CS46K Programming Languages

Structure and Interpretation of Computer Programs

Chapter 9B Syntax Analysis – Parser Design

LECTURE 13: ABSTRACT SYNTAX TREE

DR. ERIC CHOU

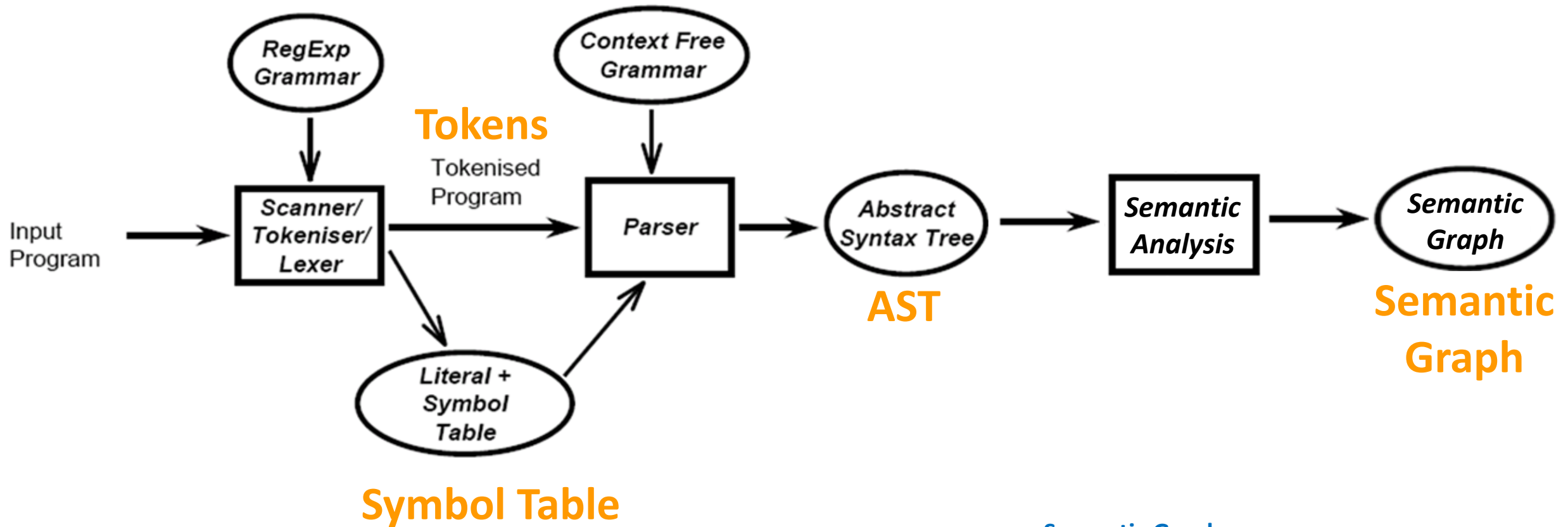IEEE SENIOR MEMBER

Objectives

- Attribute Grammar

- Evaluation of Attributes

- Action Routine

- Tree Grammars and Syntax Tree Decoration

- Parser Generators

- Parser Combinators

- Python Libraries Related to Parsing

Systematic Compiler Front-End Generation

Semantic Graph:
AST with Additional Properties and Resolved References

Attribute Grammar I

Attribute Grammars

- Both semantic analysis and (intermediate) code generation can be described in terms of annotation, or "decoration" of a parse or syntax tree

- Attribute Grammars provide a formal framework for decorating such a tree

- The notes below discuss attribute grammars and their ad-hoc cousins, Action Routines

Attribute Grammars

- We'll start with decoration of parse trees, then consider syntax trees

- Consider the following LR (bottom-up) grammar for arithmetic expressions made of constants, with precedence and associativity:

Attribute Grammars

- The attribute grammar serves to define the semantics of the input program

- Attribute rules are best thought of as definitions, not assignments

- They are not necessarily meant to be evaluated at any particular time, or in any particular order, though they do define their left-hand side in terms of the right-hand side

Bottom-Up CFG for Constant Expressions

- This grammar will generate all properly formed constant expressions over the basic arithmetic operators, but it says nothing about their meaning. To tie these expressions to mathematical concepts, we need **additional notation**.
- The most common is based on attributes.
- In our expression grammar, we can associate a **val** attribute with each **E**, **T**, **F**, and **const** in the grammar. The intent is that for any symbol **S**, **S.val** will be the meaning, as an arithmetic value, of the token string derived from **S**.

$$E \longrightarrow E + T$$

$$E \longrightarrow E - T$$

$$E \longrightarrow T$$

$$T \longrightarrow T * F$$

$$T \longrightarrow T / F$$

$$T \longrightarrow F$$

$$F \longrightarrow - F$$

$$F \longrightarrow (E)$$

$$F \longrightarrow const$$

Bottom-Up CFG for Constant Expressions

- We assume that the **val** of a **const** is provided to us by the scanner. We must hen invent a set of rules for each production, to specify how the **val**s of different symbols are related. The resulting attribute grammar (**AG**) is shown in Figure 4.1.

$$E \longrightarrow E + T$$
$$E \longrightarrow E - T$$
$$E \longrightarrow T$$
$$T \longrightarrow T * F$$
$$T \longrightarrow T / F$$
$$T \longrightarrow F$$
$$F \longrightarrow - F$$
$$F \longrightarrow (E)$$
$$F \longrightarrow \text{const}$$

Attribute Grammar (AG) from CFG

1. $E_1 \longrightarrow E_2 + T$ $\triangleright$ E_1.val := sum(E_2.val, T.val)

2. $E_1 \longrightarrow E_2 - T$ $\triangleright$ E_1.val := difference(E_2.val, T.val)

3. $E \longrightarrow T$ $\triangleright$ E.val := T.val

4. $T_1 \longrightarrow T_2 * F$ $\triangleright$ T_1.val := product(T_2.val, F.val)

5. $T_1 \longrightarrow T_2 / F$ $\triangleright$ T_1.val := quotient(T_2.val, F.val)

6. $T \longrightarrow F$ $\triangleright$ T.val := F.val

7. $F_1 \longrightarrow - F_2$ $\triangleright$ F_1.val := additive_inverse(F_2.val)

8. $F \longrightarrow (E)$ $\triangleright$ F.val := E.val

9. $F \longrightarrow$ const $\triangleright$ F.val := const.val

Attribute Grammar - Relaxed

- In a strict definition of attribute grammars, copy **rules** and **semantic function calls** are the only two kinds of permissible rules. In our examples, we use a ▷ symbol to introduce each code fragment corresponding to a single rule. In practice, it is common to allow rules to consist of small fragments of code in some well-defined notation (e.g., the language in which a compiler is being written), so that simple semantic functions can be written out "**in-line.**"

Attribute Grammar - Relaxed

- In relaxed notation, the rule for the first production in ▷
Figure 4.1 might be simply E_1.val := E_2.val T.val. As another
example, suppose we wanted to count the elements of a
comma-separated list:

L -> id LT	▷ L.c := 1 + LT.c
LT -> , L	▷ LT.c := L.c
LT -> ε	▷ LT.c := 0

Note: rule 1 sets the LHS count to be 1 greater than RHS. Like explicit semantic functions, in-line rules are not allowed to refer to any variables
or attributes outside the current productions. We will relax this restriction when we introduce action routines in Section4.4.

Attribute Grammar

- The purpose of AG is to associate meaning with the grammar. We can use any notation and types whose meaning are already well defined.

- In Examples 4.4 and 4.5, we associated numeric values with the symbols in CFG – and thus with parse tree nodes – using semantic functions drawn from ordinary arithmetic.

- In a compiler or interpreter for a full programming language, the **attributes** of tree nodes might include:

Note: For purposes other than translation – e.g., in a theorem prover or machine-independent language definition – attributes might be drawn from the disciplines of denotational, operational, or axiomatic semantics.

Attribute Grammar

1. for an identifier, a **reference** to information about it in the symbol table.

2. for an expression, its **type**

3. for a statement expression, a r**eference** to corresponding code in the compiler's Intermediate form.

4. for almost **construct indication** of the file name, line, and column where the corresponding source code begins

5. for any internal node, a list of **semantic errors** found in the subtree below

Attribute Grammar II

Attribute Grammar

- **Attribute grammar** is a special form of **context-free grammar** where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.

Attribute Grammar

- **Attribute grammar** is a medium to provide **semantics** to the **context-free grammar** and it can help specify the syntax and semantics of a programming language.

- Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

Example:

```
E → E + T { E.value = E.value + T.value }
```

The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

Example:

Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions. Based on the way the attributes get their values, they can be broadly divided into two categories : synthesized attributes and inherited attributes.

Attribute Grammars

- Carry some semantic information along through parse tree

- Useful for
  - Static semantic specification
  - Static semantic checking in compilers

Attribute Grammars

- An attribute grammar is a CFG G = (S, N, T, P) with annotations
  - For each grammar symbol x, there is a set A(x) of **attribute values**
  - Each production rule has a set of functions that define certain attributes of the non-terminals in the rule.
  - Each production rule has a (possibly empty) **set of predicates** to check for attribute consistency
    - Valid derivations have predicates true for each node

Attribute Grammars

- Synthesized attributes
  - Are determined from nodes of children in parse tree
    - If $X_0 \rightarrow X_1 \ldots X_n$ is a rule, the $S(X_0) = f(A(X_1), \ldots, A(X_n))$
    - the value of X_0 determined by children
  - Pass semantic information up the tree

Attribute Grammars

- Inherited attributes
  - Are determined from parent and siblings
    - $I(X_j) = f(A(X_0), \ldots, A(X_n))$, Often just $X_0 \ldots X_{j-1}$
    - Siblings to the left in parse tree
  - Pass semantic information down the tree

Synthesized attributes

- These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

$$S \rightarrow ABC$$

- If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

- As in our previous example (E → E + T), the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.
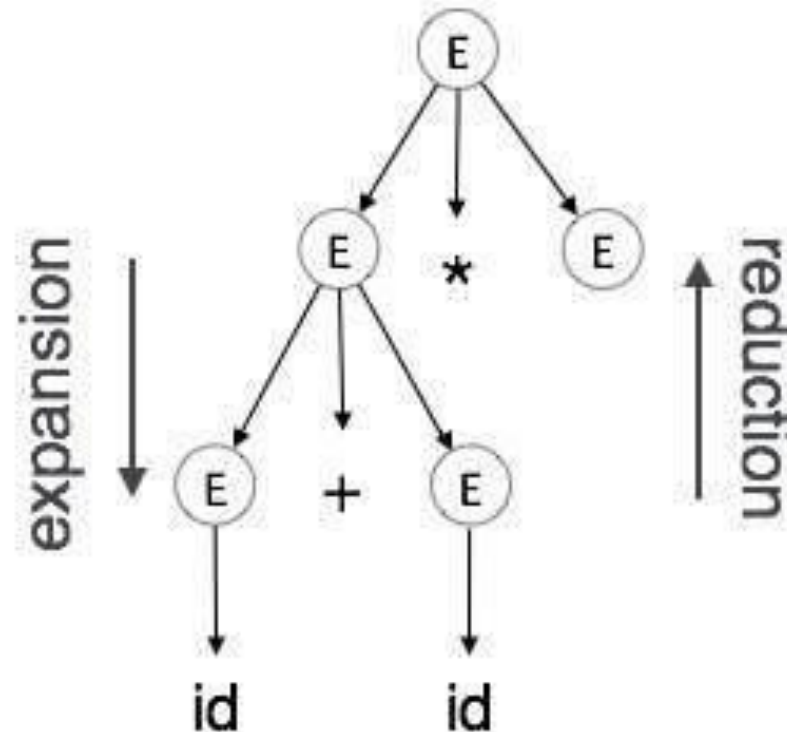
Inherited attributes

- In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,

$$S \rightarrow ABC$$

- A can get values from S, B and C.

- B can take values from S, A, and C.

- Likewise, C can take values from S, A, and B.

Expansion

When a non-terminal is expanded to terminals as per a grammatical rule:

Reduction

- When a terminal is reduced to its corresponding non-terminal according to grammar rules. Syntax trees are parsed top-down and left to right. Whenever reduction occurs, we apply its corresponding semantic rules (actions).

- Semantic analysis uses Syntax Directed Translations to perform the above tasks.

- Semantic analyzer receives **AST** (Abstract Syntax Tree) from its previous stage (syntax analysis).

- Semantic analyzer attaches attribute information with **AST**, which are called **Attributed AST**.

- Attributes are two tuple value, **<attribute name, attribute value>**
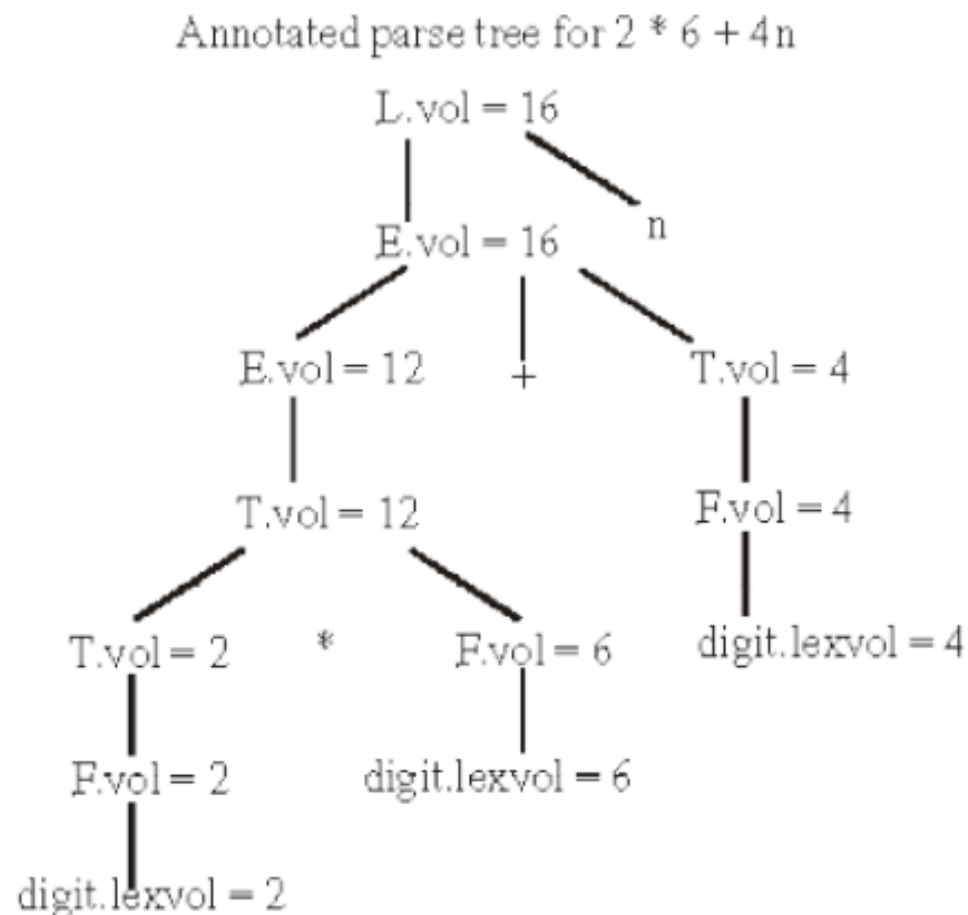
Reduction

For example:

```
int value  = 5;
<type, "integer">
<presentvalue, "5">
```

Note: For every production, we attach a semantic rule.

Syntax directed translation schemes (SDT)

Example : An annotated parse tree for 2*6+ 4n

Annotated parse tree for 2 * 6 + 4n

```
                    L.vol = 16
                   /        \
            E.vol = 16        n
           /    |    \
     E.vol = 12  +    T.vol = 4
         |              |
     T.vol = 12      F.vol = 4
      /     \           |
 T.vol = 2  *  F.vol = 6   digit.lexvol = 4
    |              |
 F.vol = 2    digit.lexvol = 6
    |
 digit.lexvol = 2
```

Syntax Directed Translation

The conceptual view of syntax-directed translation,

Input string → parse tree → dependency graph → evaluation order for semantic rules

Syntax directed definitions (SDD)

This is a generalization of context free grammar in which each grammar symbol has an associated set of attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions.
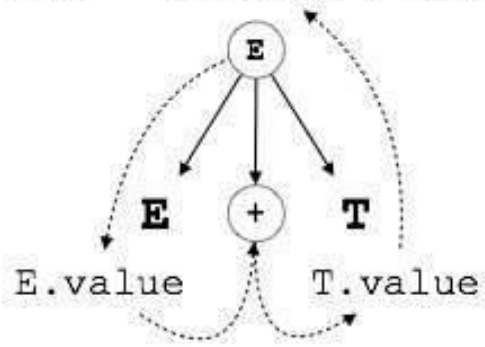
S-attributed SDT

If an SDT uses only synthesized attributes, it is called as **S-attributed SDT**. These attributes are evaluated using **S-attributed SDTs** that have their semantic actions written after the production (right hand side).

As depicted above, attributes in **S-attributed SDT**s are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

[LR-parsing]

```
E.value = E.value + T.value
```

L-attributed SDT

- This form of **SDT** uses both synthesized and inherited attributes with restriction of not taking values from right siblings.

- In **L-attributed SDTs**, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production
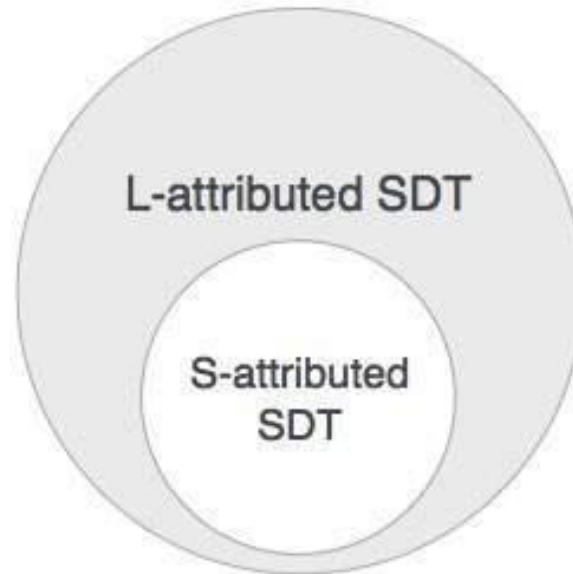
```
S → ABC
```

- S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right.

L-attributed SDT

- Attributes in **L-attributed SDTs** are evaluated by **depth-first** and **left-to-right** parsing manner.

- We may conclude that if a definition is **S-attributed**, then it is also **L-attributed** as **L-attributed** definition encloses **S-attributed** definitions.

Evaluating Attributes

Evaluating Attributes

- This is a very simple attribute grammar:

  - Each symbol has at most one **attribute**

    - the punctuation marks have no attributes

- These attributes are all so-called **Synthesized** attributes:

  - They are calculated only from the attributes of things below them in the parse tree

- In general, we are allowed both **synthesized** and **Inherited** attributes:

  - Inherited attributes may depend on things above or to the side of them in the parse tree

  - Tokens have only synthesized attributes, initialized by the scanner (name of an identifier, value of a constant, etc.).

  - Inherited attributes of the **start** symbol constitute run-time parameters of the compiler

Evaluating Attributes

- The process of evaluating attributes is called **annotation**, or **Decoration**, of the parse tree

  - When a parse tree under this grammar is fully decorated, the value of the expression will be in the **val** attribute of the root

- The code fragments for the rules are called **Semantic Functions**

  - Strictly speaking, they should be cast as functions,

    e.g., **E1.val = sum (E2.val, T.val)**

Evaluating Attributes

(1 + 3) * 2
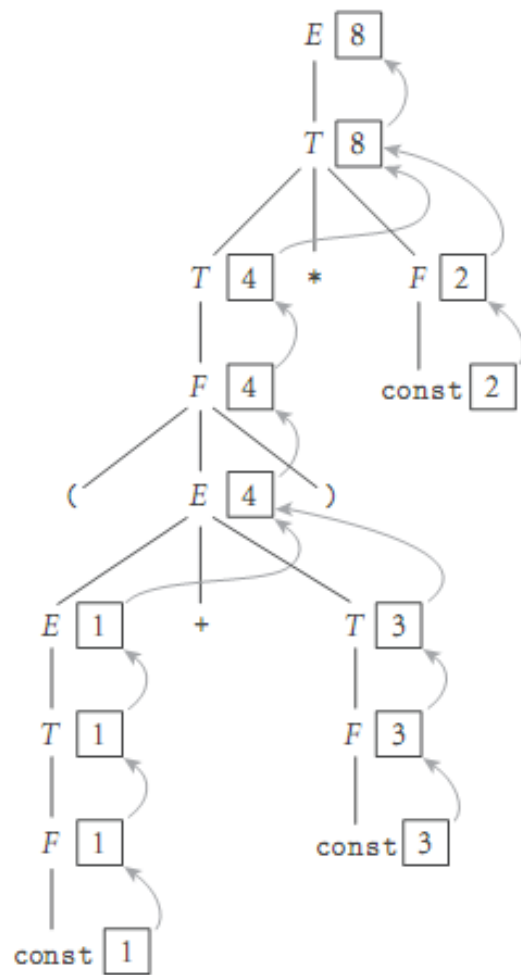
Figure 4.2 Decoration of a parse tree for (1 + 3) * 2, using the attribute grammar of Figure 4.1. The val attributes of symbols are shown in boxes. Curving arrows show the attribute flow, which is strictly upward in this case. Each box holds the output of a single semantic rule; the arrow(s) entering the box indicate the input(s) to the rule. At the second level of the tree, for example, the two arrows pointing into the box with the 8 represent application of the rule $T_1.val := product(T_2.val, F.val)$.

Top-down CFG and parse tree for subtraction

- As a simple example of inherited attributes, consider the following simplified fragment of an LL(1) expression grammar (here covering only subtraction):

$expr \longrightarrow$ const $expr\_tail$

$expr\_tail \longrightarrow$ - const $expr\_tail \mid \epsilon$

For the expression 9 − 4 − 3, we obtain the following parse tree:
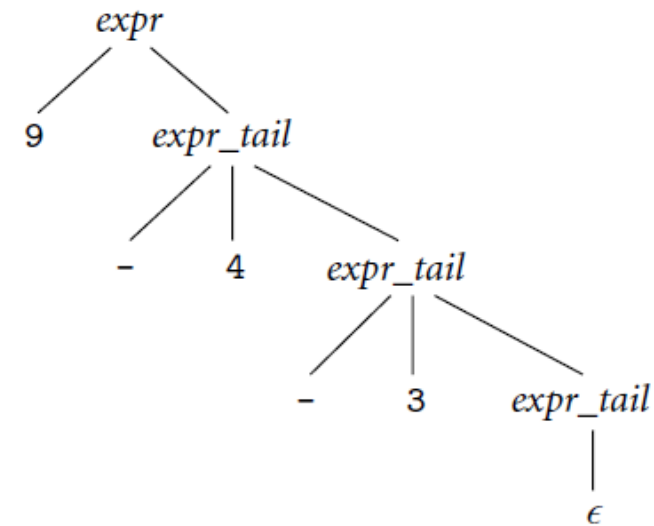
Top-down CFG and parse tree for subtraction

- If we want to create an attribute grammar that accumulates the value of the overall expression into the root of the tree, we have a problem: because **subtraction** is left associative, we cannot summarize the right subtree of the root with a single numeric value. **[No SLR]**
- If we want to decorate the tree bottom-up, with an **S-attributed** grammar, we must be prepared to describe an arbitrary number of right operands in the attributes of the top-most **expr_tail** node.
- This is indeed possible, but it defeats the purpose of the formalism: in effect, it requires us to embed the entire tree into the attributes of a single node, and do all the real work inside a single semantic function.

Note: LL(1) Grammar does not fit S-attributed grammar.

Decoration with Left-to-Right Attribute Flow (L-attributed)

- If, however, we are allowed to pass attribute values not only left-to-right attribute flow also left-to-right in the tree, then we can pass the 9 into the top-most **expr_tail** node, where it can be combined (in proper **left-associative fashion**) with the 4.
- The resulting 5 can then be passed into the middle expr tail node, combined with the 3 to make 2, and then passed upward to the root:

Decoration with Left-to-Right Attribute Flow (L-attributed)
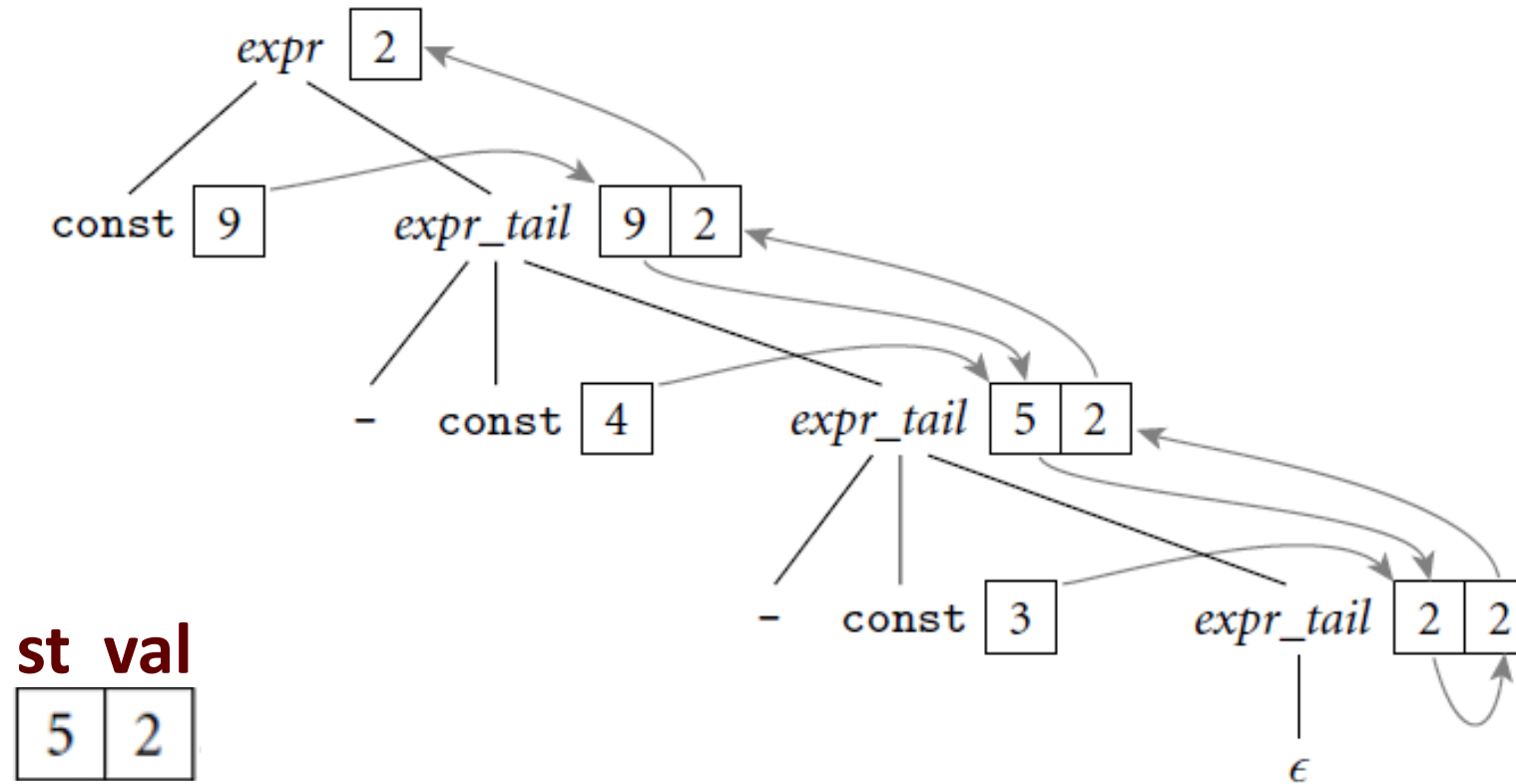
Top-down AG for Subtraction

To effect this style of decoration, we need the following attribute rules:

$$expr \longrightarrow \mathbf{const}\ expr\_tail$$
$$\triangleright\ expr\_tail.st := const.val$$
$$\triangleright\ expr.val := expr\_tail.val$$

$$expr\_tail_1 \longrightarrow \ -\ \mathbf{const}\ expr\_tail_2$$
$$\triangleright\ expr\_tail_2.st := expr\_tail_1.st - const.val$$
$$\triangleright\ expr\_tail_1.val := expr\_tail_2.val$$

$$expr\_tail \longrightarrow \ \epsilon$$
$$\triangleright\ expr\_tail.val := expr\_tail.st$$

| st	val
5	2

- In each of the first two productions, the first rule serves to copy the left context into a "subtotal" (**st**) attribute; the second rule copies the final value from the right-most leaf back up to the root.
- In the **expr_tail** nodes of the picture in Example 4.8, the left box holds the **st** attribute; the right holds **val**.

1. $E \longrightarrow T \; TT$
   - ▷ TT.st := T.val
   - ▷ E.val := TT.val

2. $TT_1 \longrightarrow + \; T \; TT_2$
   - ▷ TT_2.st := TT_1.st + T.val
   - ▷ TT_1.val := TT_2.val

3. $TT_1 \longrightarrow - \; T \; TT_2$
   - ▷ TT_2.st := TT_1.st − T.val
   - ▷ TT_1.val := TT_2.val

4. $TT \longrightarrow \epsilon$
   - ▷ TT.val := TT.st

5. $T \longrightarrow F \; FT$
   - ▷ FT.st := F.val
   - ▷ T.val := FT.val

6. $FT_1 \longrightarrow * \; F \; FT_2$
   - ▷ FT_2.st := FT_1.st × F.val
   - ▷ FT_1.val := FT_2.val

7. $FT_1 \longrightarrow / \; F \; FT_2$
   - ▷ FT_2.st := FT_1.st ÷ F.val
   - ▷ FT_1.val := FT_2.val

8. $FT \longrightarrow \epsilon$
   - ▷ FT.val := FT.st

9. $F_1 \longrightarrow - \; F_2$
   - ▷ F_1.val := − F_2.val

10. $F \longrightarrow (\; E \;)$
    - ▷ F.val := E.val

11. $F \longrightarrow \text{const}$
    - ▷ F.val := const.val

Figure 4.3

Top-Down AG for Constant Expression

- We can flesh out the grammar fragment of to produce amore complete expression grammar, as shown (with shorter symbol names) in Figure 4.3.

- The underlying **CFG** for this grammar accepts the same language as the one in **Figure 4.1**, but where that one was **SLR(1)**, this one is **LL(1)**. Attribute flow for a parse of **(1 + 3) * 2**, using the **LL(1)** grammar, appears in **Figure 4.4**.

- As in the grammar fragment of Example 4.9, the value of the left operand of each operator is carried into the **TT** and **FT** productions by the **st (subtotal)** attribute.

- The relative complexity of the **attribute flow** arises from the fact that operators are left associative, but the grammar cannot be left recursive: the left and right operands of a given operator are thus found in separate productions. Grammars to perform semantic analysis for practical languages generally require some **non–S-attributed** flow.
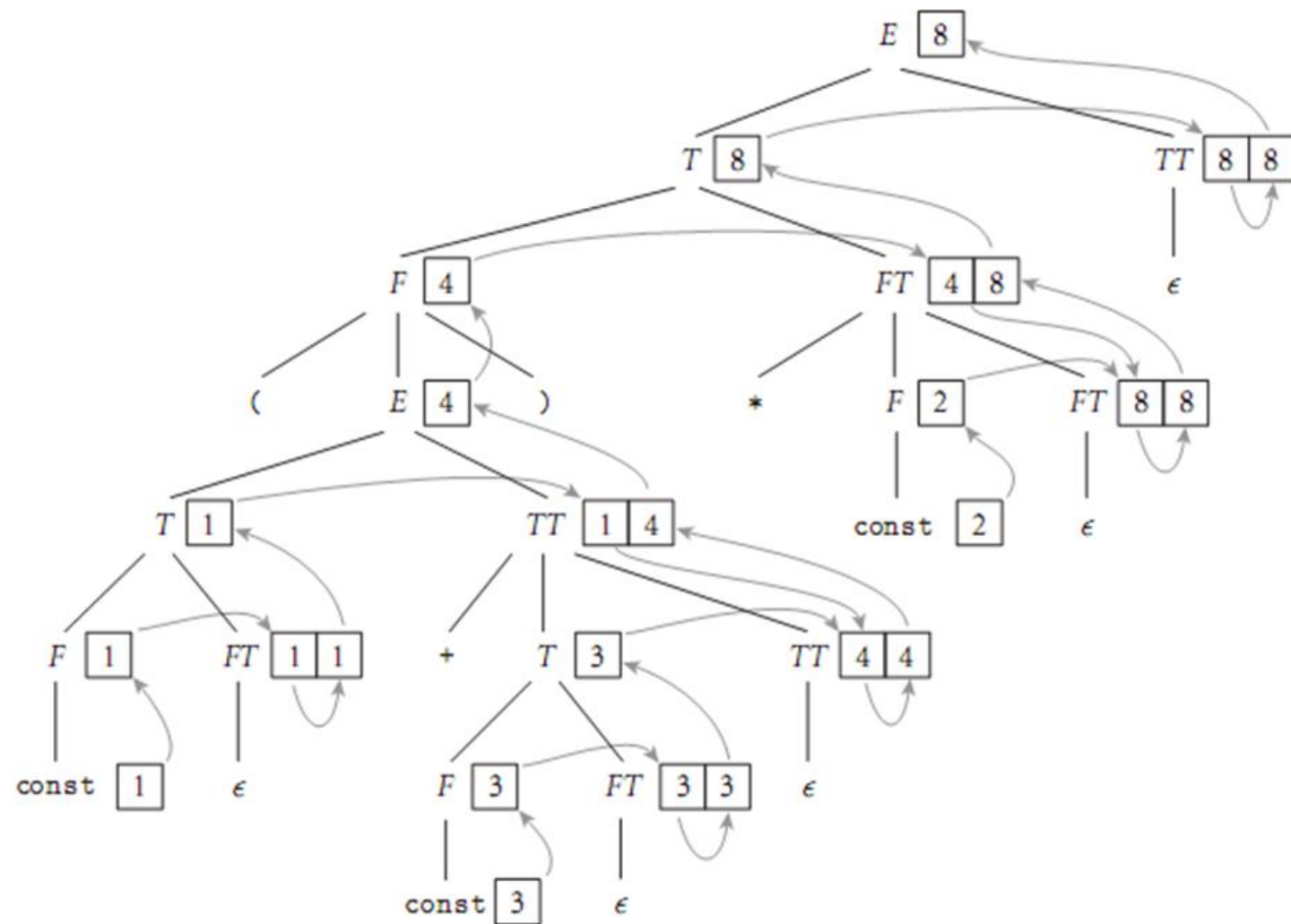
Figure 4.4 Decoration of a top-down parse tree for (1 + 3) * 2, using the AG of Figure 4.3. Curving arrows again indicate attribute flow; the arrow(s) entering a given box represent the application of a single semantic rule. Flow in this case is no longer strictly bottom-up, but it is still left-to-right. At *FT* and *TT* nodes, the left box holds the st attribute; the right holds **val**.

Evaluating Attributes

- This attribute grammar is a good bit messier than the first one, but it is still **L-Attributed**, which means that the attributes can be evaluated in a single left-to-right pass over the input

- In fact, they can be evaluated during an LL parse

- Each synthetic attribute of a **LHS** symbol (by definition of **synthetic**) depends only on attributes of its **RHS** symbols

Evaluating Attributes

- Each inherited attribute of a **RHS** symbol (by definition of **L-attributed**) depends only on
  - inherited attributes of the **LHS** symbol, or
  - synthetic or inherited attributes of symbols to its left in the **RHS**
- **L-attributed** grammars are the most general class of attribute grammars that can be evaluated during an LL parse

Evaluating Attributes

- There are certain tasks, such as generation of code for short-circuit Boolean expression evaluation, that are easiest to express with **non-L-attributed** attribute grammars

- Because of the potential cost of complex traversal schemes, however, most real-world compilers insist that the grammar be **L-attributed**

Evaluating Attributes

Attribute Flow and Syntax Trees

SECTION 4

Attribute Flow

•An attribute flow algorithm propagates attribute values through the parse tree by traversing the tree according to the set and used dependencies between attributes (an attribute must be set before it is read)

•Design for Evaluating Attributes
1. Production Rules (CFG)
2. Action (S-Attributed G/L-Attributed G)
3. Attribute Flow (Algorithm)

Production Action	Attribute flow
`<E> -> <T> <TT>` `TT.st := T.val`	

`<TT1> -> + <T> <TT2>`
`TT2.st :=`
`    TT1.st + T.val`	

`<TT> -> ε` `TT.val := TT.st`	
`<TT1> -> + <T> <TT2>` `TT1.val := TT2.val`	
`<E> -> <T> <TT>` `E.val := TT.val`	

Attribute Flow

- Both **context-free grammar** and **attribute grammar** do not specify the order in which attribute rules should be invoked.
  - Both notations are declarative: they define a set of valid trees, but they don't say how to build or decorate them.
  - The order in which attribute rules are listed for a given production is immaterial; **attribute flow** may require them to execute in any order.
  - If, in Figure 4.3, we were to reverse the order in which the rules appear in productions 1, 2, 3, 5, 6, and/or 7 (listing the rule for **symbol.val** first), it would be a purely cosmetic change; the grammar would not be altered.

Attribute Flow

- An algorithm that decorates parse trees by invoking the rules of an attribute grammar in an order consistent with the tree's attribute flow is called a **translation scheme**.

- The simplest scheme is one that makes repeated passes over a tree, invoking any semantic function whose arguments have all been defined, and stopping when it completes a pass in which no values change. Such a scheme is said to be **oblivious**, in the sense that it exploits no special knowledge of either the parse tree or the grammar.

- It will halt only if the grammar is well defined. Better performance, at least for noncircular grammars, may be achieved by a dynamic scheme that tailors the evaluation order to the structure of a given parse tree, for example by constructing a topological sort of the attribute flow graph and then invoking rules in an order consistent with the sort.

Types of compilers

- One pass compilers
- Multi pass compilers
- Load and go compilers
- Optimizing compilers

One-Pass Compilers

- A compiler that interleaves **semantic analysis** and **code generation** with **parsing** is said to be a one-pass compiler.

- It is unclear whether interleaving **semantic analysis** with parsing makes a compiler simpler or more complex; it's mainly a matter of taste. If intermediate code generation is interleaved with parsing, one need not build a syntax tree at all (unless of course the syntax tree is the intermediate code).

One-Pass Compilers

- Moreover, it is often possible to write the intermediate code to an output file on the fly, rather than accumulating it in the attributes of the root of the parse tree. The resulting space savings were important for previous generations of computers, which had very small main memories.

- On the other hand, **semantic analysis** is easier to perform during a separate traversal of a syntax tree, because that tree reflects the program's semantic structure better than the parse tree does, especially with a top-down parser, and because one has the option of traversing the tree in an order other than that chosen by the parser.

Bottom-UP and Top-Down AGs to build a Syntax Tree

- If we choose not to interleave parsing and semantic analysis, we still need to add attribute rules to the context-free grammar, but they serve only to create the syntax tree—not to enforce semantic rules or generate code. **[gcc]**

- **Figures 4.5** and **4.6** contain bottom-up and top-down attribute grammars, respectively, to build a syntax tree for constant expressions. The attributes in these grammars hold neither numeric values nor target code fragments; instead they point to nodes of the syntax tree. Function make leaf returns a pointer to a newly allocated syntax tree node containing the value of a constant. Functions make un op and make bin op return pointers to newly allocated syntax tree nodes containing a unary or binary operator, respectively, and pointers to the supplied operand(s).

- **Figures 4.7** and **4.8** show stages in the decoration of parse trees for **(1 + 3) * 2**, using the grammars of **Figures 4.5** and **4.6**, respectively.

  Note that the final syntax tree is the same in each case.

$$E_1 \longrightarrow E_2 + T$$
▷ E_1.ptr := make_bin_op("+", E_2.ptr, T.ptr)

$$E_1 \longrightarrow E_2 - T$$
▷ E_1.ptr := make_bin_op("−", E_2.ptr, T.ptr)

$$E \longrightarrow T$$
▷ E.ptr := T.ptr

$$T_1 \longrightarrow T_2 * F$$
▷ T_1.ptr := make_bin_op("×", T_2.ptr, F.ptr)

$$T_1 \longrightarrow T_2 / F$$
▷ T_1.ptr := make_bin_op("÷", T_2.ptr, F.ptr)

$$T \longrightarrow F$$
▷ T.ptr := F.ptr

$$F_1 \longrightarrow - F_2$$
▷ F_1.ptr := make_un_op("+/_", F_2.ptr)

$$F \longrightarrow (E)$$
▷ F.ptr := E.ptr

$$F \longrightarrow \text{const}$$
▷ F.ptr := make_leaf(const.val)

Figure 4.5 Bottom-up (S-attributed) attribute grammar to construct a syntax tree.
The symbol +/− is used (as it is on calculators) to indicate change of sign.

$E \longrightarrow T\ TT$
> TT.st := T.ptr
> E.ptr := TT.ptr

$TT_1 \longrightarrow +\ T\ TT_2$
> TT_2.st := make_bin_op("+", TT_1.st, T.ptr)
> TT_1.ptr := TT_2.ptr

$TT_1 \longrightarrow -\ T\ TT_2$
> TT_2.st := make_bin_op("−", TT_1.st, T.ptr)
> TT_1.ptr := TT_2.ptr

$TT \longrightarrow \epsilon$
> TT.ptr := TT.st

$T \longrightarrow F\ FT$
> FT.st := F.ptr
> T.ptr := FT.ptr

$FT_1 \longrightarrow *\ F\ FT_2$
> FT_2.st := make_bin_op("×", FT_1.st, F.ptr)
> FT_1.ptr := FT_2.ptr

$FT_1 \longrightarrow /\ F\ FT_2$
> FT_2.st := make_bin_op("÷", FT_1.st, F.ptr)
> FT_1.ptr := FT_2.ptr

$FT \longrightarrow \epsilon$
> FT.ptr := FT.st

$F_1 \longrightarrow -\ F_2$
> F_1.ptr := make_un_op("+/_", F_2.ptr)

$F \longrightarrow (\ E\ )$
> F.ptr := E.ptr

$F \longrightarrow \text{const}$
> F.ptr := make_leaf(const.val)

Figure 4.6 Top-down (L-attributed) attribute grammar to construct a syntax tree.
Here the **st** attribute, like the **ptr** attribute (and unlike the st attribute of Figure 4.3), is a pointer to a syntax tree node.
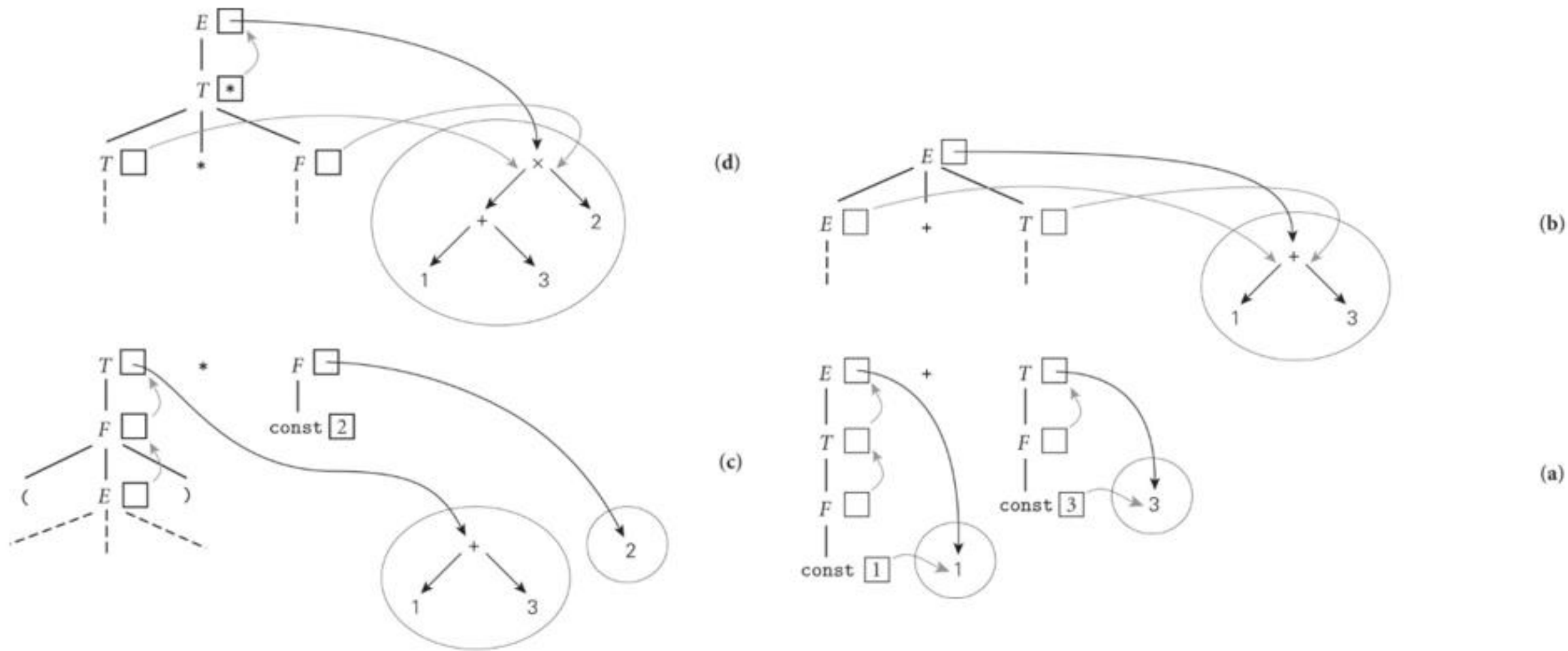
Figure 4.7 Construction of a syntax tree for (1 + 3) * 2 via decoration of a bottom-up parse tree, using the grammar of Figure 4.5. This figure reads from bottom to top. In diagram (a), the values of the constants 1 and 3 have been placed in new syntax tree leaves. Pointers to these leaves propagate up into the attributes of *E* and *T*. In (b), the pointers to these leaves become child pointers of a new internal + node. In (c) the pointer to this node propagates up into the attributes of *T*, and a new leaf is created for 2. Finally, in (d), the pointers from *T* and *F* become child pointers of a new internal × node, and a pointer to this node propagates up into the attributes of *E*.
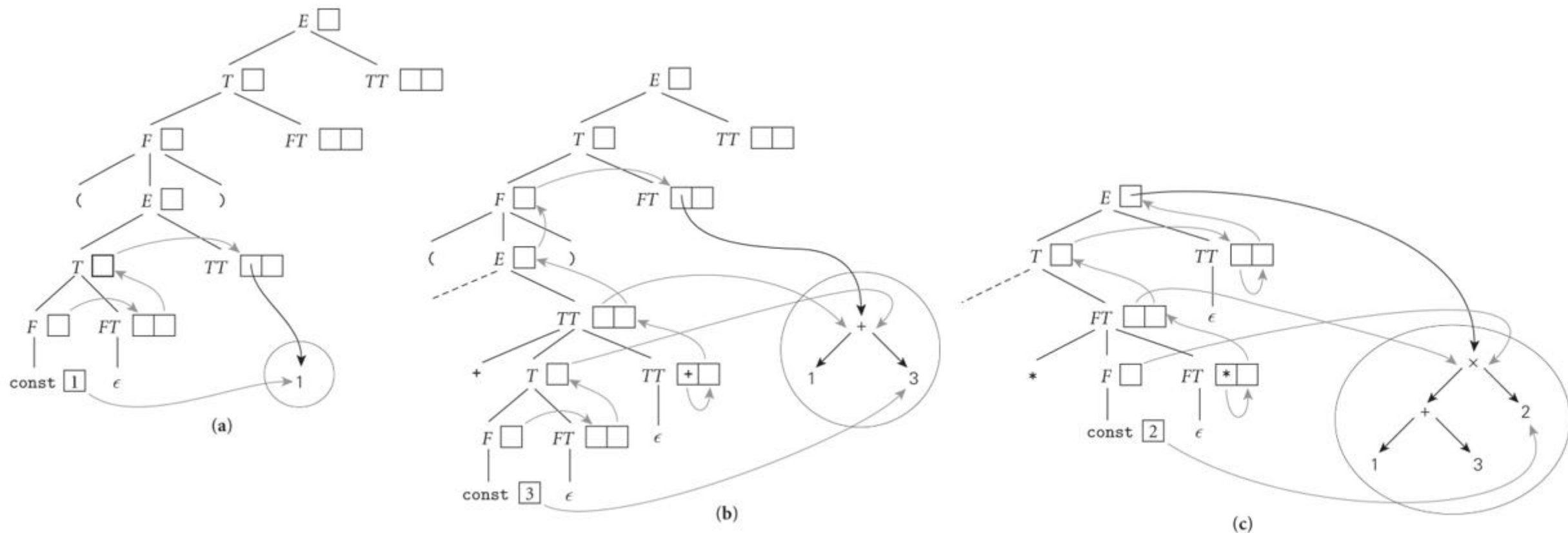
Figure 4.8 Construction of a syntax tree via decoration of a top-down parse tree, using the grammar of Figure 4.6. In the top diagram, (a), the value of the constant 1 has been placed in a new syntax tree leaf. A pointer to this leaf then propagates to the st attribute of *TT*. In (b), a second leaf has been created to hold the constant 3. Pointers to the two leaves then become child pointers of a new internal + node, a pointer to which propagates from the st attribute of the bottom-most *TT*, where it was created, all the way up and over to the st attribute of the top-most *FT*. In (c), a third leaf has been created for the constant 2. Pointers to this leaf and to the + node then become the children of a new × node, a pointer to which propagates from the st of the lower *FT*, where it was created, all the way to the root of the tree.

Action Routines

Overview of Action Routines

- There are automatic tools that will construct a **semantic analyzer (attribute evaluator)** for a given attribute grammar.

- **Applications:** Attribute evaluator generators have been used in **syntax-based editors** [RT88], **incremental compilers** [SDB84], and various aspects of programming language research.

- Most production compilers, however, use an **ad hoc**, handwritten translation scheme, interleaving parsing with at least the initial construction of a syntax tree, and possibly all of semantic analysis and intermediate code generation. Because they are able to evaluate the attributes of each production as it is parsed, they do not need to build the full parse tree.

Action Routines – f(X)

- An **Action Routine** is a **semantic function** that the programmer (grammar writer) instructs the compiler to execute at a particular point in the parse.

- Most parser generators allow the programmer to specify action routines. In an LL parser generator, an action routine can appear anywhere within a **right-hand side**.

- A routine at the beginning of a right-hand side will be called as soon as the parser predicts the production. A routine embedded in the middle of a right-hand side will be called as soon as the parser has matched the symbol to the left.

Action Routines – f(X)

- The implementation mechanism is simple: when it predicts a production, the parser pushes all of the right-hand side onto the stack, including terminals (to be matched), non-terminals (to drive future predictions), and pointers to action routines.

- When it finds a pointer to an action routine at the top of the parse stack, the parser simply calls it, passing the appropriate attributes as arguments.

Top-down Action Routines to Build a Syntax Tree

- To make this process more concrete, consider again our **LL(1)** grammar for constant expressions.

- Action routines to build a syntax tree while parsing this grammar appear in Figure 4.9. The only difference between this grammar and the one in Figure 4.6 is that the action routines (delimited here with curly braces) are embedded among the symbols of the right-hand sides; the work performed is the same.

$$E \longrightarrow T \ \{ \ TT.st := T.ptr \ \} \ TT \ \{ \ E.ptr := TT.ptr \ \}$$

$$TT_1 \longrightarrow + \ T \ \{ \ TT_2.st := make\_bin\_op("+", TT_1.st, T.ptr) \ \} \ TT_2 \ \{ \ TT_1.ptr := TT_2.ptr \ \}$$

$$TT_1 \longrightarrow - \ T \ \{ \ TT_2.st := make\_bin\_op("-", TT_1.st, T.ptr) \ \} \ TT_2 \ \{ \ TT_1.ptr := TT_2.ptr \ \}$$

$$TT \longrightarrow \epsilon \ \{ \ TT.ptr := TT.st \ \}$$

$$T \longrightarrow F \ \{ \ FT.st := F.ptr \ \} \ FT \ \{ \ T.ptr := FT.ptr \ \}$$

$$FT_1 \longrightarrow * \ F \ \{ \ FT_2.st := make\_bin\_op("\times", FT_1.st, F.ptr) \ \} \ FT_2 \ \{ \ FT_1.ptr := FT_2.ptr \ \}$$

$$FT_1 \longrightarrow / \ F \ \{ \ FT_2.st := make\_bin\_op("\div", FT_1.st, F.ptr) \ \} \ FT_2 \ \{ \ FT_1.ptr := FT_2.ptr \ \}$$

$$FT \longrightarrow \epsilon \ \{ \ FT.ptr := FT.st \ \}$$

$$F_1 \longrightarrow - \ F_2 \ \{ \ F_1.ptr := make\_un\_op("+/-", F_2.ptr) \ \}$$

$$F \longrightarrow (\ E \ ) \ \{ \ F.ptr := E.ptr \ \}$$

$$F \longrightarrow const \ \{ \ F.ptr := make\_leaf(const.ptr) \ \}$$

Figure 4.9 LL(1) grammar with action routines to build a syntax tree.

- The ease with which the attribute grammar can be transformed into the grammar with action routines is due to the fact that the attribute grammar is **L-attributed**. If it required more complicated flow, we would not be able to cast it in the form of action routines.

Recursive Descent and Action Routines

- As in ordinary parsing, there is a strong analogy between recursive descent and table-driven parsing with action routines. Figure 4.10 shows the **term_tail** routine from Figure 2.16 (page 74), modified to do its part in constructing a syntax tree.

- The behavior of this routine mirrors that of productions 2 through 5 in Figure 4.9. The routine accepts as a parameter a pointer to the syntax tree fragment contained in the attribute grammar's **TT1** .

- Then, given an upcoming **+** or **-** symbol on the input, it
    1. calls add op to parse that symbol (returning a character string representation);
    2. calls term to parse the attribute grammar's **T**;
    3. calls make bin op to create a new tree node;
    4. passes that node to **term_tail**, which parses the attribute grammar's **TT2** ; and
    5. returns the result.

$E \longrightarrow T \ \{ \ TT.\text{st} := T.\text{ptr} \ \} \ TT \ \{ \ E.\text{ptr} := TT.\text{ptr} \ \}$

$TT_1 \longrightarrow + T \ \{ \ TT_2.\text{st} := \text{make\_bin\_op}("+", TT_1.\text{st}, T.\text{ptr}) \ \} \ TT_2 \ \{ \ TT_1.\text{ptr} := TT_2.\text{ptr} \ \}$

$TT_1 \longrightarrow - T \ \{ \ TT_2.\text{st} := \text{make\_bin\_op}("-", TT_1.\text{st}, T.\text{ptr}) \ \} \ TT_2 \ \{ \ TT_1.\text{ptr} := TT_2.\text{ptr} \ \}$

$TT \longrightarrow \epsilon \ \{ \ TT.\text{ptr} := TT.\text{st} \ \}$

$T \longrightarrow F \ \{ \ FT.\text{st} := F.\text{ptr} \ \} \ FT \ \{ \ T.\text{ptr} := FT.\text{ptr} \ \}$

$FT_1 \longrightarrow * F \ \{ \ FT_2.\text{st} := \text{make\_bin\_op}("\times", FT_1.\text{st}, F.\text{ptr}) \ \} \ FT_2 \ \{ \ FT_1.\text{ptr} := FT_2.\text{ptr} \ \}$

$FT_1 \longrightarrow / F \ \{ \ FT_2.\text{st} := \text{make\_bin\_op}("\div", FT_1.\text{st}, F.\text{ptr}) \ \} \ FT_2 \ \{ \ FT_1.\text{ptr} := FT_2.\text{ptr} \ \}$

$FT \longrightarrow \epsilon \ \{ \ FT.\text{ptr} := FT.\text{st} \ \}$

$F_1 \longrightarrow - F_2 \ \{ \ F_1.\text{ptr} := \text{make\_un\_op}("+/\_", F_2.\text{ptr}) \ \}$

$F \longrightarrow (E) \ \{ \ F.\text{ptr} := E.\text{ptr} \ \}$

$F \longrightarrow \textbf{const} \ \{ \ F.\text{ptr} := \text{make\_leaf}(\text{const}.\text{ptr}) \ \}$

Figure 4.9 LL(1) grammar with action routines to build a syntax tree.

```
procedure term_tail(lhs : tree_node_ptr)
    case input_token of
        +, - :
            op : string := add_op
            return term_tail(make_bin_op(op, lhs, term))
                -- term is a recursive call with no arguments
        ), id, read, write, $$ :       -- epsilon production
            return lhs
        otherwise parse_error
```

Figure 4.10 Recursive descent parsing with embedded "action routines." Compare to the routine with the same name in Figure 2.16 (page 74) and with productions 2 through 5 in Figure 4.9.

Bottom-Up Evaluation

- In an **LR parser generator**, one cannot in general embed action routines at arbitrary places in a right-hand side, since the parser does not in general know what production it is in until it has seen all or most of the yield.

- **LR parser generators** therefore permit action routines only after the point at which the production being parsed can be identified unambiguously (this is known as the trailing part of the right-hand side; the ambiguous part is the left corner).

Bottom-Up Evaluation

- If the attribute flow of the action routines is strictly bottom-up (as it is in an **S-attributed** attribute grammar), then execution at the end of right-hand sides is all that is needed. The **attribute grammars** of Figures 4.1 and 4.5, in fact, are essentially identical to the action routine versions.

- If the action routines are responsible for a significant part of semantic analysis, however (as opposed to simply building a syntax tree), then they will often need contextual information in order to do their job. To obtain and use this information in an **LR** parse, they will need some (necessarily limited) access to inherited attributes or to information outside the current production.

Space Management for Attributes

- Any attribute evaluation method requires space to hold the attributes of the grammar symbols.

- The details differ in bottom-up and top-down parsers.

- For a bottom-up parser with an **S-attributed** grammar, the obvious approach is to maintain an attribute stack that directly mirrors the parse stack: next to every state number on the parse stack is an attribute record for the symbol we shifted when we entered that state. Entries in the attribute stack are pushed and popped automatically by the parser driver; space management is not an issue for the writer of action routines.

- For a top-down parser with an **L-attributed** grammar, we have two principal options. The first option is automatic, but more complex than for bottom-up grammars. It still uses an attribute stack, but one that does not mirror the parse stack. The second option has lower space overhead, and saves time by "shortcutting" copy rules, but requires action routines to allocate and deallocate space for attributes explicitly.

Syntax Tree Decoration I
Language

Bottom-up CFG for Calculator Language with Types

Figure 4.11 contains a bottom-up **CFG** for a calculator language with types and declarations. The grammar differs from that of Example 2.37 in three ways:

1. we allow declarations to be intermixed with statements,

2. we differentiate between integer and real constants (presumably the latter contain a decimal point), and

3. we require explicit conversions between integer and real operands.

The intended semantics of our language requires that every identifier be declared before it is used, and that types not be mixed in computations.

$program \longrightarrow stmt\_list \ \$\$$

$stmt\_list \longrightarrow stmt\_list \ decl \mid stmt\_list \ stmt \mid \epsilon$

$decl \longrightarrow$ `int id` $\mid$ `real id`

$stmt \longrightarrow$ `id :=` $expr \mid$ `read id` $\mid$ `write` $expr$

$expr \longrightarrow term \mid expr \ add\_op \ term$

$term \longrightarrow factor \mid term \ mult\_op \ factor$

$factor \longrightarrow$ `(` $expr$ `)` $\mid$ `id` $\mid$ `int_const` $\mid$ `real_const` $\mid$
        `float (` $expr$ `)` $\mid$ `trunc (` $expr$ `)`

$add\_op \longrightarrow$ `+` $\mid$ `-`

$mult\_op \longrightarrow$ `*` $\mid$ `/`

Figure 4.11 Context-free grammar for a calculator language with types and declarations. The intent is that every identifier be declared before use, and that types not be mixed in computations.

Syntax Tree to Average an Integer and a Real

- Extrapolating from the example in Figure 4.5, it is easy to add semantic functions or action routines to the grammar of Figure 4.11 to construct a syntax tree for the calculator language.

- The obvious structure for such a tree would represent expressions as we did in Figure 4.7, and would represent a program as a linked list of declarations and statements.

- As a concrete example, Figure 4.12 contains the syntax tree for a simple program to print the average of an integer and a real.

int a
read a
real b
read b
write (float (a) + b) / 2.0
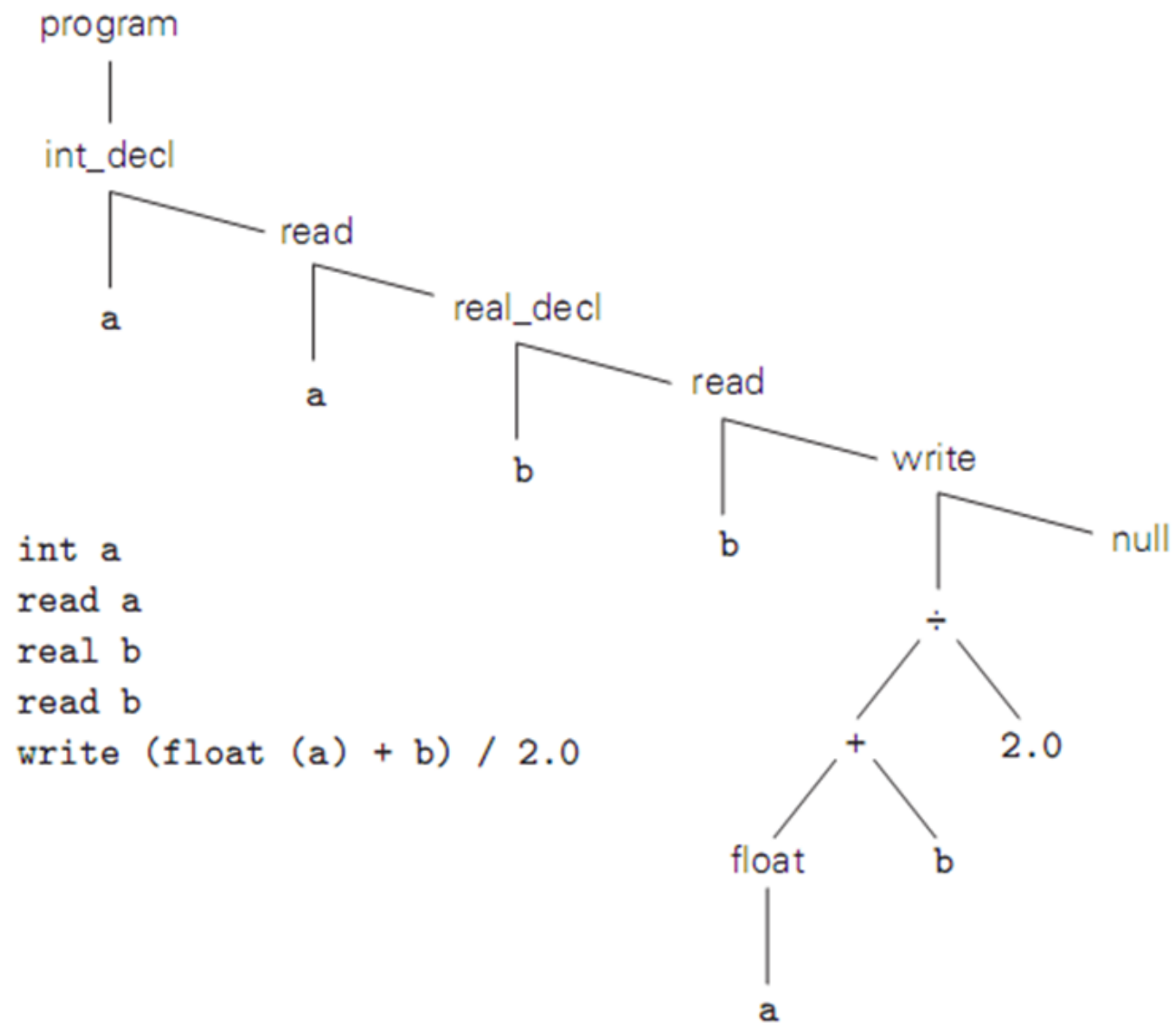
Figure 4.12   Syntax tree for a simple calculator program.

Note

This example is used as the example to conclude our discussion in Unit 1: Foundations. (Chapter 1 – Chapter 4)

This Unit 1, basically, introduces the construction of the compiler front-end of this **Calculator Language** project.

The following lecture will have its complete decorated AG (Attributed AST) design for this Calculator Language.

Syntax Tree Decoration II
Example

Tree Grammar for the Calculator Language with Types

- Much as a context-free grammar describes the possible structure of parse trees for a given programming language, we can use a tree grammar to represent the possible structure of syntax trees.

- As in a **CFG**, each production of a tree grammar represents a possible relationship between a parent and its children in the tree.

- The parent is the symbol on the left-hand side of the production; the children are the symbols on the right-hand side. The productions used in Figure 4.12 might look something like the following.

$$program \longrightarrow item$$
$$int\_decl : item \longrightarrow id\ item$$
$$read : item \longrightarrow id\ item$$
$$real\_decl : item \longrightarrow id\ item$$
$$write : item \longrightarrow expr\ item$$
$$null : item \longrightarrow \epsilon$$
$$\text{'}\div\text{'} : expr \longrightarrow expr\ expr$$
$$\text{'}+\text{'} : expr \longrightarrow expr\ expr$$
$$float : expr \longrightarrow expr$$
$$id : expr \longrightarrow \epsilon$$
$$real\_const : expr \longrightarrow \epsilon$$

Tree Grammar for the Calculator Language with Types

- The notation **A** : **B** on the left-hand side of a production means that
  1. **A** is one variant of **B**, and
  2. may appear anywhere a **B** is expected on a right-hand side.

- **Tree grammars** and **context-free grammars** differ in important ways.
  - A **context free grammar** is meant to **generate** a language composed of strings of tokens, where each string is the **yield** of a parse tree.
  - Parsing is the process of finding a tree that has a given yield. A tree grammar, as we use it here, is meant to generate the trees themselves. We have no need for a notion of parsing: we can easily inspect a tree and determine whether and how it can be generated by the grammar.

Tree Grammar for the Calculator Language with Types

- Our purpose in introducing tree grammars is to provide a framework for the decoration of syntax trees. Semantic rules attached to the productions of a tree grammar can be used to define the attribute flow of a syntax tree in exactly the same way that semantic rules attached to the productions of a context-free grammar are used to define the attribute flow of a parse tree.

- We will use a tree grammar in the remainder of this section to perform static semantic checking.

Definition of the Terms in Figures

Each Node keep track of the following data:
name: (token name)
type: (token type)
symtab: reference to the symbol table.
errors_in, errors_out, errors: error message for error propagation.

Inherited means a attribute from parent or left siblings.
Synthesized means from lower level of the tree. (By bottom-up parsing)

Attributes

Inherited	Synthesized
—	location, errors
symtab, errors_in	location, errors_out
symtab	location, type, errors, name (*id* only)

Shorthand symbol in the Figures:

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

location attribute not shown

Tree AG for the Calculator Language with Types

A complete tree attribute grammar for our calculator language with types can be constructed using the **node classes**, **variants**, and **attributes** shown in Figure 4.13.

The grammar itself appears in Figure 4.14. Once decorated, the program node at the root of the syntax tree will contain a list, in a **synthesized attribute**, of all static semantic errors in the program.

Each **item** or **expr** node has an inherited attribute **symtab** that contains a list, with types, of all identifiers declared to the left in the tree. Each **item** node also has an **inherited attribute errors** in that lists all static semantic errors found to its left in the tree, and a synthesized attribute errors out to propagate the final error list back to the root. Each expr node has one synthesized attribute that indicates its type and another that contains a list of any static semantic errors found inside.

Tree AG for the Calculator Language with Types

- Our handling of **semantic errors** illustrates a common technique:
  - In order to continue looking for other errors we must provide values for any attributes that would have been set in the absence of an error.

- To avoid cascading error messages, we choose values for those attributes that will pass quietly through subsequent checks.

- In this specific case we employ a pseudo type called **error**, which we associate with any symbol table entry or expression for which we have already generated a message.

Tree AG for the Calculator Language with Types

- Though it takes a bit of checking to verify the fact, our attribute grammar is noncircular and well defined. No attribute is ever assigned a value more than once.

- **Note:** The helper routines at the end of Figure 4.14 should be thought of as macros, rather than semantic functions. For the sake of brevity we have passed them entire tree nodes as arguments. Each macro calculates the values of two different attributes. Under a strict formulation of attribute grammars each macro would be replaced by two separate semantic functions, one per calculated attribute.

Class of node	Variants	Attributes Inherited	Synthesized
program	—	—	location, errors
item	*int_decl, real_decl, read, write, :=, null*	symtab, errors_in	location, errors_out
expr	*int_const, real_const, id, +, −, ×, ÷, float, trunc*	symtab	location, type, errors, name (*id* only)

Figure 4.13 Classes of nodes for the syntax tree attribute grammar of Figure 4.14. With the exception of name, all variants of a given class have all the class's attributes.

$program \longrightarrow item$
▷ item.symtab := null
▷ program.errors := item.errors_out
▷ item.errors_in := null

$int\_decl : item_1 \longrightarrow id\ item_2$
▷ declare_name(id, $item_1$, $item_2$, int)
▷ $item_1$.errors_out := $item_2$.errors_out

$real\_decl : item_1 \longrightarrow id\ item_2$
▷ declare_name(id, $item_1$, $item_2$, real)
▷ $item_1$.errors_out := $item_2$.errors_out

$read : item_1 \longrightarrow id\ item_2$
▷ $item_2$.symtab := $item_1$.symtab
▷ if (id.name, ?) ∈ $item_1$.symtab
    $item_2$.errors_in := $item_1$.errors_in
  else
    $item_2$.errors_in := $item_1$.errors_in + [id.name "undefined at" id.location]
▷ $item_1$.errors_out := $item_2$.errors_out

$write : item_1 \longrightarrow expr\ item_2$
▷ expr.symtab := $item_1$.symtab
▷ $item_2$.symtab := $item_1$.symtab
▷ $item_2$.errors_in := $item_1$.errors_in + expr.errors
▷ $item_1$.errors_out := $item_2$.errors_out

$':=' : item_1 \longrightarrow id\ expr\ item_2$
▷ expr.symtab := $item_1$.symtab
▷ $item_2$.symtab := $item_1$.symtab
▷ if (id.name, A) ∈ $item_1$.symtab   -- for some type A
    if A ≠ error and expr.type ≠ error and A ≠ expr.type
        $item_2$.errors_in := $item_1$.errors_in + ["type clash at" $item_1$.location]
    else
        $item_2$.errors_in := $item_1$.errors_in + expr.errors
  else
    $item_2$.errors_in := $item_1$.errors_in + [id.name "undefined at" id.location] + expr.errors
▷ $item_1$.errors_out := $item_2$.errors_out

$null : item \longrightarrow \epsilon$
▷ item.errors_out := item.errors_in

Figure 4.14 Attribute grammar to decorate an abstract syntax tree for the calculator language with types. We use square brackets to delimit error messages and pointed brackets to delimit symbol table entries. Juxtaposition indicates concatenation within error messages; the '+' and '−' operators indicate insertion and removal in lists. We assume that every node has been initialized by the scanner or by action routines in the parser to contain an indication of the location (line and column) at which the corresponding construct appears in the source (see Exercise 4.22). The '?' symbol is used as a "wild card"; it matches any type. *(continued)*

$id : expr \longrightarrow \epsilon$
▷ if ⟨id.name, A⟩ ∈ expr.symtab   −− for some type A
    expr.errors := null
    expr.type := A
▷ else
    expr.errors := [id.name "undefined at" id.location]
    expr.type := error

$int\_const : expr \longrightarrow \epsilon$
▷ expr.type := int

$real\_const : expr \longrightarrow \epsilon$
▷ expr.type := real

$`+' : expr_1 \longrightarrow expr_2\ expr_3$
▷ $expr_2$.symtab := $expr_1$.symtab
▷ $expr_3$.symtab := $expr_1$.symtab
▷ check_types($expr_1$, $expr_2$, $expr_3$)

$`-' : expr_1 \longrightarrow expr_2\ expr_3$
▷ $expr_2$.symtab := $expr_1$.symtab
▷ $expr_3$.symtab := $expr_1$.symtab
▷ check_types($expr_1$, $expr_2$, $expr_3$)

$`\times' : expr_1 \longrightarrow expr_2\ expr_3$
▷ $expr_2$.symtab := $expr_1$.symtab
▷ $expr_3$.symtab := $expr_1$.symtab
▷ check_types($expr_1$, $expr_2$, $expr_3$)

$`\div' : expr_1 \longrightarrow expr_2\ expr_3$
▷ $expr_2$.symtab := $expr_1$.symtab
▷ $expr_3$.symtab := $expr_1$.symtab
▷ check_types($expr_1$, $expr_2$, $expr_3$)

$float : expr_1 \longrightarrow expr_2$
▷ $expr_2$.symtab := $expr_1$.symtab
▷ convert_type($expr_2$, $expr_1$, int, real, "float of non-int")

$trunc : expr_1 \longrightarrow expr_2$
▷ $expr_2$.symtab := $expr_1$.symtab
▷ convert_type($expr_2$, $expr_1$, real, int, "trunc of non-real")

Figure 4.14: (Continued)

```
macro declare_name(id, cur_item, next_item : syntax_tree_node; t : type)
    if ⟨id.name, ?⟩ ∈ cur_item.symtab
        next_item.errors_in := cur_item.errors_in + ["redefinition of" id.name "at" cur_item.location]
        next_item.symtab := cur_item.symtab − ⟨id.name, ?⟩ + ⟨id.name, error⟩
    else
        next_item.errors_in := cur_item.errors_in
        next_item.symtab := cur_item.symtab + ⟨id.name, t⟩

macro check_types(result, operand1, operand2)
    if operand1.type = error or operand2.type = error
        result.type := error
        result.errors := operand1.errors + operand2.errors
    else if operand1.type ≠ operand2.type
        result.type := error
        result.errors := operand1.errors + operand2.errors + ["type clash at" result.location]
    else
        result.type := operand1.type
        result.errors := operand1.errors + operand2.errors

macro convert_type(old_expr, new_expr : syntax_tree_node; from_t, to_t : type; msg : string)
    if old_expr.type = from_t or old_expr.type = error
        new_expr.errors := old_expr.errors
        new_expr.type := to_t
    else
        new_expr.errors := old_expr.errors + [msg "at" old_expr.location]
        new_expr.type := error
```

Figure 4.14: (Continued)

Decorating a Tree with the AG of The AG Tree Example

- This Calculator Language Project:
  1. Decorated AG: Figure 4.15.
  2. Decorate CFG Grammar: Figure 4.14
  3. Syntax Tree: Figure 4.12.

The pattern of **attribute flow** appears considerably messier than in previous examples in this chapter, but this is simply because type checking is more complicated than calculating constants or building a syntax tree.

Decorating a Tree with the AG of The AG Tree Example

- Symbol table information flows along the chain of **item**s and down into **expr** trees. The **int_decl** and **real_decl** nodes add new information; other nodes simply pass the table along. Type information is synthesized at **id : expr** leaves by looking up an identifier's name in the symbol table.

- The information then propagates upward within an expression tree, and is used to type-check operators and assignments.

- Error messages flow along the chain of items via the errors in attributes, and then back to the root via the errors out attributes. Messages also flow up out of **expr** trees. Wherever a type check is performed, the type attribute may be used to help create a new message to be appended to the growing message list.

Decorating a Tree with the AG of The AG Tree Example

- In our example grammar we accumulate error messages into a synthesized attribute of the root of the syntax tree. In an **ad hoc attribute evaluator** we might print these messages on the fly as the errors are discovered. In practice, particularly in a multi-pass compiler, it makes sense to buffer the messages, so they can be interleaved with messages produced by other phases of the compiler, and printed in program order at the end of compilation. [The error message could only be a code on the tree node. e.g. Error 17]

- One could convert our attribute grammar into executable code using an **automatic attribute evaluator generator.** Alternatively, one could create an ad hoc evaluator in the form of mutually recursive subroutines. In the latter case attribute flow would be explicit in the **calling sequence of the routines**.

- We could then choose if desired to keep the symbol table in global variables, rather than passing it from node to node through attributes. Most compilers employ the **ad hoc** approach.
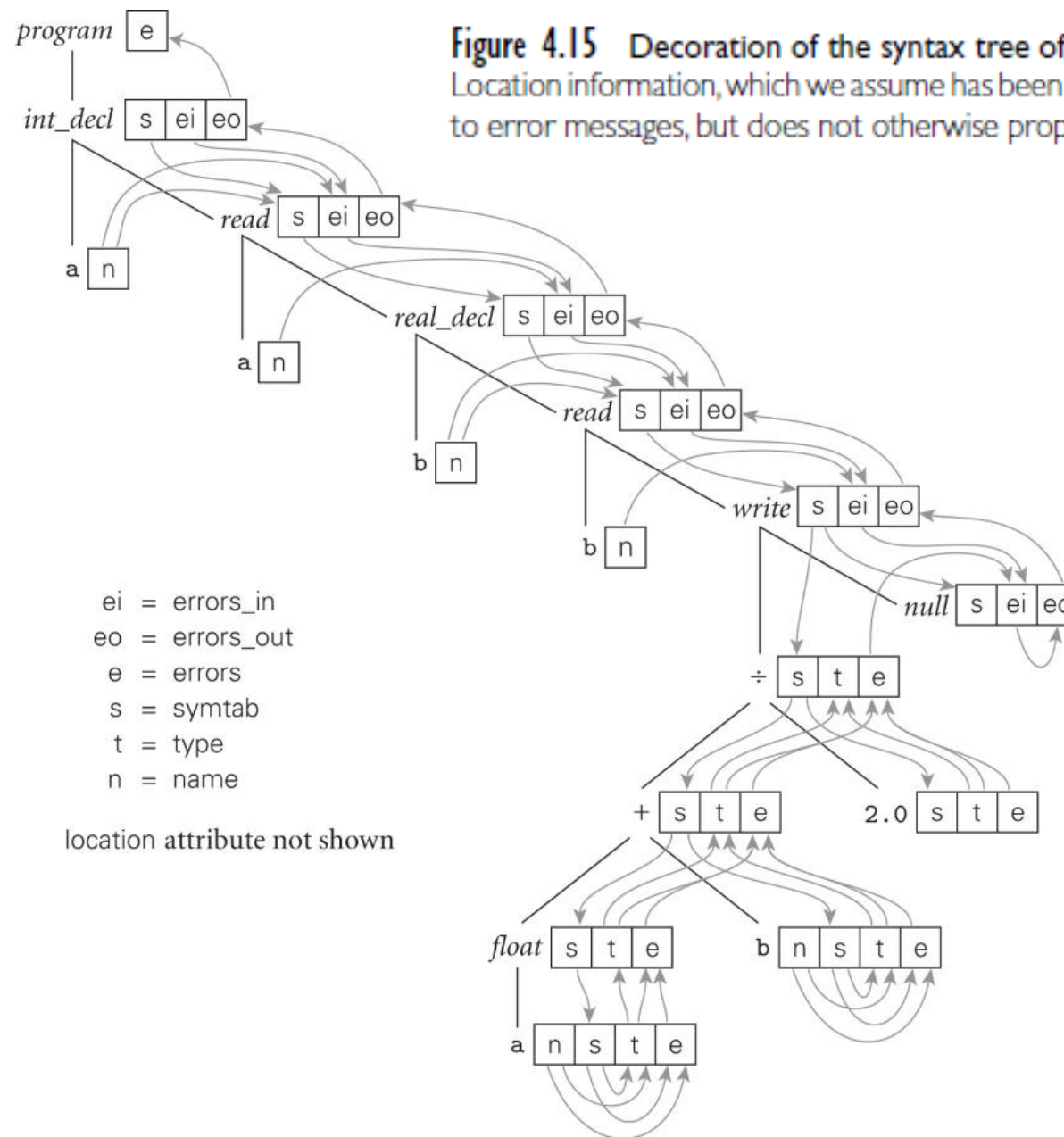
Figure 4.15 Decoration of the syntax tree of Figure 4.12, using the grammar of Figure 4.14. Location information, which we assume has been initialized in every node by the parser, contributes to error messages, but does not otherwise propagate through the tree.

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

location attribute not shown

Building Recursive Descent Parsers:

The Definitive Guide

Building Recursive Descent Parsers

- Parsers are programs that help process text. Compilers and interpreters use parsers to analyze programs before processing them further, and parsers for formats like JSON and YAML process text into the programming language's native data structures.

- In this article, we're going to look at how to build "recursive descent parsers". Recursive descent parsers are a simple but powerful way of building parsers — for each "entity" in the text that you want to process, you define a function.

Building Recursive Descent Parsers

- First, we are going to look at some of the theory underlying parsing. Then, we using Python, we're going to build a calculator and a JSON parser that supports comments, trailing commas and unquoted strings. Finally, we're going to discuss how you can build parsers for other kinds of languages that behave differently than the aforementioned examples.

- If you're already familiar with the theory, you can directly skip to the parts where we build the parser.

The Definitive Guide

How does parsing work?

How does parsing work?

- In order to process a piece of text, a program performs three tasks. In a process called "lexing" or "tokenization", the program goes over characters in the text, and extracts logical groups called "**tokens**".

- For example, in an expression such as "3 + 5 – 10", a program to process arithmetic expressions could extract the following tokens:

Token Stream (Type, ID, Value)

3 + 5 - 10

```
[{ "Type": "Number" , "Text": "3" },
{ "Type": "Operator", "Text": "+" },
{ "Type": "Number" , "Text": "5" },
{ "Type": "Operator", "Text": "-" },
{ "Type": "Number" , "Text": "10" }]
```

How does parsing work?

- The program then uses rules to identify meaningful sequences of tokens, and this is called "parsing". The rules used to make this happen are called "grammars" — much in the same way that words of a natural language like English are strung together into meaningful sequences using English grammar. Individual rules of a grammar are called "productions".

How does parsing work?

- Imagine we're building a program to process math expressions, and we're interested only in addition and subtraction. An expression must have a number (such as "3"), followed by any number of operators and numbers (such as "+ 5"). The parsing rule can be visualized like so:

```
Expression ➔ Number (Operator  Number)*
```

- The * indicates that the (Operator Number)* group may repeat any number of times (including zero).

How does parsing work?

- Finally, the program takes a set of actions for a grammar rule. For example, after parsing, our math program would need to calculate the value of the expression.

- Although we've described these as separate steps, a program can perform all of these at once. Doing all the steps at once carries a few advantages, and this is the approach we're going to take in this article.

The Definitive Guide

Writing production rules

Writing production rules

- Through the rest of this article, we're going to use production rules to visualize grammars. Therefore, we've explained how we write the production rules throughout this article. If you've previously gone through a textbook on parsing, the notation we've used here is a bit different.

- Earlier, we've seen an example of a simple production. When the grammar rule contains a token's name, and the token is fairly simple, it's common to write the token's text in the grammar. If we consider the previous example, Operator yields a + or a -, so we could rewrite the rule like so:

```
Expression  Number ( ("+" | "-" )  Number )*
```

Writing production rules

- We've also seen how we can use * to indicate that something repeats any number of times. Similarly, a + indicates that something repeats, but it should occur at least once. A ? indicates something that's optional.

- As an example, suppose we're building a parser to process a list of conditions, like "x > 10 and y > 30". Assume that the and is optional, so we could rewrite this condition as "x > 10 y > 30". You could design a grammar for the above like so:

```
Expression → Condition ( "and" ?  Condition )*

Condition → Variable  Operator  Number
```

Writing production rules

- When there are multiple alternatives, the order of alternatives matter. For a rule like `Operator → "+" | "-",` we should try matching a "+" first, and then a "-". Although in this trivial example it might seem that order does not matter, we're going to see examples later on where order becomes important.

- Finally, a production rule only focuses on tokens and other grammar symbols — it ignores whitespaces and comments.

The Definitive Guide
Considerations when building parsers

SECTION 11

Considerations when building parsers

- Previously, we've seen a few simple grammars. However, when trying to parse something more complex, a number of issues can come up. We're going to discuss them here.

Handling precedence in grammars

- Previously, we've seen a grammar that can handle a math expression consisting of additions and subtractions. Unfortunately, you can't extend the same grammar for multiplications (and divisions), since those operations have higher precedence than additions (and subtractions).

- In order to handle this scenario, we must design our grammar so that the parser defers any additions, but performs multiplications as soon as it is encountered. To achieve this, we split the grammar into two separate rules, as shown below:

```
Expression → Term ( ( "+" | "-" ) Term )*

Term → Number ( ( "*" | "/" ) Number )*
```

Handling precedence in grammars

- Let us take an expression (such as "3 + 2 * 5") to ensure that this really works. We'll assume that once our parser has finished matching a grammar rule, it automatically calculates the expression. We'll also use underlined text to indicate what the parser is looking for. In addition, we've also omitted the quotes for clarity.

```
Expression → Term  +  Term → 3  + Term →
3 + Number * Number → 3 + 2 * Number →
3 + 2 * 5
```

Handling precedence in grammars

- When 2 * 5 is parsed, the parser immediately returns the result of 10. This result is received at the Term "+" Term step, and the addition is performed at the end, yielding 13 as the result.

- In summary, you should arrange your rules so that operators with the lowest precedence are at the top, while the ones with highest precedence are at the bottom.

Avoiding left recursion

- Earlier, we have mentioned that a recursive descent parser consists of functions that process "entities" in the text. Now that we know about tokens and production rules, let us redefine this a bit more formally: each function in the parser extracts a token, or implements the logic of a production rule.

Avoiding left recursion

- Let us take a simple grammar, such as Expression \rightarrow Number "+" Number to see what this really means. If you wrote the pseudocode for this grammar, it would look something like this:

```python
def expression():
    first_number = number()
    read('+')
    second_number = number()
    # process these two numbers, e.g. adding them
```

Avoiding left recursion

- Next, consider a grammar that parses a list of comma separated numbers. You'd generally write it as:

```
List → Number ("," Number)*
```

- Now, imagine that for some reason, you wanted to avoid the * wildcard and rewrite it like so:

```
List → List "," Number | Number
```

- This grammar is theoretically correct — it either picks a single number from the text, or it expands to a list with a number at the end. The second list, in turn, can expand to another number or a list, until parser finishes through the text.

Avoiding left recursion

- However, there's a problem with this approach — you'd enter an infinite recursion! If you try calling the list() function once, you end up calling it again, and so on. You might think that rewriting the grammar as Number | List "," Number might help. However, if you tried writing a function, you'd read a single number and quit parsing. The number you read might be part of a list like "2, 4, 6" and you won't be able to read the other numbers.

- When dealing with parsing, this is called **left recursion**. The case that we saw above involves "direct left recursion", but there's also "indirect left recursion", where A calls B, B calls A and so on. In either case, you should rewrite the grammar in another way to avoid this issue.

Avoiding backtracking

- Suppose, you're trying to build a parser that parses an operation of two numbers, such as "2 + 4", and wrote a grammar in this way:

```
Expression → Addition | Subtraction
Addition → Number "+" Number
Subtraction → Number "-" Number
```

- This grammar is correct, and it will also behave in the way you expect and produce correct results. However, this grammar is not what you'd want to use.

Avoiding backtracking

- To see why this is the case, consider the input string "5 – 2". We'll first go with the addition rule, and parse a number. Then, we'd expect a "+" but when reading the string, we'll get a "-" which means we have to backtrack and apply the subtraction rule. In doing so, we've wasted time and CPU cycles on parsing a number, only to discard it and parse it again as part of the subtraction rule.

Avoiding backtracking

- Additionally, when building parsers that directly work off of data streams, like a network socket, rewinding the stream is often not an option. In this article, we'll only deal with parsers that have all of the text in memory. Nevertheless, it's a good practice to avoid backtracking and to do so, you should factor out the common parts from the left. This is how our rule would look after factoring:

```
Expression → Number ( "+"  Number | "-"  Number )
```

Avoiding backtracking

- The factoring step is complete and will avoid backtracking. However, for aesthetic reasons, we'll just write it as:

> Expression → Number ("+" | "-") Number

The Definitive Guide

Building a base for the Recursive Descent Parser in Python

Building a base for the Recursive Descent Parser in Python

- Now that we've discussed the various considerations for parsing, we'll build out the calculator and JSON parser. However, before we do that, we'll write a base "Parser" class that has some methods for common functions, such as recognizing characters and handling errors.

- This section might seem a bit too long, but it's well worth the patience. Once you complete this section, you'll have all the tooling to build complex parsers with very little code required to recognize tokens and implement production rules.

The exception class

- Since this is the easiest task by far, so we'll tackle this first. We'll define our own exception class named **ParseError** like so:

```python
class ParseError(Exception):
    def __init__(self, pos, msg, *args):
        self.pos = pos
        self.msg = msg
        self.args = args
    def __str__(self):
        return '%s at position %s' % (self.msg % self.args, self.pos)
```

The exception class

- The class accepts the text position where the error occurred, an error message (as a format string), and the arguments to the format string. Let's see of how you might use this exception type:

```
e = ParseError(13, 'Expected "{" but found "%s"', "[")
```

- When you try printing the exception, you'll get a formatted message like this:

```
Expected "{" but found "[" at position 13
```

The exception class

•Notice that we haven't used a % symbol within the format string and its arguments (such as '"Expected "{" but found "%s"' % "["). This is handled in the __str__ method of the class, where we've formatted self.msg with elements in self.pos.

•Now, you might ask, why do we want to format it in the __str__ method, instead of writing it with a %? It turns out that using the % operator to format a string takes up quite a bit of time. When parsing a string, there'll be many exceptions raised as the parser tries to apply various rules. Hence, we've deferred the string formatting to when it's really needed.

The base Parser class

- Now that we have an error class in place, the next step is to write the parser class. We'll define it as follows:

```python
class Parser:
    def __init__(self):
        self.cache = {}
    def parse(self, text):
        self.text = text
        self.pos = -1
        self.len = len(text) - 1
        rv = self.start()
        self.assert_end()
        return rv
```

The base Parser class

- Here, you'll notice a cache member in the __init__ method — we'll get back to why this is needed when we discuss recognizing characters.

- The parse method is fairly simple — first, it stores the string. Since we haven't scanned any part of the string yet, we'll set the position to -1. Also, we'll store the maximum index of the string in self.len. (The maximum index is always one less than the length of the string.)

- The assert_end method is simple: we'll check if the current position is less than the maximum index. Bear in mind that these methods are inside the class, although we've omitted the indentation for simplicity.

The base Parser class

```python
def assert_end(self):
    if self.pos < self.len:
        raise ParseError( self.pos + 1,    \
                          'Expected end of string but got %s',    \
                          self.text[self.pos + 1] )
```

The base Parser class

- Throughout our parser, we'll be looking at the character that is one more than the current index (self.pos). To see why this is the case, consider that we start out with self.pos = -1, so we'd be looking at the index 0. Once we've recognized a character successfully, self.pos goes to 0 and we'd look at the character at index 1, and so on. This is why the error uses self.pos + 1.

The base Parser class

- With most languages, whitespace between tokens can be safely discarded. Therefore, it seems logical to include a method that checks if the next character is a whitespace character, and if so, "discards" it by advancing to the next position.

```python
def eat_whitespace(self):
    while self.pos < self.len and    \
          self.text[self.pos + 1] in " \f\v\r\t\n":
        self.pos += 1
```

Processing character ranges

- Now, we'll write a function to help us recognize characters in the text. Before we do that though, we'll consider our function from a user's perspective — we'll give it a set of characters to match from the text. For example, if you wanted to recognize an alphabet or underscore, you might write something like:

```
self.char('A-Za-z_')
```

- We'll give the char method containing a list of characters or ranges (such as A-Z in the above example). Ranges are denoted through two characters separated by a -. The first character has a numerically smaller value than the one on the right.

Processing character ranges

- We'll give the char method containing a list of characters or ranges (such as A-Z in the above example). Ranges are denoted through two characters separated by a -. The first character has a numerically smaller value than the one on the right.

- Now, if the character at self.text[self.pos + 1] matches something in the argument of self.char, we'll return it. Otherwise, we'll raise an exception.

- Internally, we'll process the string into something that's easier to handle — a list with separate characters or ranges such as ['A-Z', 'a-z', '_']. So, we'll write a function to split the ranges:

```python
def split_char_ranges(self, chars):
    try:
        return self.cache[chars]
    except KeyError:
        pass
    rv = []
    index = 0
    length = len(chars)
    while index < length:
        if index + 2 < length and chars[index + 1] == '-':
            if chars[index] >= chars[index + 2]:
                raise ValueError('Bad character range')
            rv.append(chars[index:index + 3])
            index += 3
        else:
            rv.append(chars[index])
            index += 1
    self.cache[chars] = rv
    return rv
```

Processing character ranges

- Here, we loop through the chars string, looking for a - followed by another character. If this condition matches and the first character is smaller than the last, we add the entire range (such as A-Z) to the list rv. Otherwise, we add the character at chars[index] to rv.

- Then, we add the list in rv it to the cache. If we see this string a second time, we'll return it from the cache using the try … except KeyError: … block at the top.

- Admittedly, we could have just provided a list like ['A-Z', 'a-z', '_'] to the char method. However, in our opinion, doing it in this way makes calls to char() look a bit cleaner.

Extracting characters from the text

- Now that we have a method that gives a list containing ranges and characters, we can write our function to extract a character from the text. Here's the code for the char method:

```python
def char(self, chars=None):
    if self.pos >= self.len:
        raise ParseError( self.pos + 1,
                          'Expected %s but got end of string',
                          'character' if chars is None else '[%s]' % chars
                        )
    next_char = self.text[self.pos + 1]
    if chars == None:
        self.pos += 1
        return next_char
    for char_range in self.split_char_ranges(chars):
        if len(char_range) == 1:
            if next_char == char_range:
                self.pos += 1
                return next_char
        elif char_range[0] <= next_char <= char_range[2]:
            self.pos += 1
            return next_char
    raise ParseError( self.pos + 1,
                      'Expected %s but got %s',
                      'character' if chars is None else '[%s]' % chars, next_char
                    )
```

Extracting characters from the text

- Let us first focus on the argument chars=None. This allows you to call self.char() without giving it a set of characters. This is useful when you want to simply extract a character without restricting it to a specific set. Otherwise, you can call it with a range of characters, as we've seen in the previous section.

- This function is fairly straightforward — first, we check if we've run out of text. Then, we pick the next character and check if chars is None, in which case we can increment the position and return the character immediately.

Extracting characters from the text

- However, if there are a set of characters in chars, we split it into a list of individual characters and ranges, such as ['A-Z', 'a-z', '_']. In this list, a string of length 1 is a character, and anything else is a range. It checks if the next character matches the character or the range, and if it does, we return it. If we failed to match it against anything, it exits the loop which raises ParseError, stating that we couldn't get the expected character.

- The 'character' if chars is None else '[%s]' % chars is just a way of providing a more readable exception. If chars is None, the exception's message would read Expected character but got ..., but if char was set to something like A-Za-z_, we'd get Expected [A-Za-z_] but got ....

- Later on, we'll see how to use this function to recognize tokens.

Extracting keywords and symbols

- Apart from extracting individual characters, recognizing keywords is a common task when building a parser. We're using "keywords" to loosely refer to any contiguous string that is it's "own entity", and may have whitespace before and after it. For example, in JSON { could be a keyword, and in a programming language if, else could be a keyword, and so on.

- This is the code for recognizing keywords:

```python
def keyword(self, *keywords):
    self.eat_whitespace()
    if self.pos >= self.len:
        raise ParseError( self.pos + 1,
                          'Expected %s but got end of string',
                          ','.join(keywords)
                        )

    for keyword in keywords:
        low = self.pos + 1
        high = low + len(keyword)
        if self.text[low:high] == keyword:
            self.pos += len(keyword)
            self.eat_whitespace()
            return keyword
    raise ParseError( self.pos + 1,
                      'Expected %s but got %s',
                      ','.join(keywords),
                      self.text[self.pos + 1]
                    )
```

Extracting keywords and symbols

- This method takes keywords, such as self.keyword('if', 'and', 'or'). The method strips whitespace and then checks if we've run out of text to parse. It then iterates through each keyword, checking if the keyword exists in the text. If it does find something, we'll strip the whitespace following the keyword, and then return the keyword.

- However, if it doesn't find something, it exits the loop which raises ParseError, stating that the keywords couldn't be found.

A helper for matching multiple productions

- In the previous sections, you've noticed that we use exceptions to report errors. We also know that to write a recursive descent parser, you write functions that recognize a token or a production rule. Now, imagine that you wanted to parse a simple text, where you expect a number or a word. The production rule might look like:

```
Item  Number | Word
```

A helper for matching multiple productions

• We'll also assume that the text contains a word. When the parser tries to look for a digit (perhaps by using char()), it won't find it, and this causes it to raise an exception. In addition, you have to strip the whitespace before and after matching a production rule. So, the code for item() might look like this:

```python
def item(self):
    self.eat_whitespace()
    try:
        rv = self.number()
    except ParseError:
        rv = self.word()
    self.eat_whitespace()
    return rv
```

A helper for matching multiple productions

- This already looks like a lot to implement a simple rule! Imagine what it would be like to implement a complex rule — you'd have to use try...except blocks all over the place.

- So, we'll write a match() function that'll strip the whitespace and try to match multiple rules. The function is as follows:

```python
def match(self, *rules):
    self.eat_whitespace()
    last_error_pos = -1
    last_exception = None
    last_error_rules = []
    for rule in rules:
        initial_pos = self.pos
        try:
            rv = getattr(self, rule)()
            self.eat_whitespace()
            return rv
        except ParseError as e:
            self.pos = initial_pos
            if e.pos > last_error_pos:
                last_exception = e
                last_error_pos = e.pos last_error_rules.clear()
                last_error_rules.append(rule)
            elif e.pos == last_error_pos:
                last_error_rules.append(rule)

    # continued def level
    if len(last_error_rules) == 1:
        raise last_exception
    else:
        raise ParseError(
            last_error_pos,
            'Expected %s but got %s',
            ','.join(last_error_rules),
            self.text[last_error_pos]
        )
```

A helper for matching multiple productions

- Before we discuss how it works, let's see how simple it is to rewrite our previous example using match():

```python
def item(self):
    return self.match('number', 'word')
```

A helper for matching multiple productions

- So, how does that work? match() takes a method name to run (such as number or word in the above example). First, it gets rid of the whitespace at the beginning. Then, it iterates over all the method names and fetches each method using its name via getattr(). It then tries to invoke the method, and if all goes well, it'll strip whitespace after the text as well. Finally, it returns the value that it received by calling the method

A helper for matching multiple productions

However, if there's an error, it resets self.pos to the value that was there before trying to match a rule. Then, it tries to select a good error message by using the following rules:

- When matching multiple rules, many of them might generate an error. The error that was generated after parsing most of the text is probably the error message we want.

  To see why this is the case, consider the string "abc1". Trying to call number() would fail at position 0, whereas word() would fail at position 2. Looking at the string, it's quite likely that the user wanted to enter a word, but made a typo.

- If two or more erroring rules end up in a "tie", we prefer to tell the user about all the rules which failed.

Looking ahead at what's next – helper functions

- At this point, we have all the necessary things in our parser, and all failures throw exceptions. However, sometimes we want to only match a character, keyword or rule if it's there, without the inconvenience of handling exceptions.

- We'll introduce three small helpers to do just that. In the case of an exception when looking for stuff, these will return None:

```python
def maybe_char(self, chars=None):
    try:
        return self.char(chars)
    except ParseError:
        return None


def maybe_match(self, *rules):
    try:
        return self.match(*rules)
    except ParseError:
        return None


def maybe_keyword(self, *keywords):
    try:
        return self.keyword(*keywords)
    except ParseError:
        return None
```

Looking ahead at what's next – helper functions

- Using these functions are easy. This is how you might use them:

```python
operator = self.maybe_keyword('+', '-'):
if operator == '+':
    # add two numbers
elif operator == '-':
    # subtract two numbers
else: # operator is None
    # do something else
```

The Definitive Guide

First parser example: a calculator

First parser example: a calculator

- Now that we've built the foundation for writing a parser, we'll build our first parser. It'll be able to parse mathematical expressions with additions, subtractions, multiplications, divisions, and also handle parenthesized expressions like "(2 + 3) * 5".

- We will begin by visualizing the grammar in the form of production rules.

Production rules for the calculator grammar

- We've previously seen a grammar to handle everything except for parenthesized expressions:

```
Expression → Term ( ( "+" | "-" ) Term )*
Term → Number ( ( "*" | "/" ) Number )*
```

- Now, let's think about how parenthesized expressions fit in to this grammar. When evaluating "(2 + 3) * 5", we'd have to calculate "(2 + 3)" and reduce it to a number. Which means that in the above grammar, the term "Number" can either refer to a parenthesized expression, or something that's really a number, like "5".

Production rules for the calculator grammar

- So, we'll tweak our rules to accommodate both. In the rule for "Term", we'll replace "Number" for a more appropriate term like "Factor". "Factor" can then either refer to a parenthesized expression, or a "Number". The modified grammar looks like this:

```
Expression ➔ Term ( ( "+" | "-" ) Term )*
Term ➔ Factor ( ( "*" | "/" ) Factor )*
Factor ➔ "(" Expression ")" | Number
```

- With that out of the way, let's implement the production rules!

Implementing the parser

- We'll implement our Calculator parser by extending the base Parser class. We begin defining the class like so:

```python
class CalcParser(Parser):
    def start(self):
        return self.expression()
```

Implementing the parser

- Previously, when implementing the base parser class, we had a start() method to start parsing. We'll simply call the expression() method here, which we'll define as below:

```python
def expression(self):
    rv = self.match('term')
    while True:
        op = self.maybe_keyword('+', '-')
        if op is None:
            break
        term = self.match('term')
        if op == '+':
            rv += term
        else:
            rv -= term
    return rv
```

Implementing the parser

- The grammar rule for `Expression` → `Term ( ( "+" | "-" ) Term )*`. So, we start by reading a term from the text. We then set up an infinite loop, and look for a "+" or "-". If we don't find either, this means that we've finished reading through the list of additional terms as given by ("+" | "-") Term)*, so we can break out of the loop.

- Otherwise, we read another term from the text. Then, depending upon the operator, we either add or subtract it from the initial value. We continue with this process until we fail to get a "+" or "-", and return the value at the end.

Implementing the parser

- We'll implement term() in the same way:

```python
def term(self):
    rv = self.match('factor')
    while True:
        op = self.maybe_keyword('*', '/')
        if op is None:
            break
        term = self.match('factor')
        if op == '*':
            rv *= term
        else:
            rv /= term
    return rv
```

Implementing the parser

- Next, we'll implement "Factor". We'll try to read a left parenthesis first, which means that there's a parenthesized expression. If we do find a parenthesis, we read an expression and the closing parenthesis, and return the value of the expression. Otherwise, we simply read and return a number.

```python
def factor(self):
    if self.maybe_keyword('('):
        rv = self.match('expression')
        self.keyword(')')
        return rv
    return self.match('number')
```

- In the next section, we're going to make our parser recognize a number.

Recognizing numbers

- In order to recognize a number, we need to look at the individual characters in our text. Numbers consist of one or more digits, optionally followed by a decimal part. The decimal part has a period (.), and followed by at least one digit. In addition, numbers may have a sign such as "+" or "-" before them, such as "-124.33".

- We'll implement the number() method like so:

```python
def number(self):
    chars = []
    sign = self.maybe_keyword('+', '-')
    if sign is not None:
        chars.append(sign)
    chars.append(self.char('0-9'))
    while True:
        char = self.maybe_char('0-9')
        if char is None:
            break
        chars.append(char)
    if self.maybe_char('.'):
        chars.append('.')
        chars.append(self.char('0-9'))
        while True:
            char = self.maybe_char('0-9')
            if char is None:
                break
            chars.append(char)
    rv = float(''.join(chars))
    return rv
```

Recognizing numbers

- Although the function is long, it's fairly simple. We have a chars list in which we put the characters of the number. First, we look at any "+" or "-" symbols that may be present before the number. If it is present, we add it to the list. Then, we look for the first mandatory digit of the number and continue looking for more digits using the maybe_char() method. Once we get None through maybe_char, we know that we're past the set of digits, and terminate the loop.

- Similarly, we look for the decimal part and its digits. Finally, we convert all the characters into a string, which in turn, we convert to a floating point number.

An interface for our parser

- We're done building our parser. As a final step, we'll add a little bit of code in the global scope that'll allow us to enter expressions and see results:

```python
if __name__ == '__main__':
    parser = CalcParser()
    while True:
        try:
            print(parser.parse(input('> ')))
        except KeyboardInterrupt:
            print()
        except (EOFError, SystemExit):
            print()
            break
        except (ParseError, ZeroDivisionError) as e:
            print('Error: %s' % e)
```

An interface for our parser

- If you've followed along so far, congratulations! You have built your first recursive descent parser!

The Definitive Guide

Second parser example: an "extended" JSON parser

Second parser example: an "extended" JSON parser

- JSON is a data interchange format that supports basic data structures like numbers, strings and lists. It's a very widely used format, although it's simplicity means that it's lacking things like comments and strict about the things it accepts. For example, you can't have a trailing comma in a list or have an explicit "+" in front of a number to indicate its sign.

- Since Python already has a JSON module, we're going to aim for a little higher and build a JSON parser that supports comments, trailing commas and unquoted strings.

- Implementing this parser will teach you how to handle comments. It'll also illustrate how making an easy to use language can make parsing complicated, and how you can deal with such situations.

Second parser example: an "extended" JSON parser

```
{
        # Comments begin with a '#' and continue till the end of line.

        # You can skip quotes on strings if they don't have hashes,
        # brackets or commas.
        Size: 1.5x,

        # Trailing commas are allowed on lists and maps.
        Things to buy: {
                Eggs : 6,
                Bread: 4,
                Meat : 2,
        },

        # And of course, plain JSON stuff is supported too!
        "Names": ["John", "Mary"],
        "Is the sky blue?": true
}
```

Production rules for the JSON grammar

- Our extended JSON sticks to the same data types that JSON supports. JSON has four primitive data types — null, booleans, numbers and strings, and two complex data types — lists (or arrays), and maps (also called hashes or dictionaries). Lists and hashes look similar to the way they do in Python.

Production rules for the JSON grammar

Since JSON data would consist of one of these types, you might write the production rules like so:

```
Start ➔ AnyType
AnyType ➔  ComplexType | PrimitiveType
```

Then, you can break up these two types into JSON's types:

```
ComplexType ➔ List | Map
PrimitiveType ➔ Null | Boolean | Number | String
```

Production rules for the JSON grammar

- This grammar is fine for parsing regular JSON but needs a bit of tweaking for our case. Since we're going to support unquoted strings, which means "1.5" (without the quotes) is a number, but "1.5x" (again, without the quotes), is a string. Our current grammar would read the "1.5" from "1.5x", and then cause an error because "x" wasn't expected after a number.

- This means that first, we'd have to get the full set of unquoted characters. Next, we'll analyze the characters and determine if it's a number or a string. So, we'll combine numbers and unquoted strings into a single category, "Unquoted". The quoted strings are fine, so we'll separate them into another category, "QuotedString".

Production rules for the JSON grammar

- Our modified production rule for "PrimitiveType" now looks like this:

```
PrimitiveType → Null | Boolean |
                QuotedString | Unquoted
```

- In addition, the order of the rules is important. Since we have unquoted keys, we must first try parsing the text as a null or a boolean. Otherwise, we might end up recognizing "null" or "true" as an unquoted string.

Implementing the parser and dealing comments

- To implement the JSON parser, we'll begin by extending the base parser class like so:

```
class JSONParser(Parser):
    # ...
```

- We will first tackle the problem of dealing with comments. If you think about it, comments are really similar to whitespace — they occur in the same places where whitespaces can occur, and they can be discarded just like whitespace. So, we'll reimplement eat_whitespace() to deal with the comments, like so:

Implementing the parser and dealing comments

```python
def eat_whitespace(self):
    is_processing_comment = False
    while self.pos < self.len:
        char = self.text[self.pos + 1]
        if is_processing_comment:
            if char == '\n':
                is_processing_comment = False
        else:
            if char == '#':
                is_processing_comment = True
            elif char not in ' \f\v\r\t\n':
                break
        self.pos += 1
```

Implementing the parser and dealing comments

- Here, we need to keep track of whether we're processing whitespaces or a comment. We loop over the text character-by-character, checking if we have whitespaces or a #. When there's a #, we update is_processing_comment to True and in the next iterations of the while loop, we can safely discard all characters until we reach the end of the line. However, when processing whitespace characters, we must stop once a non-whitespace character appears.

Implementing the parser and dealing comments

- Next, we'll implement the production rules and the start() method. The input text will have a JSON type contained in it, so we'll simply call any_type() in the start() method:

```python
def start(self):
    return self.match('any_type')
def any_type(self):
    return self.match('complex_type', 'primitive_type')
def primitive_type(self):
    return self.match('null', 'boolean', 'quoted_string', 'unquoted')
def complex_type(self):
    return self.match('list', 'map')
```

Parsing lists and maps

• Previously, we've seen how to parse comma separated lists. JSON's lists are just a comma separated list with square brackets tacked onto them, and the list may have zero or more elements. Let's see how you would implement parsing a list:

```python
def list(self):
    rv = []
    self.keyword('[')
    while True:
        item = self.maybe_match('any_type')
        if item is None: break
        rv.append(item)
        if not self.maybe_keyword(','): break
    self.keyword(']')
    return rv
```

Parsing lists and maps

- We begin by reading off the initial square bracket followed by an item from the list using self.maybe_match('any_type'). If we failed to get an item, this indicates that we're probably done going through all the items, so we break out of the loop. Otherwise, we add the item to the list. We then try to read a comma from the list, and the absence of a comma also indicates that we're done with the list.

- Similarly, maps are just comma separated lists of "pairs" with braces, where a pair is a string key followed by a colon(:) and a value. Unlike Python's dicts which can have any "hashable" type as a key (which includes ints and tuples), JSON only supports string keys.

- This is how you'd implement the rules for maps and pairs:

```python
    def map(self):
        rv = {}
        self.keyword('{')
        while True:
            item = self.maybe_match('pair')
            if item is None:
                break
            rv[item[0]] = item[1]
            if not self.maybe_keyword(','):
                break
        self.keyword('}')
        return rv
    def pair(self):
        key = self.match('quoted_string', 'unquoted')
        if type(key) is not str:
            raise ParseError( self.pos + 1,
                             'Expected string but got number',
                             self.text[self.pos + 1] )

        self.keyword(':')
        value = self.match('any_type')
        return key, value
```

Parsing lists and maps

- In pair(), we try to read a "QuotedString" or "Unquoted" for the key. As we've mentioned previously, "Unquoted" can either return a number or a string, so we explicitly check if the key that we've read is a string. pair() then returns a tuple with a key and value, and map() calls pair() and adds these to a Python dict.

Recognizing null and boolean

- Now that the main parts of our parser are implemented, let us begin by recognizing simple tokens such as null and booleans:

```python
def null(self):
    self.keyword('null')
    return None
def boolean(self):
    boolean = self.keyword('true', 'false')
    return boolean[0] == 't'
```

- Python's None is the closest analogue to JSON's null. In the case of booleans, the first characters is sufficient to tell whether it's true or false, and we return the result of the comparison.

Recognizing unquoted strings and numbers

- Before moving on to recognize unquoted strings, let us first define the set of acceptable characters. We'll leave out anything that's considered special in such as braces, quotes, colons, hashes (since they are used in comments) and backslashes (because they're used for escape sequences). We'll also include spaces and tabs in the acceptable set of characters so that you can write strings like "Things to buy" without using quotes.

Recognizing null and boolean

- So, what are the characters that we want to accept? We can use Python to figure this out:

```
>>> import string
>>> ''.join(sorted(set(string.printable) - set('{}[]:#"\'\f\v\r\n')))
'\t !$%&()*+,-
./0123456789;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ\\^_`abcdef
ghijklmnopqrstuvwxyz|~'
```

Recognizing null and boolean

- The next question is, how do you figure out if the text you read is a number or a string? The answer is — we cheat! Since Python's int() and float() can take a string and return a number, we'll use them and if those result in a ValueError, we'll return a string. As for when to use int() or float(), we'll use a simple rule. If the text contains "E", "e" or ".", (for example, "12.3" or "2e10"), we'll call float(); otherwise, we'll use int().

```python
def unquoted(self):
    acceptable_chars = '0-9A-Za-z \t!$%&()*+./;<=>?^_`|~-'
    number_type = int
    chars = [self.char(acceptable_chars)]

    while True:
        char = self.maybe_char(acceptable_chars)
        if char is None:
            break
        if char in 'Ee.':
            number_type = float
        chars.append(char)

    rv = ''.join(chars).rstrip(' \t')
    try:
        return number_type(rv)
    except ValueError:
        return rv
```

Recognizing null and boolean

- Since matching rules are handled by match(), this takes care of stripping any initial whitespace before unquoted() can run. In this way, we can be sure that unquoted() won't return a string consisting of only whitespaces. Any whitespace at the end is stripped off before we parse it as a number or return the string itself.

Recognizing quoted strings

- Quoted strings are fairly simple to implement. Most programming languages (including Python) have single and double quotes that behave in the same way, and we'll implement both of them.

- We'll support the following escape sequences — \b (backspace), \f (line feed), \n (newline), \r (carriage return), \t (tab) and \u(four hexadecimal digits) where those digits are used to represent an Unicode "code point". For anything else that has a backslash followed by a character, we'll ignore the backslash. This handles cases like using the backslash to escape itself (\\) or escaping quotes (\").

```python
def quoted_string(self):
    quote = self.char('"\'')
    chars = []
    escape_sequences = { 'b': '\b', 'f': '\f', 'n': '\n', 'r': '\r', 't': '\t' }
    while True:
        char = self.char()
        if char == quote: break
        elif char == '\\':
            escape = self.char()
            if escape == 'u':
                code_point = []
                for i in range(4):
                    code_point.append(self.char('0-9a-fA-F'))
                chars.append(chr(int(''.join(code_point), 16)))
            else:
                chars.append(escape_sequences.get(char, char))
        else:
            chars.append(char)
    return ''.join(chars)
```

Recognizing quoted strings

- We first read a quote and then read additional characters in a loop. If this character is the same type of quote as the first character we read, we can stop reading further and return a string by joining the elements of chars.

- Otherwise, we check if it's an escape sequence and handle it in the aforementioned way, and add it to chars. Finally, if it matches neither of them, we treat it as a regular character and add it to our list of characters.

An interface for our JSON parser

- To make sure we can interact with our parser, we'll add a few lines of code that accepts input and parses it as JSON. Again, this code will be in global scope:

```python
if __name__ == '__main__':
    import sys
    from pprint import pprint
    parser = JSONParser()
    try:
        pprint(parser.parse(sys.stdin.read()))
    except ParseError as e:
        print('Error: '+ str(e))
```

The Definitive Guide
Building other kinds of parsers

Building other kinds of parsers

- Now that we've gone through building two parsers, you might have a fairly good idea of how to roll your own for the task at hand. While every language is unique (and therefore, their parsers), there are some common situations that we haven't covered. For the sake of brevity, we won't cover any code in these sections.

Whitespace sensitive languages

- Before we begin, let us clarify that we're not referring to languages that use whitespace-based indentation — we'll cover them in the next section. Here, we'll discuss languages where the meaning of things change based on the whitespace you put between elements.

- An example of such a language would be where "a=10" is different than "a = 10". With bash and some other shells, "a=10" sets an environment variable, whereas "a = 10" runs the program "a" with "=" and "10" as command-line arguments. You can even combine the two! Consider this:

```
a=10 b=20 c = 30
```

Whitespace sensitive languages

- This sets the environment variables "a" and "b" only for the program "c". The only way to parse such a language is to handle the whitespaces manually, and you'd have to remove all the eat_whitespace() calls in keyword() and match(). This is how you might write the production rules:

```
Command ➔ Variables  ProgramArguments?
Variables ➔  VariableDeclaration (Whitespace VariableDeclaration)*
VariableDeclaration ➔ Name  "="  VariableValue
```

Whitespace based indentation

- Languages like C and Javascript use curly braces to indicate the body of loops and if-statements. However, Python uses indentation for the same purpose, which makes parsing tricky.

- One of the methods to handle this is to introduce a term such as "INDENT" to keep track of indented statements. To see how this would work, consider the following production rule:

```
WhileLoop → "while"  Condition (  INDENT  Statement )*
```

Whitespace based indentation

- After matching a condition, our parser would look for INDENT. Since this is the first time it's trying to match INDENT, it takes whatever whitespace (except for newlines) that appears along with a non-whitespace character (since that would be part of a statement).

- In subsequent iterations, it looks for the same whitespace in the text. If it encounters a different whitespace, it can raise an error. However, if there's no whitespace at all, it indicates that the "while" loop is finished.

Whitespace based indentation

- For nested indentations, you'd need to maintain a list of all indentations that you're currently handling. When the parser handles a new indented block, it should add the new indentation string on the list. You can tell if you're done parsing the block, by checking that you were able to retrieve less whitespace than you did previously. At this point, you should pop off the last indentation off the list.

End of Chapter 9B