Notes for CSC 2/454, Feb. 13 and 18, 2025

==========

Static Analysis and Action Routines

Static semantics are enforced at compile time, dynamic semantics at run time. In principle, we don't need static semantics at all: everything could be figured out at run time. From this perspective, static semantics is an optimization—a chance to get error messages sooner and to move work off the critical path of run-time execution. Language theorists tend to define semantics as purely dynamic. Then they write static semantic rules (the ones for the type system tend to be the most complex). The static semantics is said to be *sound* if everything it deduces at compile time would always have come out the same way at run time.

Some things have to be dynamic semantics because of ***late binding*** (discussed in Chap. 3): we lack the necessary info (e.g., input values) at compile time, or inferring what we want is uncomputable.

A smart compiler may avoid run-time checks when it ***is*** able to verify compliance at compile time. This makes programs run faster.

- array bounds
- variant record tags
- dangling references

Similarly, a conservative code improver will apply optimizations only when it knows they are both safe and beneficial. A more aggressive compiler may use optimizations that are always safe and *often* beneficial, or generate multiple versions with a dynamic check to see if the optimized version is safe.

Alternatively, the language designer may tighten rules

- type checking in ML v. Lisp (cons: 'a * 'a list -> 'a list)
- definite assignment in Java/C# v. C
- ownership in Rust

Semantic analysis can be interleaved with parsing in a variety of ways. The most common approach today is to use action routines to build an AST, then perform semantic analysis on each class or subroutine as soon as it has been parsed.

In a recursive descent parser, AST fragments can be returned from and, when needed, passed to RD routines:

```
AST_node expr():
   case input_token of
      id, literal, ( :
         T := term()
         return term_tail(T)
   else error

AST_node term_tail(T1):
   case input_token of
      +, - :
         O := add_op()
         T2 := term()
         N := new Bin_op(O, T1, T2)   // subclass of AST_node
         return term_tail(N)
      ), id, read, write, $$ :
         return T1                    // epsilon
   else error
```

Here code in black is the original RD parser; red has been added to build the AST. In the input to a parser generator, it's conventional to specify this code as ***action routines*** embedded in the CFG. . The book presents this in Fig. 4.6. Here's a slightly different version more consistent with the notation of project 2:

E ⟶ T { TT.st := T.n } TT { E.n := TT.n }

TT_1 ⟶ ao T { TT_2.st := make_bin_op(ao.op, TT_1.st, T.n) } TT_2 { TT_1.n := TT_2.n }

TT ⟶ ε { TT.n := TT.st }

T ⟶ F { FT.st := F.n } FT { T.n := FT.n }

FT_1 ⟶ mo F { FT_2.st := make_bin_op(mo.op, FT_1.st, F.n) } FT_2 { FT_1.n := FT_2.n }

FT ⟶ ε { FT.n := FT.st }

F ⟶ (E) { F.n := E.n }

F ⟶ id { F.n := id.n } // id.n comes from scanner

F ⟶ lit { F.n := lit.n } // as does lit.n

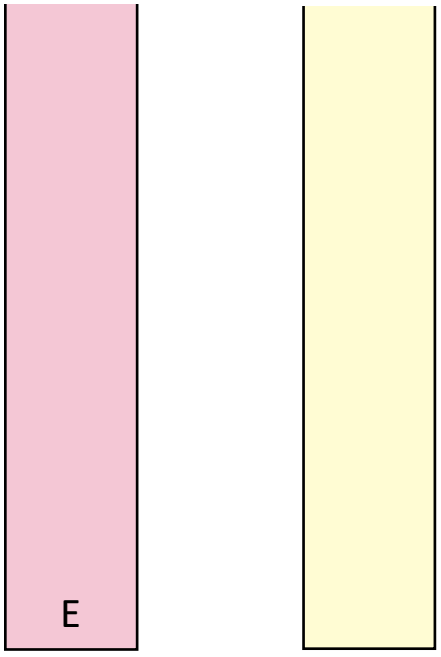
Here the subscripts distinguish among instances of the same symbol in a given production. The .n and and .st suffixes are ***attributes*** (fields) of symbols.
I've elided the ao and mo productions.

Where are these attributes stored? The parse stack won't do: it contains the future, not the past. One appealing option is to maintain an "attribute stack" that holds everything that's active: ***all the symbols of all the productions*** on the path from the root to the current top-of-parse-stack symbol.

- maintain *lhs* and *rhs* indices into the attribute stack
- when predicting production for nonterminal A,
    - push a marker underneath the RHS in the parse stack
      (indicates LHS symbol and RHS length)
    - save *lhs* and *rhs* on a separate stack
    - push space for all symbols of RHS onto the attribute stack
    - point *lhs* at the attribute stack symbol for A
    - point *rhs* at the new attribute stack symbols
- update attribute stack symbols for tokens when matching them
- at end of production
    - pop attribute stack space used by RHS
    - restore *lhs* and *rsh* indices

1: E ⟶ T TT
2: TT ⟶ ao T TT
3: T ⟶ F FT
4: FT ⟶ mo F FT
5: F ⟶ (E)
6: F ⟶ id
7: F ⟶ lit
. . .

(A – 1) * B

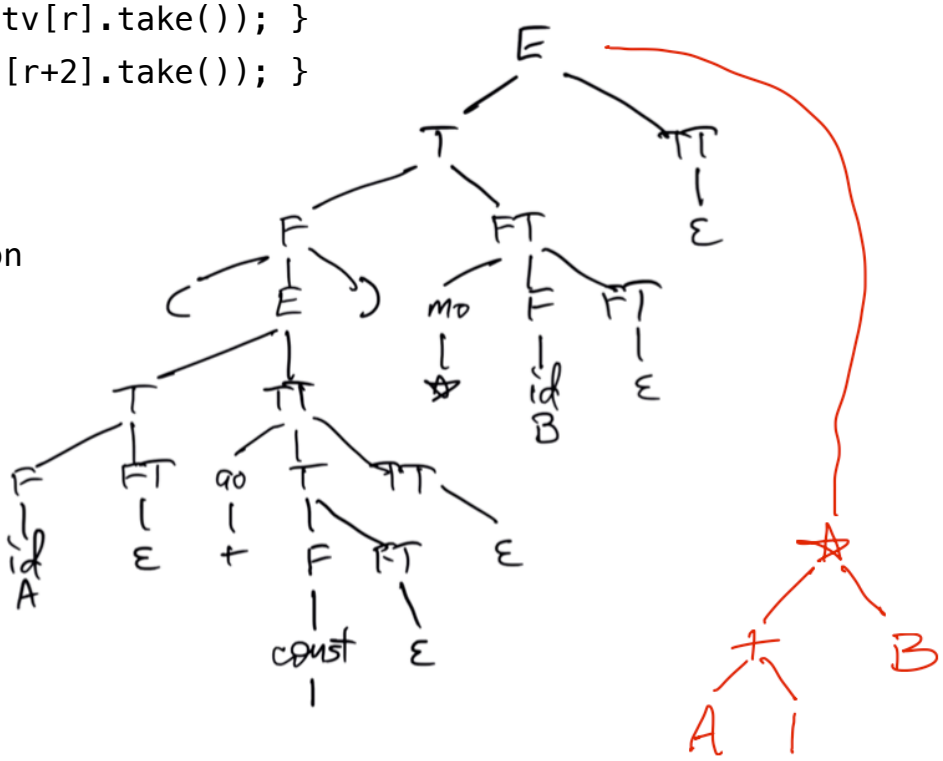


A simplified version of this scheme can also be used to build an explicit parse tree completely automatically, in which case the AST can be built during a separate parse-tree traversal.

Using the syntax of project A2, the E ⟶ T {2} TT {3}
production would have action routines

```
2 => { atv[r+2].set(atv[r].take()); }
3 => { atv[l].set(atv[r+2].take()); }
```

Try tracing AST construction
for (A + 1) * B :



For a table-driven LL parser, the parser generator gives each action routine a number, pushes these into the parse stack along with other RHS symbols, and executes them as they are encountered, e.g., by calling a do_action(#) routine with a big switch inside.

==========

AST structure

Inside the compiler, AST nodes are structs. To facilitate formal semantics, these are typically defined with an ***abstract grammar*.**
Before we give an example, let's extend the calculator grammar with types and declarations, so it has some static semantics worth checking:

program	⟶	*stmt_list* $$
stmt_list	⟶	*decl stmt_list* | *stmt stmt*_list | ε
decl	⟶	int id | real id
stmt	⟶	id := *expr* | read id | write *expr*

expr ⟶ *term term_tail*
term_tail ⟶ *add_op term term_tail* | ε
term ⟶ *factor factor_tail*
factor_tail ⟶ *mul_op factor factor_tail* | ε
factor ⟶ (*expr*) | id | *int_const* | *real_const*
| float (*expr*) | trunc (*expr*)
add_op ⟶ + | –
mul_op ⟶ * | /

Now we can
- require declaration before use
- require type match on arithmetic ops, assignment, and float/trunc

Each "production" of the abstract grammar has an AST node type (class) on the left-hand side and a set of variants (subclasses), separated by vertical bars, on the right-hand side. Note that the abstract grammar is ***not for parsing***; it's to describe the trees that should be built by the parser and checked/annotated by the semantic analyzer.

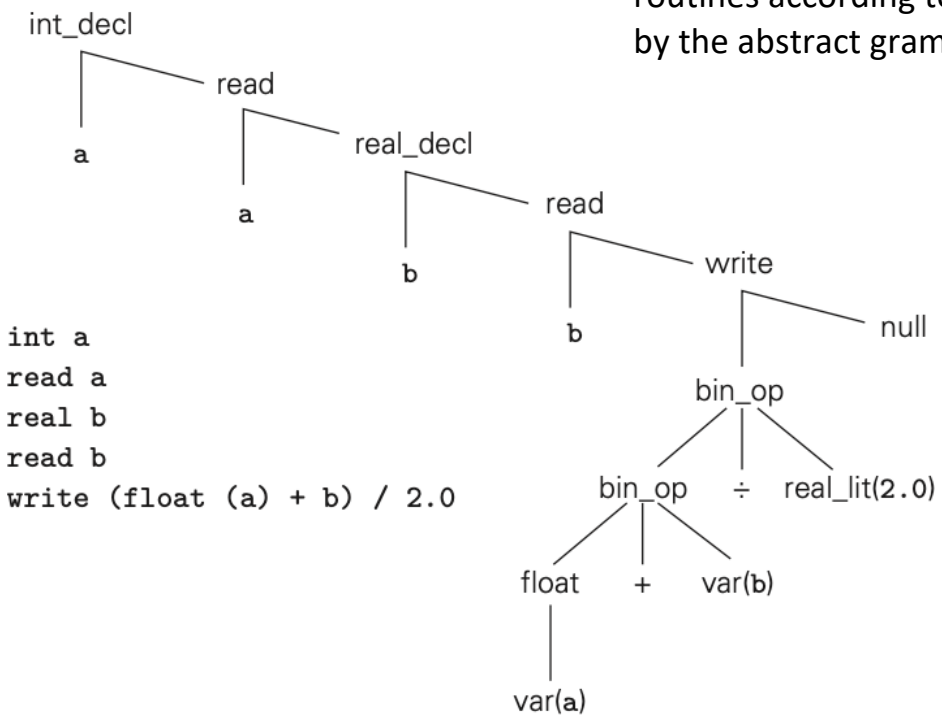
For convenience, we also provide a linear form for trees, to facilitate writing down semantic rules. We add parentheses when necessary to disambiguate.

Example for the extended calculator language:

s ⟶	int_decl (*x, s*)	int *x* ; *s*
	| real_decl (*x, s*)	real *x* ; *s*
	| assign (*x, e, s*)	*x* := *e* ; *s*
	| read (*x, s*)	read *x* ; *s*
	| write (*e, s*)	write *e* ; *s*
	| null	*ε*
e ⟶	var (*x*)	*x*
	| int_lit (*n*)	*n*
	| real_lit (*r*)	*r*
	| float (*e*)	float *e*
	| trunc (*e*)	trunc *e*
	| bin_op (*e, o, e*)	*e o e*

o ∈ {+, −, ∗, /}
x ∈ *variables*
n ∈ *integers*
r ∈ *reals*

Here's a syntax tree for a tiny program, generated by (handwritten) action routines according to structure given by the abstract grammar.




```
int a
read a
real b
read b
write (float (a) + b) / 2.0
```

Remember: abstract grammars are not CFGs. Language for a CFG is the set of possible ***fringes*** of parse trees. Language for an abstract grammar is the set of possible ***whole abstract trees***. No comparable notion of parsing: structure of tree is self-evident.

In addition to specifying AST structure, the abstract grammar provides a framework for specifying dynamic and static semantics, typically in the form of ***inference rules*** that describe relationships among the annotations of parent and children nodes in the AST. Inference rules are sort of like action routines, but written in mathematical notation and without explicit specification of what is executed when.

Automatic tools to convert inference rules into a semantic analyzer are a current topic of research. In practice, semantic analyzers are still written by hand. That said, a good set of inference rules

- imposes discipline on our thinking as we define the language
- provides a concise specification of semantics that is more readable than the code and more precise than English
- defines a common standard—a formal characterization of the language that determines whether a hand-written implementation is correct or not

Most languages don't have formal definitions, but they're clearly the wave of the future. WebAssembly is a great example.

Inference rules can be used to specify all aspects of program semantics. The typical modern semantic framework specifies ***dynamic semantics*** of the (abstract) language as a set of inference rules that define the behavior of the language on an ***abstract* machine,** determining the output of a <program, input> pair.

Compilers (and, to a lesser extent, interpreters) typically also specify ***static semantics*** to pre-compute whatever they can. In particular, they perform ***static type checking*** in order to reduce overhead during eventual execution and to catch errors early. This type checking typically involves more than the usual programmer thinks of as types: it includes things like

- passing the right # of parameters to subroutines
- using only disjoint constants as case statement labels
- putting a return statement at the end of (every code path of) every function
- putting a break statement only inside a loop
- . . .

(Theoreticians, in fact, consider type checking a *purely static* activity. They don't use the term "type checking" for what happens at run time in a dynamically typed language like Python—they call that "safety" instead.)

Static semantics is said to be ***sound*** if every judgment it reaches matches what dynamic semantics would have concluded at run time. (It is generally not *complete*—it does not reach all the judgments that can be reached at run time.)

An inference rule is typically written with a long horizontal line, with *predicates* above the line and a *conclusion* below the line. Both predicates and conclusions are referred to as ***judgments***; they often (though not always) describe properties of nodes in a parent-and-children neighborhood of an AST. As an example, in an AST subtree comprising a constant expression, we might write

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad n_1 + n_2 = n_3}{e_1 + e_2 \Downarrow n_3} \; \textit{ev-add-n}$$

The *ev-add-n* rule specifies that if e_1 evaluates to n_1, e_2 evaluates to n_2, and $n_1 + n_2 = n_3$, then $e_1 + e_2$ (i.e., bin_op(e_1, +, e_2)) evaluates to n_3.

If we flesh out formal semantics for the calculator language with variables, types, and multiple operators it has a more sophisticated version of this rule:

$$\frac{E \vdash e_1 \Downarrow v_1 \quad E \vdash e_2 \Downarrow v_2 \quad type(v_1) = type(v_2) \quad v_1 \oplus v_2 = v_3}{E \vdash e_1 \oplus e_2 \Downarrow v_3} \; \textit{ev-binop}$$

This introduces the notion of an ***environment*** (a mapping from names to values, where values have types—not the same as a referencing environment). It then generalizes across types and operators. We say: "If e_1 evaluates to v_1 in environment *E,* e_2 evaluates to v_2 in environment *E*, v_1 and v_1 have the same type, and $v_1 \oplus v_2 = v_3$, then $e_1 \oplus e_2$ (i.e., bin_op(e_1, $\oplus$, e_2)) evaluates to v_3 in environment *E.* The $\vdash$ symbol is called a "turnstile." In the last premise, $\oplus$ is a math operator; in the conclusion the corresponding syntactic operator.

Remember that this is a dynamic semantic rule: we only know values at run time.

Other inference rules change the environment (note the left-pointing turnstile):

$$\frac{x \notin dom(E_1) \quad E_1[x \mapsto 0] \vdash s \dashv E_2}{E_1 \vdash \texttt{int}\ x\ \texttt{;}\ s \dashv E_2} \; \textit{ev-int-decl}$$

$$\frac{E_1 \vdash e \Downarrow v \quad type(v) = type(E_1(x)) \quad E_1[x \mapsto v] \vdash s \dashv E_2}{E_1 \vdash x\ \texttt{:=}\ e\ \texttt{;}\ s \dashv E_2} \; \textit{ev-assign}$$

An integer declaration introduces a new name into the environment (with, here, an initial value of 0). An assignment updates the value in the environment. In both cases, the new environment is used for subsequent statements.

Inference rules do not specify the order in which they should be evaluated. There exist tools to figure that out, but in practice (in a real compiler) we typically figure out the order by hand and write recursive routines that walk the AST and compute the values of fields.

The book gives inference rules and recursive routines to type-check programs in the extended calculator grammar. The recursive routines tag nodes with essential information (e.g., type, scope). We could pass the symbol table and error messages among nodes, too, but it's more common to make these globals:

insert errors, as found, into a list or tree, sorted by source location
for symtab, label each construct with list of active scopes
look up <name, scope> pairs, starting with closest scope

To avoid cascading errors, it's common to have an "error" value for an attribute that means "I already complained about this." So in the following we label the '+' node with type "error" to avoid a message for the ":=" node:

```
int a
real b
int c
a := b + c
```

Type checking (with error list and symtab as globals) for expressions:

```
function typecheck_expr(scope : Scope, a : AST) : Type
    case a of
        int_lit(n) : return integer
        real_lit(r) : return real
        var(x) :
            typ : Type := symbol_table.get_type(x, scope, a)
                -- if x is not found, get_type will call error(''variable not declared'', a)
                -- and add x to scope with error_type, to avoid cascading messages
            return typ
        float(a1) :
            typ1 : Type := typecheck_expr(scope, a1)
            if typ1 ∉ {integer, error_type} then error(''already a real'', a)
            return real
```

```
        trunc(a1) :
            typ1 : Type := typecheck_expr(scope, a1)
            if typ1 ∉ {real, error_type} then error("already an integer", a)
            return integer
        bin_op(a1, op, a2) :
            typ1 : Type := typecheck_expr(scope, a1)
            typ2 : Type := typecheck_expr(scope, a2)
            if typ1 = typ2 then return typ1
            else if typ1 = error_type then return typ2
            else if typ2 = error_type then return typ1
            else error("mismatched types", a); return error_type
```

We've assumed here that variables have names (and numbers, values), initialized by the scanner. We've also assumed that the code in the parser that builds the AST labels all constructs with their location.
For statements:

```
function typecheck_stmt(scope : Scope, a : AST)
    case a of
        int_decl(x, s) :
            symbol_table.add(x, integer, scope, a)
                -- if x is already present and not of error_type, add will
                -- call error(''variable already declared in scope'', a)
                -- and set the type of x to error_type if the two declarations differ
            typecheck_stmt(scope, s)
        real_decl(x, s) : . . .                                    -- analogous to int_decl
        assign(x, e, s) :
            typ_expr := typecheck_expr(scope, e)
            typ_x := symbol_table.get_type(x, scope, a)        -- see notes on get_type above
            if typ_expr ≠ typ_x and typ_expr ≠ error_type and typ_x ≠ error type
                error(''mismatched types'')
            typecheck_stmt(scope, s)
        read(x, s) :
            typ_x := symbol_table.get_type(x, scope, a)        -- see notes on get_type above
            typecheck_stmt(scope, s)
        write(e, s) :
            typecheck_expr(scope, e)
            typecheck_stmt(scope, s)
        null : return
```

The calculator grammar is simple enough that we can interpret the entire program in a single left-to-right pass over the tree. In more realistic languages, we might need to do multiple traversals—e.g., one to identify all the names and insert them in the symbol table, another to make sure the names have all been used consistently (think of calls to mutually-recursive methods, which may appear before the corresponding method declaration), and a third to actually "execute."

If we were building a compiler instead of an interpreter, the final pass wouldn't "execute" the program but rather spit out a translated version.

For reference, here's the complete dynamic semantics for the calculator language with types:

$\boxed{E \vdash e \Downarrow v}$

$$\frac{E(x) = v}{E \vdash x \Downarrow v}\ \textit{ev-var} \qquad \frac{}{v \Downarrow v}\ \textit{ev-const}$$

$$\frac{E \vdash e \Downarrow n \quad \textit{to\_float}(n) = r}{E \vdash \texttt{float}(e) \Downarrow r}\ \textit{ev-float} \qquad \frac{E \vdash e \Downarrow r \quad \textit{truncate}(r) = n}{E \vdash \texttt{trunc}(e) \Downarrow r}\ \textit{ev-trunc}$$

$$\frac{E \vdash e_1 \Downarrow v_1 \quad E \vdash e_2 \Downarrow v_2 \quad \textit{type}(v_1) = \textit{type}(v_2) \quad v_1 \oplus v_2 = v_3}{E \vdash e_1 \oplus e_2 \Downarrow v_3}\ \textit{ev-binop}$$

$\boxed{E_1 \vdash s \dashv E_2}$

$$\frac{x \notin \textit{dom}(E_1) \quad E_1[x \mapsto 0] \vdash s \dashv E_2}{E_1 \vdash \texttt{int}\ x\ \texttt{;}\ s \dashv E_2}\ \textit{ev-int-decl}$$

$$\frac{x \notin \textit{dom}(E_1) \quad E_1[x \mapsto 0.0] \vdash s \dashv E_2}{E_1 \vdash \texttt{real}\ x\ \texttt{;}\ s \dashv E_2}\ \textit{ev-real-decl} \qquad \frac{}{E \vdash \varepsilon \dashv E}\ \textit{ev-empty}$$

$$\frac{E_1 \vdash e \Downarrow v \quad \textit{type}(v) = \textit{type}(E_1(x)) \quad E_1[x \mapsto v] \vdash s \dashv E_2}{E_1 \vdash x\ \texttt{:=}\ e\ \texttt{;}\ s \dashv E_2}\ \textit{ev-assign}$$

$$\frac{\textit{read\_console}() = v \quad \textit{type}(v) = \textit{type}(E_1(x)) \quad E_1[x \mapsto v] \vdash s \dashv E_2}{E_1 \vdash \texttt{read}\ x\ \texttt{;}\ s \dashv E_2}\ \textit{ev-read}$$

$$\frac{E_1 \vdash e \Downarrow v \quad \textit{write\_console}(v) = \text{OK} \quad E_1 \vdash s \dashv E_2}{E_1 \vdash \texttt{write}\ e\ \texttt{;}\ s \dashv E_2}\ \textit{ev-write}$$

Static semantics introduce rules with a "typing context" Γ. This context functions a lot like the environment *E* of the dynamic semantics, but instead of mapping names to values (which have self-evident types), it maps names to types. It supports judgments like $\Gamma \vdash e : \tau$, meaning "in typing context Γ, expression e has type τ.

The basic soundness theorem then asserts that if $\Gamma \vdash e : \tau$, $\Gamma \vdash E$, and $E \vdash e \Downarrow v$, then $\Gamma \vdash v : \tau$. That is, if *e* has type τ at compile time, *E* is well formed in Γ, and *e* evaluates to *v* at run time, then *v* will have type τ. By "*E* is well formed in Γ," we mean

$$\frac{x : \tau \in \Gamma \Rightarrow \Gamma \vdash E(x) : \tau}{\Gamma \vdash E} \; \textit{env-var}$$

That is, *E* is well formed in Γ if it's always the case that when *x* has type τ in Γ we know that the value of *x* in *E* has type τ. (Note here that *x* is a variable, while *e* is an (arbitrarily complex) expression.