all variables are initialized before they are used?

```
e:=n | x | e1+e2 | e1 && e2
s:=x←e
  | nop
  | if(e,s1,s2)
  | while(e,s)
  | return(e)
  | seq(s1,s2)
  | decl(x,τ,s)
```

the init predicate

$$\frac{}{\mathsf{init}(\mathsf{nop})} \qquad \frac{\mathsf{init}(s_1) \quad \mathsf{init}(s_2)}{\mathsf{init}(\mathsf{seq}(s_1, s_2))}$$

$$\frac{\mathsf{init}(s) \quad \neg\mathsf{live}(s, x)}{\mathsf{init}(\mathsf{decl}(x, \tau, s))}$$

live predicate (take 1)

$$\frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{assign}(y, e), x)}$$

the use predicate

no rule for $\mathsf{use}(n, x)$

$$\frac{}{\mathsf{use}(x, x)}$$

no rule for $\mathsf{use}(y, x), y \neq x$

$$\frac{\mathsf{use}(e_1, x)}{\mathsf{use}(e_1 \oplus e_2, x)} \qquad \frac{\mathsf{use}(e_2, x)}{\mathsf{use}(e_1 \oplus e_2, x)}$$

$$\frac{\mathsf{use}(e_1, x)}{\mathsf{use}(e_1 \,\&\&\, e_2, x)} \qquad \frac{\mathsf{use}(e_2, x)}{\mathsf{use}(e_1 \,\&\&\, e_2, x)}$$

live predicate (take 2)

$$\frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{assign}(y, e), x)}$$

$$\frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{if}(e, s_1, s_2), x)} \qquad \frac{\mathsf{live}(s_1, x)}{\mathsf{live}(\mathsf{if}(e, s_1, s_2), x)} \qquad \frac{\mathsf{live}(s_2, x)}{\mathsf{live}(\mathsf{if}(e, s_1, s_2), x)}$$

$$\frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{while}(e, s), x)} \qquad \frac{\mathsf{live}(s, x)}{\mathsf{live}(\mathsf{while}(e, s), x)}$$

$$\frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{return}(e), x)}$$

no rule for $\mathsf{live}(\mathsf{nop}, x)$

$$\frac{\mathsf{live}(x, s) \quad y \neq x}{\mathsf{live}(\mathsf{decl}(y, \tau, s), x)}$$

$$\frac{\mathsf{live}(s_1, x)}{\mathsf{live}(\mathsf{seq}(s_1, s_2), x)} \qquad \frac{\neg\mathsf{def}(s_1, x) \quad \mathsf{live}(s_2, x)}{\mathsf{live}(\mathsf{seq}(s_1, s_2), x)}$$

live predicate (take 2)

$$\frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{assign}(y, e), x)}$$

$$\frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{if}(e, s_1, s_2), x)} \qquad \frac{\mathsf{live}(s_1, x)}{\mathsf{live}(\mathsf{if}(e, s_1, s_2), x)} \qquad \frac{\mathsf{live}(s_2, x)}{\mathsf{live}(\mathsf{if}(e, s_1, s_2), x)}$$

$$\frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{while}(e, s), x)} \qquad \frac{\mathsf{live}(s, x)}{\mathsf{live}(\mathsf{while}(e, s), x)}$$

$$\frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{return}(e), x)}$$

no rule for $\mathsf{live}(\mathsf{nop}, x)$

$$\boxed{\frac{\mathsf{live}(x, s) \quad y \neq x}{\mathsf{live}(\mathsf{decl}(y, \tau, s), x)}}$$

$$\frac{\mathsf{live}(s_1, x)}{\mathsf{live}(\mathsf{seq}(s_1, s_2), x)} \qquad \frac{\neg\mathsf{def}(s_1, x) \quad \mathsf{live}(s_2, x)}{\mathsf{live}(\mathsf{seq}(s_1, s_2), x)}$$

live predicate (take 2)

$$\frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{assign}(y, e), x)}$$

$$\frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{if}(e, s_1, s_2), x)} \qquad \frac{\mathsf{live}(s_1, x)}{\mathsf{live}(\mathsf{if}(e, s_1, s_2), x)} \qquad \frac{\mathsf{live}(s_2, x)}{\mathsf{live}(\mathsf{if}(e, s_1, s_2), x)}$$

$$\frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{while}(e, s), x)} \qquad \frac{\mathsf{live}(s, x)}{\mathsf{live}(\mathsf{while}(e, s), x)}$$

$$\frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{return}(e), x)} \qquad \begin{array}{c}\text{no rule for}\\ \mathsf{live}(\mathsf{nop}, x)\end{array} \qquad \frac{\mathsf{live}(x, s) \quad y \neq x}{\mathsf{live}(\mathsf{decl}(y, \tau, s), x)}$$

$$\frac{\mathsf{live}(s_1, x)}{\mathsf{live}(\mathsf{seq}(s_1, s_2), x)} \qquad \boxed{\frac{\neg\mathsf{def}(s_1, x) \quad \mathsf{live}(s_2, x)}{\mathsf{live}(\mathsf{seq}(s_1, s_2), x)}}$$

the def predicate

$$\frac{}{\mathsf{def}(\mathsf{assign}(x,e),x)}$$

no rule for
$\mathsf{def}(\mathsf{assign}(y,e),x), y \neq x$

$$\frac{\mathsf{def}(s_1,x) \quad \mathsf{def}(s_2,x)}{\mathsf{def}(\mathsf{if}(e,s_1,s_2),x)}$$

no rule for
$\mathsf{def}(\mathsf{while}(e,s),x)$

no rule for
$\mathsf{def}(\mathsf{nop},x)$

$$\frac{\mathsf{def}(s_1,x)}{\mathsf{def}(\mathsf{seq}(s_1,s_2),x)}$$

$$\frac{\mathsf{def}(s_2,x)}{\mathsf{def}(\mathsf{seq}(s_1,s_2),x)}$$

$$\frac{\mathsf{def}(s,x) \quad y \neq x}{\mathsf{def}(\mathsf{decl}(y,\tau,s),x)}$$

$$\frac{}{\mathsf{def}(\mathsf{return}(e),x)}$$

the def predicate

$$\frac{}{\mathsf{def}(\mathsf{assign}(x,e),x)}$$

no rule for
$\mathsf{def}(\mathsf{assign}(y,e),x), y \neq x$

$$\frac{\mathsf{def}(s_1,x) \quad \mathsf{def}(s_2,x)}{\mathsf{def}(\mathsf{if}(e,s_1,s_2),x)}$$

no rule for
$\mathsf{def}(\mathsf{while}(e,s),x)$

no rule for
$\mathsf{def}(\mathsf{nop},x)$

$$\frac{\mathsf{def}(s_1,x)}{\mathsf{def}(\mathsf{seq}(s_1,s_2),x)}$$

$$\frac{\mathsf{def}(s_2,x)}{\mathsf{def}(\mathsf{seq}(s_1,s_2),x)}$$

$$\boxed{\frac{\mathsf{def}(s,x) \quad y \neq x}{\mathsf{def}(\mathsf{decl}(y,\tau,s),x)}}$$

$$\frac{}{\mathsf{def}(\mathsf{return}(e),x)}$$

the def predicate

$$\frac{}{\mathsf{def}(\mathsf{assign}(x,e),x)}$$

no rule for
$\mathsf{def}(\mathsf{assign}(y,e),x), y \neq x$

$$\frac{\mathsf{def}(s_1,x) \quad \mathsf{def}(s_2,x)}{\mathsf{def}(\mathsf{if}(e,s_1,s_2),x)}$$

no rule for
$\mathsf{def}(\mathsf{while}(e,s),x)$

no rule for
$\mathsf{def}(\mathsf{nop},x)$

$$\frac{\mathsf{def}(s_1,x)}{\mathsf{def}(\mathsf{seq}(s_1,s_2),x)} \qquad \frac{\mathsf{def}(s_2,x)}{\mathsf{def}(\mathsf{seq}(s_1,s_2),x)}$$

$$\frac{\mathsf{def}(s,x) \quad y \neq x}{\mathsf{def}(\mathsf{decl}(y,\tau,s),x)}$$

$$\boxed{\frac{}{\mathsf{def}(\mathsf{return}(e),x)}}$$

the init predicate

$$\frac{}{\mathsf{init}(\mathsf{nop})} \qquad \frac{\mathsf{init}(s_1) \quad \mathsf{init}(s_2)}{\mathsf{init}(\mathsf{seq}(s_1, s_2))}$$

$$\frac{\mathsf{init}(s) \quad \neg\mathsf{live}(s, x)}{\mathsf{init}(\mathsf{decl}(x, \tau, s))}$$

After Static Semantics …

- Translate AST to IR
- Then (or simultaneously) create Basic Blocks and CFG

Basic Blocks

- Each basic block starts with a “leader”
  - function entry
  - label
- Ends with **return** or **`jmp`**
- Only 1 entry, only 1 exit
- If last statement is conditional jump, two possible successors in control flow graph