Notes for CSC 2/454, Sept. 20 and 25, 2023

==========

Static Analysis and Action Routines

Recall that static semantics are enforced at compile time, and dynamic semantics are enforced 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

- alias analysis
  - caching in registers
  - computation out of order or in parallel
- escape analysis
  - limited extent
  - non-synchronized
- subtype analysis
  - static dispatch of virtual methods

A more aggressive compiler may
- use optimizations that are always safe and *often* beneficial
  - prefetching
  - trace scheduling
- generate multiple versions with a dynamic check to dispatch
- use optimizations that are often safe and often beneficial, so long as it checks along the way at tun time, to make sure it's safe, and is prepared to roll back if necessary
  - transactional memory

Alternatively, language designer may tighten rules
- type checking in ML v. Lisp (cons: 'a * 'a list -> 'a list)
- definite assignment in Java/C# v. C

As noted in Chap. 1, job of semantic analyzer is to
(1) enforce rules
(2) connect the syntax of the program (as discovered by the parser) to something else that has semantics (meaning) – e.g.,
  - value for constant expressions
  - code for subroutines

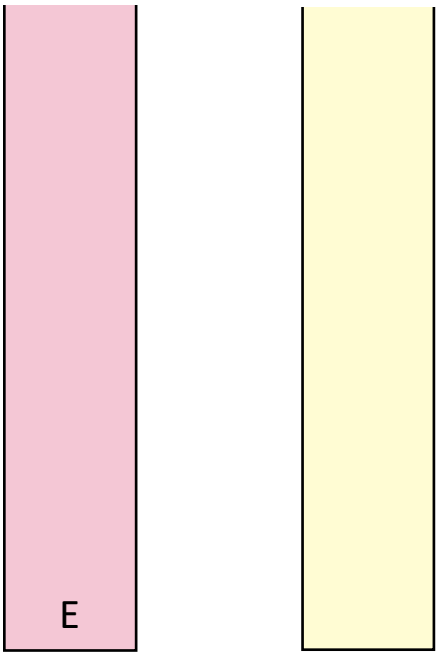
This work can be interleaved with parsing in a variety of ways.
- At one extreme: build an explicit parse tree, then call the semantic analyzer as a separate pass.
- At the other extreme, perform all static dynamic checks and generate intermediate form while parsing, using ***action routines*** called from the parser.
- The most common approach today is intermediate: use action routines to build an AST, then perform semantic analysis on each top-level AST fragment (class, function) as it is completed.

We'll focus on this intermediate approach. But first, it's instructive to see how we ***could*** build an explicit parse tree if we wanted. This will help motivate the code to build an AST.

recursive descent
    each routine returns its subtree
table-driven top-down
    push markers at end-of-production
    each, when popped, pulls *k* subtrees off separate ***attribute stack***
        and pushes new subtree, where *k* is length of RHS

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



So how do we build a syntax tree instead?
Start with RD, work with parameters, local variables, return values:

```
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.

It's standard practice to express the extra code as ***action routines*** in the CFG:

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 }

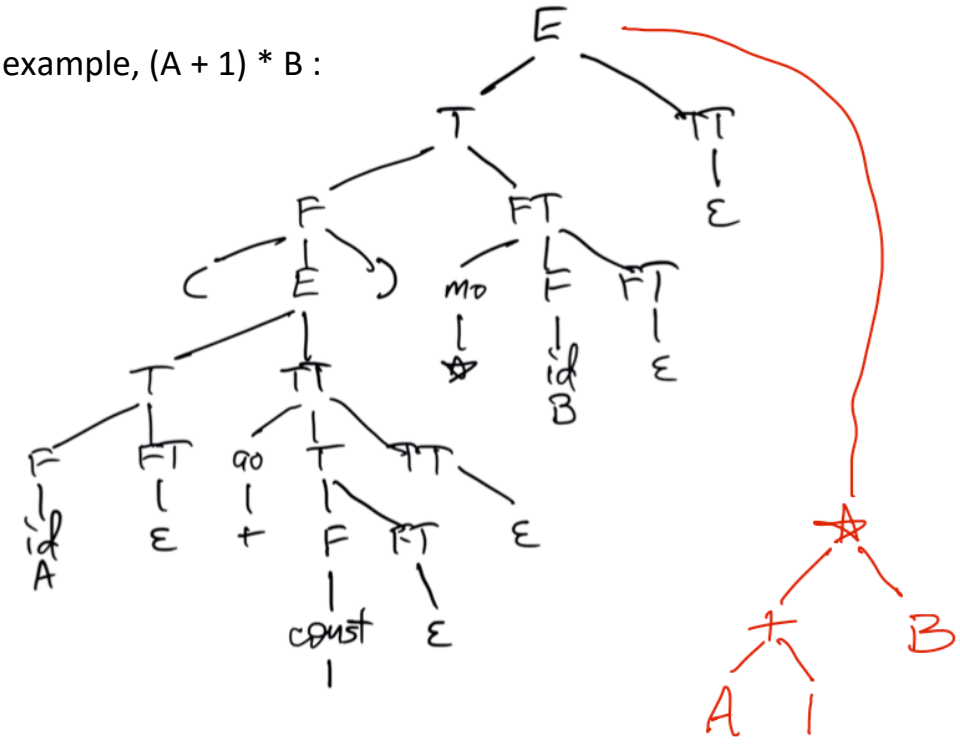
F ⟶ (E) { F.n := E.n }

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.

See how this handles, for example, (A + 1) * B :



A parser generator like ANTLR can turn the grammar w/ action routines into an RD parser that builds a syntax tree.

It's also straightforward to turn that grammar into a table-driven TD parser.

Give each action routine a number
Push these into the stack along with other RHS symbols
Execute them as they are encountered. That is:
- match terminals
- expand nonterminals by predicting productions
- execute action routines
    e.g., by calling a do_action(#) routine with a big switch
        statement inside

requires space management for attributes; companion site (Sec. 4.5.2)
    explains how to maintain that space automatically
extension of the attribute stack we used to build a parse tree above
    space for all symbols of all productions on path from root
    to current top-of-parse-stack symbol
- when predict, push space for all symbols of RHS
- maintain *lhs* and *rsh* indices into the stack
- at end of production, pop space used by RHS; update *lhs* and *rsh* indices

==========

Decorating a Syntax Tree

The calculator language we've been using for examples doesn't have sufficiently interesting semantics.
Consider an extended version with types and declarations:

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

We could do some of this checking while building the AST.
We could even do it while building an explicit parse tree.

The more common strategy is to implement checks once the AST is built
easier -- tree has nicer structure
more flexible -- can accommodate non depth-first left-to-right traversals
- mutually recursive definitions
e.g., methods of a class in most languages
- type inference based on use
- switch statement label checking
etc.

Assume the parser builds the AST and tags every node with a source location.

Tagging of tree nodes is ***annotation***
inside the compiler, tree nodes are structs
annotations and pointers to children are fields
(annotation can also be done to an explicit parse tree; we'll stick to ASTs)

But first: what do we want the AST to look like?
The book uses what it calls a ***tree grammar***.
This is nice and clear, but it doesn't match the literature, which uses an equivalent but superficially different notation called an ***abstract grammar***.
The 5th edition of the text will use this more standard notation.

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

- we want the parser to build
- we need to annotate

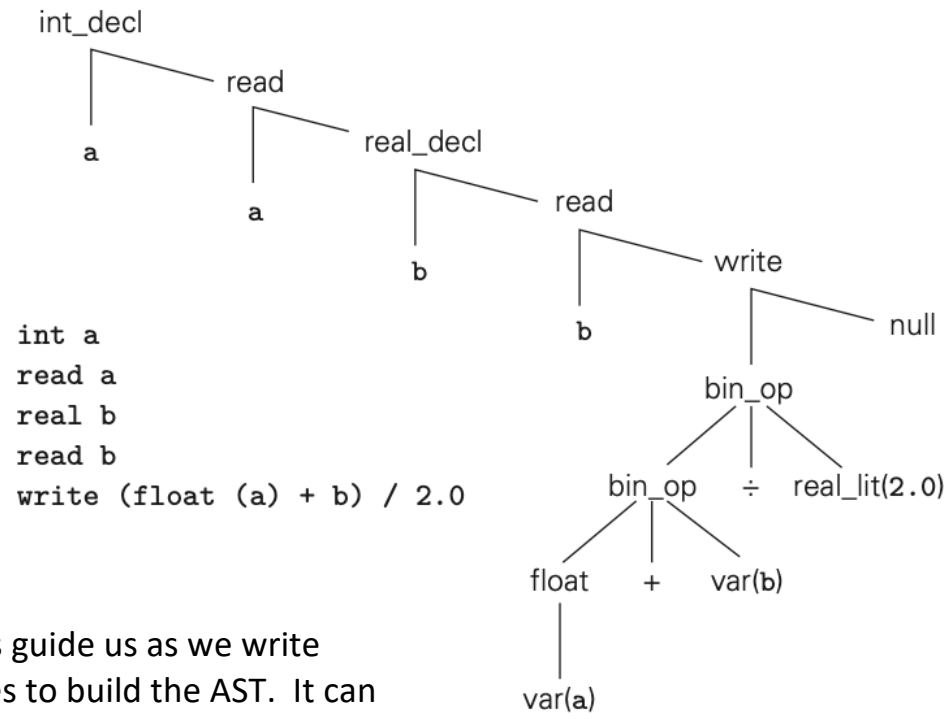
For convenience, we also provide a linear form for trees, to facilitate writing down semantic rules. We'll insert parentheses into this linear form when necessary for disambiguation.

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. Structure is given by the abstract grammar. Construction would be via execution of appropriate action routines embedded in a CFG.

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



Our abstract grammar helps guide us as we write (by hand) the action routines to build the AST. It can also help guide us in writing recursive tree-walking routines to perform semantic checks and (later) generate mid-level intermediate code.

- Helpful to augment the tree grammar with ***semantic rules*** that describe relationships among annotations of parent and children.
- Semantic rules are like action routines, but without explicit specification of what is executed when.

Semantic rules on an AST can be specified in multiple ways. The book uses ***attribute grammars*** (AGs), which specify the value of AST node fields (attributes) as functions of the values of other attributes in the same local parent-and-children neighborhood of the tree.

AGs are not used much in production compilers these days, but have proven useful for prototyping (e.g., the first validated Ada implementation [Dewar et al., 1980]) and for some cool language-based tools

- syntax-directed editing [Reps, 1984]
- parallel CSS [Meyerovich et al., 2013]

More common these days are ***inference rules***, which are more declarative than AGs, and more amenable to formalization and automatic proofs of correctness (not covered here). They will be used in the 5th edition of the text.

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). 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." Note that the $\oplus$ in the last premise is a math operator; the $\oplus$ in the conclusion is 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):

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

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

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.

Neither attribute grammars nor inference rules actually 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 an extended (AG) example for declaration and type checking in the extended calculator grammar. Similar checking can be written as a set of inference rules (and will be in the 5th edition of the text). In either case, we tag AST nodes with essential information (e.g., type, scope). We can also pass more general information (symbol table, error messages) among nodes, 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
for calculator language, which has no scopes, can enforce
    declare-before-use in a simple left-to-right traversal of the tree
        - complain at any re-definition
        - or any use w/out prior definition

To avoid cascading errors, it's common to have an "error" value for an attribute that means "I already complained about this." So, for example, in

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

We label the '+' tree node with type "error" so we don't generate a second message for the ":=" node.

A few example recursive routines (with error list and symtab as globals):

```
int_decl (x, s):        //  s  is rest of program
    if <x.name, ?, ?> ∈ symtab
        errors.insert("redefinition of" x.name, this.location)
    else
        symtab.insert(<x.name, int, 0>)
    s()              //  call routine for appropriate variant

var (x):
    if <x.name, t, v> ∈ symtab
        this.type := t;  this.value := v
    else
        errors.insert(x.name "undefined", x.location)
        this.type := error;  this.value := ⊥  // undefined

bin_op(e1, o, e2):
    e1(); e2()        // call routines for appropriate variants
    if e1.type = error or e2.type = error
        this.type := error;  this.value := ⊥
    else if e1.type <> e2.type
        this.type := error;  this.value := ⊥
        errors.insert("type clash", this.location)
    else
        this.type := e1.type;
        this.value := o.op(e1.value, e2.value)
```

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.

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 has 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, if whenever the fact that x has type τ in Γ we know that the value of x in E has type τ, then E is well formed in Γ.