Semantic Analysis

(a short summary of material from Chap. 4)
• We enforce rules in local neighborhoods of a parse or syntax tree.
• So, for example, in a syntax tree:
  – look up A in symtab
  – look up B in symtab
  – annotate A & B nodes with type, or error if not defined
  – annotate “+” node with type, or error if A & B don’t have compatible types
• We describe the rules with *semantic functions*, each of which has access to *attributes* of the current neighborhood in the tree.
• There are lots of ways to go about this. In the next assignment you will:
  – build an explicit parse tree
  – traverse the parse tree to build a syntax tree
  – traverse the syntax tree to enforce rules
• A real compiler is more likely to combine the first two steps:
  – build the syntax tree during the parse, using action routines
  – traverse the syntax tree to enforce rules
Building a syntax tree while parsing

- Don’t need a parse tree if you have some other place to store attributes of active productions
  - local variables in recursive descent
  - attribute stack in table-driven parser
Action routines to build syntax tree for exprs

\[
E \rightarrow T \{ \text{TT.st := T.ptr} \} \quad TT \{ \text{E.ptr := TT.ptr} \}
\]

\[
TT_1 \rightarrow + T \{ \text{TT}_2.st := \text{make_bin_op("+", TT_1.st, T.ptr)} \} \quad TT_2 \{ \text{TT}_1.ptr := TT_2.ptr \}
\]

\[
TT_1 \rightarrow - T \{ \text{TT}_2.st := \text{make_bin_op("-", TT_1.st, T.ptr)} \} \quad TT_2 \{ \text{TT}_1.ptr := TT_2.ptr \}
\]

\[
TT \rightarrow \epsilon \{ \text{TT.ptr := TT.st} \}
\]

\[
T \rightarrow F \{ \text{FT.st := F.ptr} \} \quad FT \{ \text{T.ptr := FT.ptr} \}
\]

\[
FT_1 \rightarrow * F \{ \text{FT}_2.st := \text{make_bin_op("\times", FT_1.st, F.ptr)} \} \quad FT_2 \{ \text{FT}_1.ptr := FT_2.ptr \}
\]

\[
FT_1 \rightarrow / F \{ \text{FT}_2.st := \text{make_bin_op("\div", FT_1.st, F.ptr)} \} \quad FT_2 \{ \text{FT}_1.ptr := FT_2.ptr \}
\]

\[
FT \rightarrow \epsilon \{ \text{FT.ptr := FT.st} \}
\]

\[
F_1 \rightarrow - F_2 \{ \text{F}_1.ptr := \text{make_un_op("+/-", F_2.ptr)} \}
\]

\[
F \rightarrow ( E ) \{ \text{F.ptr := E.ptr} \}
\]

\[
F \rightarrow \text{const} \{ \text{F.ptr := make_leaf(const.ptr)} \}
\]
How does the parser actually call these routines?

1. Embed them in the arms of a recursive descent case statement
   – (some) inherited attributes are parameters to routine
   – (some) synthesized attributes are return values
   – other attributes may just be local variables

```plaintext
proc factor_tail(st):
    case next_tok of
      * : match(*); f_t = factor();
      ft_t = factor_tail(new bin_op(*, st, f_t));
    return ft_t;
...
How does the parser actually call these routines?

2. Push markers into the stack of a table-driven parser
   - parser generator makes attributes of current production available as lhs and rhs[]

\[
\text{FT} \rightarrow * \text{ F #1 FT #2}
\]

\[
\text{lhs rhs[0 1 2]}
\]

do\_action(n):
  case n of
    1 : rhs[2].st =
        new bin\_op(*, lhs.st, rhs[1].t)
    2 : lhs.t = rhs[2].t
• So: action routines in parser, with automatic space management for attributes, suffice to build a syntax tree
• (In the next assignment we’ll build the parse tree anyway, using a simplified attribute stack)
• Ad hoc traversal of syntax tree then enforces semantic rules
• But attribute grammar is still a handy way to think about the rules for that ad hoc traversal
  – tree grammar
### Calculator language with types

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>program</code></td>
<td><code>stmt_list $\$$</code></td>
</tr>
<tr>
<td><code>stmt_list</code></td>
<td>`decl stmt_list</td>
</tr>
<tr>
<td><code>decl</code></td>
<td>`int id</td>
</tr>
<tr>
<td><code>stmt</code></td>
<td>`id := expr</td>
</tr>
<tr>
<td><code>expr</code></td>
<td><code>term term_tail</code></td>
</tr>
<tr>
<td><code>term_tail</code></td>
<td>`add_op term term_tail</td>
</tr>
<tr>
<td><code>term</code></td>
<td><code>factor factor_tail</code></td>
</tr>
<tr>
<td><code>factor_tail</code></td>
<td>`mult_op factor factor_tail</td>
</tr>
<tr>
<td><code>factor</code></td>
<td>`(expr)</td>
</tr>
<tr>
<td><code>add_op</code></td>
<td>`+</td>
</tr>
<tr>
<td><code>mult_op</code></td>
<td>`*</td>
</tr>
</tbody>
</table>
• A simple program and a possible syntax tree:

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

```
program
  int_decl
    read
      a
    real_decl
      read
        a
        b
      write
        b
        null
        2.0
        +
        float
          b
          a
```
• Decorate tree via ad hoc traversal; describe the rules with an AG based on a *tree grammar*

```
program → item
int_decl : item → id item
read : item → id item
real_decl : item → id item
write : item → expr item
null : item → ε
'÷' : expr → expr expr
'+' : expr → expr expr
float : expr → expr
id : expr → ε
real_const : expr → ε
```
Semantic functions for that (partial) tree AG fill three pages in the book. Here are the attributes:

<table>
<thead>
<tr>
<th>Class of node</th>
<th>Variants</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>program</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>item</td>
<td>int_decl, real_decl, read, write, :, null</td>
<td>symtab, errors_in, location, errors_out</td>
</tr>
<tr>
<td>expr</td>
<td>int_const, real_const, id, +, -, x, ÷, float, trunc</td>
<td>symtab, location, type, errors, name (id only)</td>
</tr>
</tbody>
</table>
And here is the flow.

Remember: the traversal is ad hoc.

ei = errors_in
eo = errors_out
e = errors
s = symtab
t = type
n = name

location attribute not shown