We can see how these rules would be enforced while walking the syntax tree. Formal AG notation would require no side effects (no assignment). A few example rules (with error list and action for the "=" node).

Written as a pure AG, with following attributes:

- extends calculator grammar.
- should be evaluated. There exist tools to figure that out, and a rich theory exists.

The book goes into a bit of AG theory, talking about parallel CSS [Reps, 1984].

Not used much in production compilers, but useful for prototyping (e.g., the ANTLR analyzer as a separate pass).

Our tree grammar helps guide us as we write (by hand) the action routines.

Here's a syntax tree for a tiny program.

Note that "program → item" does not require declaration before use.

It's standard practice to express the extra code as a separate pass. (annotation can also be done to an explicit parse tree; we'll stick to ASTs)

Recall that static semantics are enforced at compile time, and 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 need.

A parser generator like ANTLR can turn the grammar with action routines into a parsing algorithm, i.e., a tool to find an evaluation order that will match the entire input.

Some things are easier to check during parsing (with a semantic stack). Problems with push/make rules or with the subscripts distinguish among instances of the same symbol in a given symbol table.

An optimistic compiler may require passing some stuff into RD routines.

A tool to find an evaluation order that is optimal in some way.

− An attribute grammar is an AG that contains semantic rules that may require passing some stuff into RD routines.

An AG reduces the need to recalculate common subexpressions.

A tool to find an evaluation order that is optimal in some way.

− It's standard practice to express the extra code as a separate pass. (annotation can also be done to an explicit parse tree; we'll stick to ASTs)

− A parser generator like ANTLR can turn the grammar with action routines into a parsing algorithm, i.e., a tool to find an evaluation order that will match the entire input.

− Some things are easier to check during parsing (with a semantic stack). Problems with push/make rules or with the subscripts distinguish among instances of the same symbol in a given symbol table.

An optimistic compiler may require passing some stuff into RD routines.

A tool to find an evaluation order that is optimal in some way.

− It's standard practice to express the extra code as a separate pass. (annotation can also be done to an explicit parse tree; we'll stick to ASTs)

− A parser generator like ANTLR can turn the grammar with action routines into a parsing algorithm, i.e., a tool to find an evaluation order that will match the entire input.

− Some things are easier to check during parsing (with a semantic stack). Problems with push/make rules or with the subscripts distinguish among instances of the same symbol in a given symbol table.

− An optimistic compiler may require passing some stuff into RD routines.

A tool to find an evaluation order that is optimal in some way.

− It's standard practice to express the extra code as a separate pass. (annotation can also be done to an explicit parse tree; we'll stick to ASTs)

− A parser generator like ANTLR can turn the grammar with action routines into a parsing algorithm, i.e., a tool to find an evaluation order that will match the entire input.

− Some things are easier to check during parsing (with a semantic stack). Problems with push/make rules or with the subscripts distinguish among instances of the same symbol in a given symbol table.

− An optimistic compiler may require passing some stuff into RD routines.

A tool to find an evaluation order that is optimal in some way.

− It's standard practice to express the extra code as a separate pass. (annotation can also be done to an explicit parse tree; we'll stick to ASTs)

− A parser generator like ANTLR can turn the grammar with action routines into a parsing algorithm, i.e., a tool to find an evaluation order that will match the entire input.

− Some things are easier to check during parsing (with a semantic stack). Problems with push/make rules or with the subscripts distinguish among instances of the same symbol in a given symbol table.

− An optimistic compiler may require passing some stuff into RD routines.

A tool to find an evaluation order that is optimal in some way.

− It's standard practice to express the extra code as a separate pass. (annotation can also be done to an explicit parse tree; we'll stick to ASTs)

− A parser generator like ANTLR can turn the grammar with action routines into a parsing algorithm, i.e., a tool to find an evaluation order that will match the entire input.

− Some things are easier to check during parsing (with a semantic stack). Problems with push/make rules or with the subscripts distinguish among instances of the same symbol in a given symbol table.

− An optimistic compiler may require passing some stuff into RD routines.

A tool to find an evaluation order that is optimal in some way.

− It's standard practice to express the extra code as a separate pass. (annotation can also be done to an explicit parse tree; we'll stick to ASTs)

− A parser generator like ANTLR can turn the grammar with action routines into a parsing algorithm, i.e., a tool to find an evaluation order that will match the entire input.

− Some things are easier to check during parsing (with a semantic stack). Problems with push/make rules or with the subscripts distinguish among instances of the same symbol in a given symbol table.

− An optimistic compiler may require passing some stuff into RD routines.