15-411: Compiler Design — Spring 2025

Recitation 3: Static Semantics and Lexing — 7 February

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:

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

The main tricky part of elaboration was handling declarations, for which our AST node $\mathtt{decl}(x, \tau, s)$ 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 $\mathtt{seq}(s_1, \mathtt{decl}(x, \tau, \mathtt{seq}(s_3, \mathtt{nop})))$.

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 0

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;
}
```

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:

- use(e, x): the variable x *might* be used when evaluating expression e.
- def(s, x): the variable x *must* be initialized after executing statement s.
- live(s, x): x *might* be used before initialization when executing s.
- 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 1

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

Checkpoint 2

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

$$\text{decl}(x, \texttt{int}, \text{seq}(\text{assign}(x, 3), \text{return}(x)))$$

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 $\mathrm{dom}\,\Gamma$ 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

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 3

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

Checkpoint 4

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

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.