The Middle-End

15-411/15-611 Compiler Design

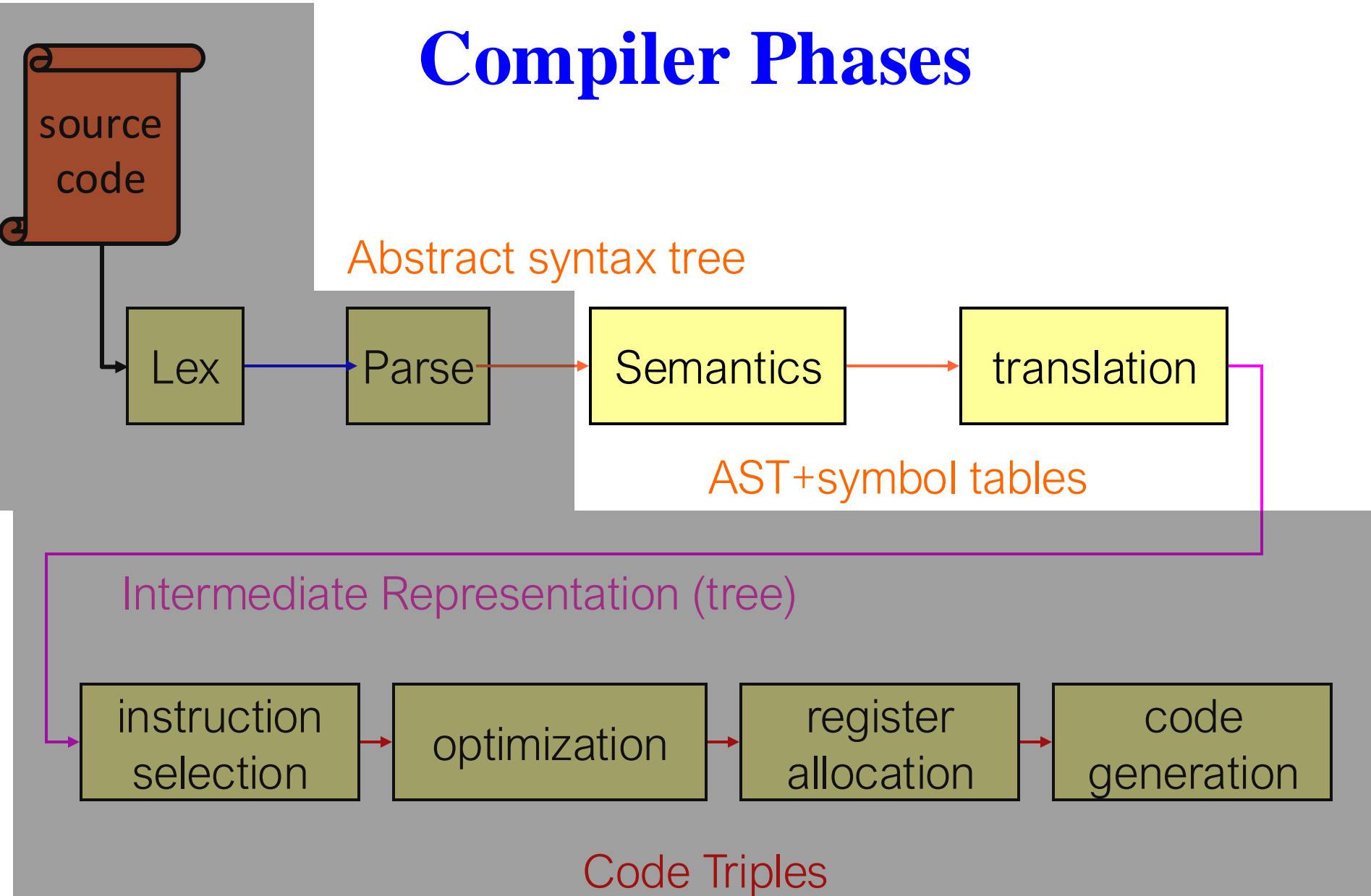Seth Copen Goldstein

February 4, 2025

Today

- lab2

- Elaboration

- Static Semantics

  - scope

  - symbol tables

- Type Checking (in brief)

- Inference Rules

  - Control Flow Checks

  - Initialization checks

- Basic Blocks

L2

$$\begin{aligned}
\langle program \rangle \quad &::= \quad \textbf{int main ()} \; \langle block \rangle \\
\langle block \rangle \quad &::= \quad \textbf{\{} \; \langle stmts \rangle \; \textbf{\}} \\
\langle type \rangle \quad &::= \quad \textbf{int} \;\mid\; \textbf{bool} \\
\langle decl \rangle \quad &::= \quad \langle type \rangle \; \textbf{ident} \;\mid\; \langle type \rangle \; \textbf{ident} = \langle exp \rangle \\
\langle stmts \rangle \quad &::= \quad \epsilon \;\mid\; \langle stmt \rangle \; \langle stmts \rangle \\
\langle stmt \rangle \quad &::= \quad \langle simp \rangle \; \textbf{;} \;\mid\; \langle control \rangle \;\mid\; \langle block \rangle \\
\langle simp \rangle \quad &::= \quad \langle lvalue \rangle \; \langle asop \rangle \; \langle exp \rangle \;\mid\; \langle lvalue \rangle \; \langle postop \rangle \;\mid\; \langle decl \rangle \;\mid\; \langle exp \rangle \\
\langle simpopt \rangle \quad &::= \quad \epsilon \;\mid\; \langle simp \rangle \\
\langle lvalue \rangle \quad &::= \quad \textbf{ident} \;\mid\; \textbf{(} \; \langle lvalue \rangle \; \textbf{)} \\
\langle elseopt \rangle \quad &::= \quad \epsilon \;\mid\; \textbf{else} \; \langle stmt \rangle \\
\langle control \rangle \quad &::= \quad \textbf{if (} \; \langle exp \rangle \; \textbf{)} \; \langle stmt \rangle \; \langle elseopt \rangle \\
&\quad\mid\quad \textbf{while (} \; \langle exp \rangle \; \textbf{)} \; \langle stmt \rangle \\
&\quad\mid\quad \textbf{for (} \; \langle simpopt \rangle \; \textbf{;} \; \langle exp \rangle \; \textbf{;} \; \langle simpopt \rangle \; \textbf{)} \; \langle stmt \rangle \\
&\quad\mid\quad \textbf{return} \; \langle exp \rangle \; \textbf{;} \\
\langle exp \rangle \quad &::= \quad \textbf{(} \; \langle exp \rangle \; \textbf{)} \;\mid\; \langle intconst \rangle \;\mid\; \textbf{true} \;\mid\; \textbf{false} \;\mid\; \textbf{ident} \\
&\quad\mid\quad \langle unop \rangle \; \langle exp \rangle \;\mid\; \langle exp \rangle \; \langle binop \rangle \; \langle exp \rangle \;\mid\; \langle exp \rangle \; \textbf{?} \; \langle exp \rangle \; \textbf{:} \; \langle exp \rangle \\
\langle intconst \rangle \quad &::= \quad \textbf{num} \\
\langle asop \rangle \quad &::= \quad \texttt{=} \mid \texttt{+=} \mid \texttt{-=} \mid \texttt{*=} \mid \texttt{/=} \mid \texttt{\%=} \mid \texttt{\&=} \mid \texttt{\^{}=} \mid \texttt{|=} \mid \texttt{<<=} \mid \texttt{>>=} \\
\langle binop \rangle \quad &::= \quad \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{==} \mid \texttt{!=} \\
&\quad\mid\quad \texttt{\&\&} \mid \texttt{||} \mid \texttt{\&} \mid \texttt{\^{}} \mid \texttt{|} \mid \texttt{<<} \mid \texttt{>>} \\
\langle unop \rangle \quad &::= \quad \texttt{!} \mid \texttt{\~{}} \mid \texttt{-} \\
\langle postop \rangle \quad &::= \quad \texttt{++} \mid \texttt{--}
\end{aligned}$$

Compiler Phases

source code

Abstract syntax tree

Lex → Parse → Semantics → translation

AST+symbol tables

Intermediate Representation (tree)

instruction selection → optimization → register allocation → code generation

Code Triples

Elaboration

- Eliminate syntactic sugar

- Simplify future analysis

- For example:
  - **for (init; test; incr) stmt**
  - **while (test) stmt**
  - **expr && expr**
  - **expr || expr**
  - others?

© 2019-21 Goldstein

for loop

```
for (init; test; incr) stmt


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

© 2019-21 Goldstein

X && Y

`exp1 && exp2`

$\quad \Rightarrow$

`exp1 || exp2`

$\quad \Rightarrow$

© 2019-21 Goldstein

X && Y

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

© 2019-21 Goldstein

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 = e;`
  - 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?

© 2019-21 Goldstein

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?

© 2019-21 Goldstein

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!

© 2019-21 Goldstein

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 "

© 2019-21 Goldstein

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, …

© 2019-21 Goldstein

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

© 2019-21 Goldstein

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.

© 2019-21 Goldstein

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

© 2019-21 Goldstein

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

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

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

decl?
if?
while?
nop?
assign?

© 2019-21 Goldstein

Check for Proper Returns

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

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

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

Iplementation

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

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

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

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

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

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)
  ```

© 2019-21 Goldstein

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)
```

If variable is live at point of declaration, then we have an error.

Plan for Verifying Proper Init

- ## If variable is live at point of declaration, then we have an error.

  - Determine if a variable is live at a statement
  - Will depend on whether there is a use of a variable in an expression
  - Determine if a statement will define a variable
  - Put it all together in a predicate to check for proper initialization.

the init predicate

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

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

If variable is live at point of declaration, then we have an error.

Plan for Verifying Proper Init

- If variable is live at point of declaration, then we have an error.
  - Determine if a variable is live at a statement
  - Will depend on whether there is a use of a variable in an expression
  - Determine if a statement will define a variable
  - Put it all together in a predicate to check for proper initialization.

live predicate (take 1)

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

© 2019-21 Goldstein

Plan for Verifying Proper Init

- ## If variable is live at point of declaration, then we have an error.

  - Determine if a variable is live at a statement
  - Will depend on whether there is a use of a variable in an expression
  - Determine if a statement will define a variable
  - Put it all together in a predicate to check for proper initialization.

the use predicate

$$\text{no rule for } \mathsf{use}(n, x) \qquad \frac{}{\mathsf{use}(x, x)} \qquad \text{no rule for } \mathsf{use}(y, x), y \neq x$$

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

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

use(e, x) is a may-property: no guarantee x will be used, but it may be used.

live predicate (take 2)

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

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

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

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

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

live predicate (take 2)

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

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

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

$$\frac{\text{use}(e, x)}{\text{live}(\text{return}(e), x)} \qquad \text{no rule for} \atop \text{live}(\text{nop}, x) \qquad \boxed{\frac{\text{live}(x, s) \quad y \neq x}{\text{live}(\text{decl}(y, \tau, s), x)}}$$

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

live predicate (take 2)

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

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

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

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

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

© 2019-21 Goldstein

Plan for Verifying Proper Init

- If variable is live at point of declaration, then we have an error.

  – Determine if a variable is live at a statement

  – Will depend on whether there is a use of a variable in an expression

  – Determine if a statement will define a variable

  – Put it all together in a predicate to check for proper initialization.

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
$$\mathsf{def}(\mathsf{nop},x)$$

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

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

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

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

s is in scope of y

© 2019-21 Goldstein

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 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