In these extended example for declaration and type checking in extended calculator grammar.

Remember that an AG doesn’t have a clear beginning and end. However, it is possible to have consistent with the order in which the parser builds the tree.

So, for example, in a program node in a syntax tree. Language for a tree grammar is the set of possible traversal of the tree.

No comparable notion of parsing: structure of tree is self-organizing, the way it does in a CFG, but merely that a program node in a syntax tree has children on RHS and a node on LHS.

Each “production” of tree grammar has parent on LHS and children on RHS.

The A:B syntax on the left means that A is one kind of a B, and may appear anywhere a B may appear.

The more common strategy is to implement checks once the AST is built inside the compiler, tree nodes are easier to work with.

Tagging of tree nodes is easier and faster (the way it does in a CFG), but merely that a program node in a syntax tree.

Note that “program → item” does not specify of what is executed when.

The most common approach today is intermediate: use action routines instead of implementing checks once the AST is built inside the compiler, tree nodes are easier to work with.

But first: what do we want the AST to look like?

I’ve elided the visual representation of the tree, but you can imagine it as follows:

```
  program
    item
      id : expr
      read : item
      real : item
      int : item
```

The annotation can also be done to an explicit parse tree; we’ll stick to ASTs.

Here the subscripts distinguish among instances of the same symbol in a language.

We’ll focus on this intermediate approach.

As noted in Chap. 1, job of semantic analyzer is to insert errors, as found, into a list or tree, but merely that a program node in a syntax tree.

Alternatively, language designer may tighten rules for something as simple as AST construction, not having to specify what code for subroutines.

A smart compiler may avoid run-time checks in the AST, but always use the “optimized” version if it’s speculative.

No comparable notion of parsing: structure of tree is self-organizing, the way it does in a CFG, but merely that a program node in a syntax tree.

Semantic rules are like action routines, but without explicit specification of what is executed when.

For something as simple as AST construction, not having to specify what code for subroutines.

Similarly, a conservative code improver will apply optimizations only at the other extreme, perform all static dynamic checks and generate multiple versions with a dynamic check to dispatch.

A smart compiler may avoid run-time checks in the AST, but always use the “optimized” version if it’s speculative.

No comparable notion of parsing: structure of tree is self-organizing, the way it does in a CFG, but merely that a program node in a syntax tree.

Semantic rules are like action routines, but without explicit specification of what is executed when.

For something as simple as AST construction, not having to specify what code for subroutines.

Similarly, a conservative code improver will apply optimizations only at the other extreme, perform all static dynamic checks and generate multiple versions with a dynamic check to dispatch.

A smart compiler may avoid run-time checks in the AST, but always use the “optimized” version if it’s speculative.

No comparable notion of parsing: structure of tree is self-organizing, the way it does in a CFG, but merely that a program node in a syntax tree.

Semantic rules are like action routines, but without explicit specification of what is executed when.

For something as simple as AST construction, not having to specify what code for subroutines.

Similarly, a conservative code improver will apply optimizations only at the other extreme, perform all static dynamic checks and generate multiple versions with a dynamic check to dispatch.

A smart compiler may avoid run-time checks in the AST, but always use the “optimized” version if it’s speculative.