Recitation 3: Static Semantics and Lexing Solutions 6 February

Solution:

[TA only: Remember to Announce]

- L2 Compiler due Saturday 2/14.
- Written 2 due Wednesday 2/11. Submissions are through Gradescope.

Liveness Analysis Revisited

Liveness analysis is one instance of a dataflow problem; specifically it is a "backward, may" analysis. This means that the code will be iterated in reverse order, propagating information from successor to predecessor. The "may" part comes from how we union all information from the successor to pass to the predecessor. You will see many other dataflow analysis later on.

Previously, we have gone over computing liveness information for straight line code. Recall that a basic block starts with a function entry or label and ends with a return or jump instruction. To extend these ideas to programs with control flow, we can adopt a **worklist approach** where nodes are basic blocks as follows:

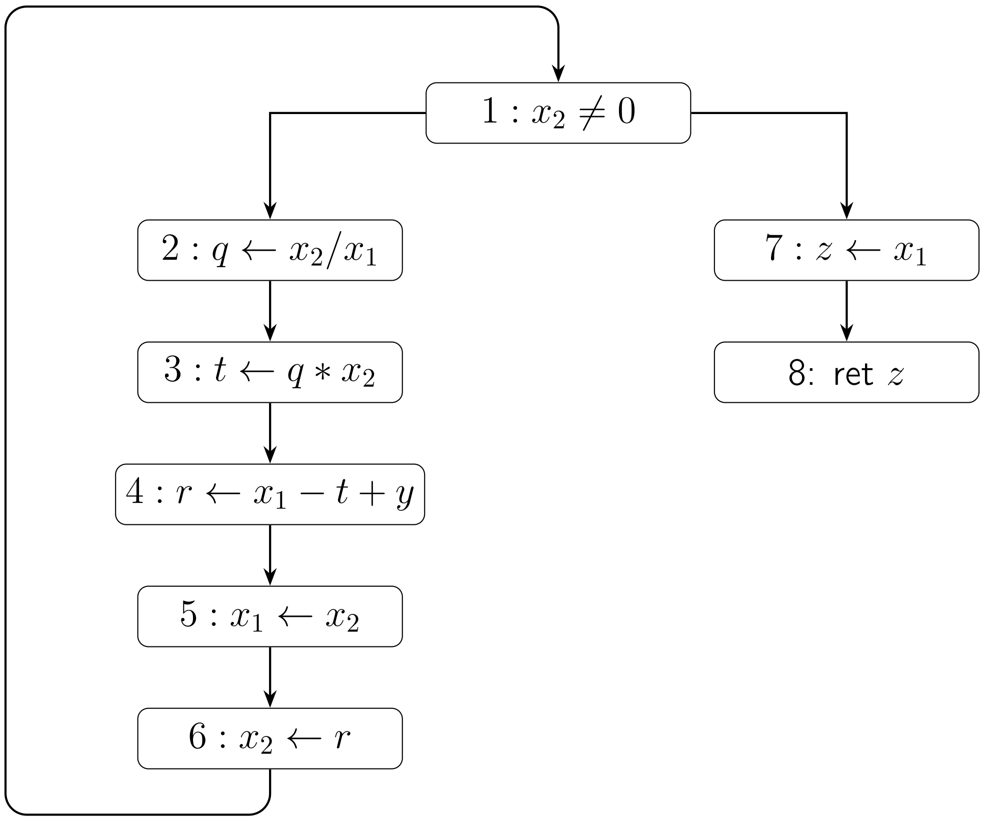
- For every node $n \in CFG$, set $In(n) = \emptyset$.
- Initialize a worklist W containing all basic blocks in the CFG. We will use a stack though other data structures also work.
- While W is not empty:
  - Remove a block p from W.
  - Compute $Out(p) = \bigcup_{s \in \text{succ}(p)} In(s)$
  - Apply the dataflow equation to calculate (note uses only includes variables used before they are defined in the same block)

$$In'(p) = Uses(p) \cup (Out(p) - Def(p))$$

  - **If** $In'(p) \neq In(p)$:
    - $In(p) \leftarrow In'(p)$
    - Add all predecessors of p back to W to propagate the change backward

Once we have the live-in and live-out sets for blocks, we can compute the live-in and live-out sets for each program line by running standard liveness analysis for each block separately.

Note while it is possible to construct the CFG using basic blocks from individual instructions, it can be significantly slower to reach a fixed-point.



Checkpoint 0

Identify the basic blocks in the CFG. Then, perform the worklist approach algorithm to compute the live-in and live-out sets for each basic block. For simplicity, initialize the stack as [Block C].

Solution: I'd recommend just working through this together for the sake of time.

Block A: 1 Block B: $2, 3, 4, 5, 6$ Block C: $7, 8$

$$Def(A) = \{\}, Uses(A) = \{x_2\}$$
$$Def(B) = \{q, t, r, x_1, x_2\}, Uses(B) = \{x_1, x_2, y\}$$
$$Def(C) = \{z\}, Uses(C) = \{x_1\}$$

Old W	$Out(p)$	$In'(p) = Uses(p) \cup (Out(p) - Def(p))$	New W
$[C]$	$\emptyset$	$\{x_1\} \cup (\emptyset - \{z\}) = \{x_1\}$	$[A]$
$[A]$	$\{x_1\}$	$\{x_2\} \cup (\{x_1\} - \emptyset) = \{x_1, x_2\}$	$[B]$
$[B]$	$\{x_1, x_2\}$	$\{x_1, x_2, y\} \cup (\{x_1, x_2\} - \{q, t, r, x_1, x_2\}) = \{x_1, x_2, y\}$	$[A]$
$[A]$	$\{x_1, x_2, y\}$	$\{x_2\} \cup (\{x_1, x_2, y\} - \emptyset) = \{x_1, x_2, y\}$	$[B]$
$[B]$	$\{x_1, x_2, y\}$	$\{x_1, x_2, y\} \cup (\{x_1, x_2, y\} - Def(B)) = \{x_1, x_2, y\}$	$[\,]$

$$In(A) = \{x_1, x_2, y\}, Out(A) = \{x_1, x_2, y\}$$

$$In(B) = \{x_1, x_2, y\}, Out(B) = \{x_1, x_2, y\}$$
$$In(C) = \{x_1\}, Out(C) = \{\}$$

With this liveness information between blocks, we can run per-line liveness analysis on each block separately and use it to construct an interference graph!

Elaboration in L2

The AST generated by your parser should reflect the source-code **syntactic** structure as closely as possible. You can then perform an elaboration pass to generate an AST that reflects the **semantic** structure of L2.

During L1, because all programs were simply a list of statements, they could for the most part be elaborated into `seq` and `nop` in a simple right-associative nesting:

$$s_1;\ s_2;\ s_3; \quad \Longrightarrow_{\text{elab}} \quad \texttt{seq}(s_1, \texttt{seq}(s_2, \texttt{seq}(s_3, \texttt{nop})))$$

The main tricky part of elaboration was handling declarations, for which our AST node $\texttt{decl}(x, \tau, s)$ contains s to clearly mark the scope of x. So if s_2 above was actually some declaration $\tau\ x$, the elaboration would instead be $\texttt{seq}(s_1, \texttt{decl}(x, \tau, \texttt{seq}(s_3, \texttt{nop})))$.

In L2 however, programs can get more complex. A block, which is a list of statements surrounded by braces, is itself a statement. Yet, our (post-elaboration) AST does not require an additional variant for blocks. We can represent them with only `seq` and `nop`.

Checkpoint 1

Write out the post-elaboration AST of the following program. Be careful about the nesting of `seq`'s and `decl`'s.

```
int main () {
  int a;
  {
    int b;
    b = 5;
    a = b;
  }
  int b;
  b = a;
  return b;
}
```

Solution:

```
decl(a, int,
  seq(
    decl(b, int,
      seq(assign(b, 5),
      seq(assign(a, b),
      nop))
    ),
    decl(b, int,
      seq(assign(b, a),
      seq(return(b),
      nop))
    )
  )
)
```

Of course, you are not required to elaborate in this specific manner. The way we have presented `decl`, `seq`, and `nop` makes it easier for us to define judgements for their static semantics, but it may affect the debugability of the post-elaboration AST in your compiler. You are free to not elaborate blocks and keep them as lists of statements.

Static Semantics of Initialization

For a C0 program to be valid, all variables must be declared and initialized before use. A compiler should confirm this property of a user program. To formally check this, we need to come up with a set of judgements and their associated inference rules. In class, we saw 2 different presentations of judgements that could achieve this. One of them used the following judgements:

- $\mathsf{use}(e, x)$: the variable x *might* be used when evaluating expression e.
- $\mathsf{def}(s, x)$: the variable x *must* be initialized after executing statement s.
- $\mathsf{live}[-\mathsf{in}](s, x)$: x *might* be used before initialization when executing s.
- $\mathsf{init}(s)$: all variables declared in s *must* be initialized before use in s.

While this presentation might be more intuitive, we will focus on another version which explicitly tracks the set of initialized variables. We denote a set of variables with δ and define the following two judgments:

- $\delta \vdash s \Rightarrow \delta'$
  Assuming all the variables in δ are defined when s is reached, no uninitialized variable will be referenced and after its execution all the variables in δ' will be defined.
- $\delta \vdash e$
  e will only reference variables defined in δ.

Here are some of the rules that define the judgement $\delta \vdash s \Rightarrow \delta'$:

$$\frac{}{\delta \vdash \texttt{nop} \Rightarrow \delta} \qquad \frac{\delta \vdash s_1 \Rightarrow \delta_1 \quad \delta_1 \vdash s_2 \Rightarrow \delta_2}{\delta \vdash \texttt{seq}(s_1, s_2) \Rightarrow \delta_2} \qquad \frac{\delta \vdash e}{\delta \vdash \texttt{assign}(x, e) \Rightarrow \delta \cup \{x\}} \qquad \frac{\delta \vdash e \quad \delta \vdash s \Rightarrow \delta'}{\delta \vdash \texttt{while}(e, s) \Rightarrow \delta}$$

$$\frac{\delta \vdash s \Rightarrow \delta'}{\delta \vdash \texttt{decl}(y, \tau, s) \Rightarrow \delta' - \{y\}} \qquad \frac{\delta \vdash e}{\delta \vdash \texttt{return}(e) \Rightarrow \{x \mid x \text{ in scope}\}}$$

In these judgments we have traded the complexity of traversing statements multiple times with the complexity of maintaining variables sets.

Checkpoint 2

Write the missing inference rule for $\delta \vdash \texttt{if}(e, s_1, s_2) \Rightarrow \delta'$.

Solution:

$$\frac{\delta \vdash e \quad \delta \vdash s_1 \Rightarrow \delta_1 \quad \delta \vdash s_2 \Rightarrow \delta_2}{\delta \vdash \texttt{if}(e, s_1, s_2) \Rightarrow \delta_1 \cap \delta_2}$$

Checkpoint 3

Using the inference rules as given, try to derive $\{\} \vdash s \Rightarrow \delta$ for the following program:

$$s = \mathsf{decl}(x, \texttt{int}, \mathsf{seq}(\mathsf{assign}(x, 3), \mathsf{return}(x)))$$

Solution:

$$\frac{\dfrac{\dfrac{\{\} \vdash 3}{\{\} \vdash \mathsf{assign}(x, 3) \Rightarrow \{x\}} \qquad \dfrac{\{x\} \vdash x}{\{x\} \vdash \mathsf{return}(x) \Rightarrow \{x\}}}{\{\} \vdash \mathsf{seq}(\mathsf{assign}(x, 3), \mathsf{return}(x)) \Rightarrow \{x\}}}{\{\} \vdash \mathsf{decl}(x, \texttt{int}, \mathsf{seq}(\mathsf{assign}(x, 3), \mathsf{return}(x))) \Rightarrow \{\}}$$

Unifying Static Semantics of Initialization and Typing

If you looked carefully earlier, or if you read the lecture notes, you'll notice that the rule for `return` is a bit strange, as it deals with scope in an informal way. The lecture notes suggest getting around this by also tracking a set γ of variables currently in scope. It turns out we can actually just use the typechecking context Γ that maps variables to types, as its domain $\operatorname{dom}\Gamma$ will be exactly the variables that are in scope. Incidentally, this means we can even combine it with the typing judgement for statements to create the following new judgment

$$\Gamma;\ \Delta \vdash s : [\tau] \Rightarrow \Delta'$$

The statement s

- is in the scope of the variables in Γ, which were *declared* with their corresponding types
- only uses *initialized* variables from Δ
- leaves the variables in Δ' *initialized* after execution
- returns a value of type τ if it returns (intuitively, this is the return type of the function s belongs in)

We can similarly define $\Gamma;\ \Delta \vdash e : \tau$ to mean "e uses only variables in Δ and has type τ given the context Γ". Here are some of the rules that define the judgement $\Gamma;\ \Delta \vdash s : [\tau] \Rightarrow \Delta'$:

$$\frac{\Gamma;\ \Delta \vdash s_1 : [\tau] \Rightarrow \Delta' \qquad \Gamma;\ \Delta' \vdash s_2 : [\tau] \Rightarrow \Delta''}{\Gamma;\ \Delta \vdash \mathsf{seq}(s_1, s_2) : [\tau] \Rightarrow \Delta''} \qquad \frac{}{\Gamma;\ \Delta \vdash \mathsf{nop} : [\tau] \Rightarrow \Delta}$$

$$\frac{\Gamma, x : \tau';\ \Delta \vdash s : [\tau] \Rightarrow \Delta'}{\Gamma;\ \Delta \vdash \mathsf{declare}(x, \tau', s) : [\tau] \Rightarrow \Delta' \setminus \{x\}}$$

$$\frac{\Gamma;\ \Delta \vdash e : \mathsf{bool} \qquad \Gamma;\ \Delta \vdash s_1 : [\tau] \Rightarrow \Delta' \qquad \Gamma;\ \Delta \vdash s_2 : [\tau] \Rightarrow \Delta''}{\Gamma;\ \Delta \vdash \mathsf{if}(e, s_1, s_2) : [\tau] \Rightarrow \Delta' \cap \Delta''}$$

$$\frac{\Gamma;\ \Delta \vdash e : \mathsf{bool} \qquad \Gamma;\ \Delta \vdash s : [\tau] \Rightarrow \Delta'}{\Gamma;\ \Delta \vdash \mathsf{while}(e, s) : [\tau] \Rightarrow \Delta}$$

Checkpoint 4

Write the missing inference rules for $\Gamma;\ \Delta \vdash \mathsf{assign}(x, e) : [\tau] \Rightarrow \Delta'$ and $\Gamma;\ \Delta \vdash \mathsf{return}(e) : [\tau] \Rightarrow \Delta'$.

Solution:

$$\frac{\Gamma(x) = \tau' \qquad \Gamma;\ \Delta \vdash e : \tau'}{\Gamma;\ \Delta \vdash \mathsf{assign}(x, e) : [\tau] \Rightarrow \Delta \cup \{x\}} \qquad \frac{\Gamma;\ \Delta \vdash e : \tau}{\Gamma;\ \Delta \vdash \mathsf{return}(e) : [\tau] \Rightarrow \operatorname{dom}\Gamma}$$

Checkpoint 5

Write the inference rule for $\Gamma;\ \Delta \vdash x : \tau$.

Solution: $\dfrac{\Gamma(x) = \tau \qquad x \in \Delta}{\Gamma;\ \Delta \vdash x : \tau}$

We have shown that it is quite possible, and not too inelegant, to implement static semantics for initialization and typing as a single judgement. It is up to you whether to do this in your own compiler – you could definitely check initialization and typing in separate passes.