Chapter 3: CONTEXT-FREE GRAMMARS AND PARSING – Part 2
3.3 Parse Trees and Abstract Syntax Trees

3.3.1 Parse trees
1. Derivation V.S. Structure
   • Derivations do not uniquely represent the structure of the strings
     – There are many derivations for the same string.
   • The string of tokens:
     – (number - number) * number
   • There exist two different derivations for above string

(1) exp => exp op exp
(2) => exp op number
(3) => exp * number
(4) => ( exp ) * number
(5) =>( exp op exp ) * number
(6) => (exp op number) * number
(7) => (exp - number) * number
(8) => (number - number) * number

---------------

(1) exp => exp op exp
(2) => (exp) op exp
(3) => (exp op) op exp
(4) => (number op exp) op exp
(5) => (number - exp) op exp
(6) => (number - number) op exp
(7) => (number - number) * exp
(8) => (number - number) * number

2. Parsing Tree
   • A parse tree corresponding to a derivation is a labeled tree.
     – The interior nodes are labeled by non-terminals, the leaf nodes are labeled by terminals;
     – And the children of each internal node represent the replacement of the associated non-terminal in one step of the derivation.
   • The example:
     – exp => exp op exp => number op exp => number + exp => number + number
   • The example:
     – exp => exp op exp => number op exp => number + exp => number + number
   • Corresponding to the parse tree:
• The above parse tree is corresponds to the three derivations:
  • **Left most derivation**
    (1) \( exp \Rightarrow exp \ op \ exp \)
    (2) \( \Rightarrow \ number \ op \ exp \)
    (3) \( \Rightarrow \ number \ + \ exp \)
    (4) \( \Rightarrow \ number \ + \ number \)
  • **Right most derivation**
    (1) \( exp \Rightarrow exp \ op \ exp \)
    (2) \( \Rightarrow \ exp \ op \ number \)
    (3) \( \Rightarrow \ exp \ + \ number \)
    (4) \( \Rightarrow \ number \ + \ number \)
  • **Neither leftmost nor rightmost derivation**
    (1) \( exp \Rightarrow exp \ op \ exp \)
    (2) \( \Rightarrow \ exp \ + \ exp \)
    (3) \( \Rightarrow \ number \ + \ exp \)
    (4) \( \Rightarrow \ number \ + \ number \)
• Generally, a parse tree corresponds to many derivations
  – represent the same basic structure for the parsed string of terminals.
• It is possible to distinguish particular derivations that are uniquely associated with the parse tree.
  • A leftmost derivation:
    – A derivation in which the leftmost non-terminal is replaced at each step in the derivation.
    – Corresponds to the preorder numbering of the internal nodes of its associated parse tree.
  • A rightmost derivation:
    – A derivation in which the rightmost non-terminal is replaced at each step in the derivation.
    – Corresponds to the postorder numbering of the internal nodes of its associated parse tree.
• The parse tree corresponds to the first derivation.

```
1 exp
   |
  2 exp 3 op 4 exp
     |
number + number
```

Example: The expression \((34-3)*42\)
• The parse tree for the above arithmetic expression

```
1 exp
   |
  4 exp 3 op 2 exp
     |
    5 exp *
      |
number
```

```
8 exp 7 op 6 exp
    |
number - number
```
3.3.2 Abstract syntax trees

1. Why Abstract Syntax-Tree

- The parse tree contains more information than is absolutely necessary for a compiler
- For the example: 3*4

```
exp
  / \  /
exp op exp
  |   |
number * number
  (3) (4)
```

- The principle of syntax-directed translation
  - The meaning, or semantics, of the string 3+4 should be directly related to its syntactic structure as represented by the parse tree.
  - In this case, the parse tree should imply that the value 3 and the value 4 are to be added.
  - A much simpler way to represent this same information, namely, as the tree

```
+  3       4
```

Tree for expression (34-3)*42

- The expression (34-3)*42 whose parse tree can be represented more simply by the tree:

```
*  - 42
  34  3
```

- The parentheses tokens have actually disappeared
  - still represents precisely the semantic content of subtracting 3 from 34, and then multiplying by 42.

2. Abstract Syntax Trees or Syntax Trees

- Syntax trees represent abstractions of the actual source code token sequences,
  - The token sequences cannot be recovered from them (unlike parse trees).
  - Nevertheless they contain all the information needed for translation, in a more efficient form than parse trees.

Examples

- Example 3.8:
  - The grammar for simplified if-statements

```
statement → if-stmt | other
if-stmt → if ( exp ) statement | if ( exp ) statement else statement
exp → 0 | 1
```

- The parse tree for the string:
  - if (0) other else other

```
statement
  /   
if-stmt
    / 
if ( exp ) statement else statement
     /   
0 other other
```
• Using the grammar of Example 3.6
  \[ \text{statement} \rightarrow \text{if-stmt} \mid \text{other} \]
  \[ \text{if-stmt} \rightarrow \text{if ( exp ) statement else-part} \]
  \[ \text{else-part} \rightarrow \text{else statement} \mid \epsilon \]
  \[ \text{exp} \rightarrow 0 \mid 1 \]

• This same string has the following parse tree:
  \[ \text{if ( 0 ) other else other} \]
  \[ \text{statement} \]
  \[ \text{if-stmt} \]
  \[ \text{if ( exp ) statement else-part} \]
  \[ 0 \text{ other else other} \]
  \[ \text{statement} \]

• A syntax tree for the previous string (using either the grammar of Example 3.4 or 3.6) would be:
  \[ \text{if ( 0 ) other else other} \]

• Example 3.9:
  \[ \text{The grammar of a sequence of statements separated by semicolons from Example 3.7:} \]
  \[ \text{stmt-sequence} \rightarrow \text{stmt ; stmt-sequence } \mid \text{stmt} \]
  \[ \text{stmt} \rightarrow \text{s} \]

• The string \text{s ; s ; s} has the following parse tree
  with respect to this grammar:

• A possible syntax tree for this same string is:

• Bind all the statement nodes in a sequence together with just one node, so that the previous syntax tree would become
3.4 Ambiguity

1. What is Ambiguity
   - Parse trees and syntax trees uniquely express the structure of syntax
   - But it is possible for a grammar to permit a string to have more than one parse tree
   - For example, the simple integer arithmetic grammar:
     \[ \text{exp} \rightarrow \text{exp} \text{ op } \text{exp} | ( \text{exp} ) | \text{number} \]
     \[ \text{op} \rightarrow + | - | * \]
     The string: 34-3*42
     This string has two different parse trees.

Corresponding to the two leftmost derivations

\[
\begin{align*}
\text{exp} \Rightarrow & \text{exp op exp} \\
\Rightarrow & \text{exp op exp exp} \\
\Rightarrow & \text{number} \text{ op exp exp} \\
\Rightarrow & \text{number} - \text{exp exp} \\
\Rightarrow & \text{number} - \text{number} \text{ op exp} \\
\Rightarrow & \text{number} - \text{number} * \text{exp} \\
\Rightarrow & \text{number} - \text{number} * \text{number}
\end{align*}
\]

The associated syntax trees are

\[
\begin{align*}
\text{*} & \quad 42 \\
34 & \quad 3 \quad \text{AND}
\end{align*}
\]

2. An Ambiguous Grammar
   - A grammar that generates a string with two distinct parse trees
   - Such a grammar represents a serious problem for a parser
     - Not specify precisely the syntactic structure of a program
   - In some sense, an ambiguous grammar is like a non-deterministic automaton
     - Two separate paths can accept the same string
   - Ambiguity in grammars cannot be removed nearly as easily as non-determinism in finite automata
     - No algorithm for doing so, unlike the situation in the case of automata
• Ambiguous grammars always fail the tests that we introduce later for the standard parsing algorithms
  – A body of standard techniques have been developed to deal with typical ambiguities that come up in programming languages.

3. Two Basic Methods dealing with Ambiguity
• One is to state a rule that specifies in each ambiguous case which of the parse trees (or syntax trees) is the correct one, called a disambiguating rule.
  – The advantage: it corrects the ambiguity without changing (and possibly complicating) the grammar.
  – The disadvantage: the syntactic structure of the language is no longer given by the grammar alone.
• The alternative is to Change the grammar into a form that forces the construction of the correct parse tree, thus removing the ambiguity.
• Of course, in either method we must first decide which of the trees in an ambiguous case is the correct one.

4. Remove The Ambiguity in Simple Expression Grammar
• Simply state a disambiguating rule that establishes the relative precedence of the three operations represented.
  – The standard solution is to give addition and subtraction the same precedence, and to give multiplication a higher precedence.
• A further disambiguating rule is the associativity of each of the operations of addition, subtraction, and multiplication.
  – Specify that all three of these operations are left associative
• Specify that an operation is nonassociative
  – A sequence of more than one operator in an expression is not allowed.
• For instance, writing simple expression grammar in the following form: fully parenthesized expressions
  \[
  \begin{align*}
  \text{exp} & \rightarrow \text{factor op factor} \mid \text{factor} \\
  \text{factor} & \rightarrow ( \text{exp} ) \mid \text{number} \\
  \text{op} & \rightarrow + \mid - \mid * \\
  \end{align*}
  \]
  • Strings such as 34-3-42 and even 34-3*42 are now illegal, and must instead be written with parentheses
    – such as (34-3) -42 and 34- (3*42).
  • Not only changed the grammar, also changed the language being recognized.

3.4.2 Precedence and Associativity
1. Group of Equal Precedence
• The precedence can be added to our simple expression grammar as follows:
  \[
  \begin{align*}
  \text{exp} & \rightarrow \text{exp addop exp} \mid \text{term} \\
  \text{addop} & \rightarrow + \mid - \\
  \text{term} & \rightarrow \text{term mulop term} \mid \text{factor} \\
  \text{mulop} & \rightarrow * \\
  \text{factor} & \rightarrow ( \text{exp} ) \mid \text{number} \\
  \end{align*}
  \]
  • Addition and subtraction will appear "higher" (that is, closer to the root) in the parse and syntax trees
    – Receive lower precedence.

2. Precedence Cascade
• Grouping operators into different precedence levels.
  – Cascade is a standard method in syntactic specification using BNF.
• Replacing the rule
  \[
  \text{exp} \rightarrow \text{exp addop exp} \mid \text{term}
  \]
3. Removal of Ambiguity
   - Removal of ambiguity in the BNF rules for simple arithmetic expressions
     - write the rules to make all the operations left associative
       exp \rightarrow exp \ addop \ term \ | \ term
       addop \rightarrow + \ | \ -
       term \rightarrow term \ mulop \ factor \ | \ factor
       mulop \rightarrow *
       factor \rightarrow ( \ exp \ ) \ | \ number

   New Parse Tree
   - The parse tree for the expression 34-3*42 is
     \[
     \begin{array}{c}
     \text{exp} \\
     \text{addop} \\
     \text{term} \\
     \text{term} \\
     \text{mulop} \\
     \text{factor} \\
     \text{number} \\
     \text{number}
     \end{array}
     \]

   - The parse tree for the expression 34-3-42
     \[
     \begin{array}{c}
     \text{exp} \\
     \text{addop} \\
     \text{term} \\
     \text{term} \\
     \text{mulop} \\
     \text{factor} \\
     \text{number} \\
     \text{number}
     \end{array}
     \]

   - The precedence cascades cause the parse trees to become much more complex
   - The syntax trees, however, are not affected

3.4.3 The dangling else problem
1. An Ambiguity Grammar
   - Consider the grammar from:
     \[
     \begin{array}{c}
     \text{statement} \rightarrow \text{if-stmt} \ | \ \text{other} \\
     \text{if-stmt} \rightarrow \text{if} \ ( \ exp \ ) \ \text{statement} \\
     | \ \text{if} \ ( \ exp \ ) \ \text{statement} \ \text{else} \ \text{statement} \\
     \text{exp} \rightarrow 0 \ | \ 1
     \end{array}
     \]
   - This grammar is ambiguous as a result of the optional else. Consider the string
     if (0) if (1) other else other
This string has two parse trees:

```
statement
  |  
  if-stmt
    | 
  if (exp) statement else statement
    | 
  0 if-stmt other
        | 
    if (exp) statement
        | 
  1 other
```

### 2. Dangling else problem

- Which tree is correct depends on associating the single else-part with the first or the second if-statement.
  - The first associates the else-part with the first if-statement;
  - The second associates it with the second if-statement.
- This ambiguity called dangling else problem
- This disambiguating rule is the most closely nested rule
  - implies that the second parse tree above is the correct one.

#### An Example

- For example:
  ```
  if (x != 0)
      if (y == 1/x) ok = TRUE;
      else z = 1/x;
  ```
- Note that, if we wanted we could associate the else-part with the first if-statement by using brackets {...} in C, as in
  ```
  if (x != 0)
      { if (y == 1/x) ok = TRUE; }
  ```
- else z = 1/x;

### 3. A Solution to the dangling else ambiguity in the BNF

```
statement → matched-stmt | unmatched-stmt
matched-stmt → if (exp) matched-stmt else matched-stmt | other
unmatched-stmt → if (exp) statement                
                | if (exp) matched-stmt else unmatched-stmt
exp → 0 | 1
```

- Permitting only a matched-stmt to come before an else in an if-statement, thus
  - forcing all else-parts to be matched as soon as possible.
The associated parse tree for our sample string now becomes

\[
\text{statement} \\
| \\
\text{unmatched-stmt} \\
| \\
\text{if } ( \text{exp} ) \text{statement} \\
| \\
| \\
| \\
| \\
| \\
| \\
| \\
| \\
\]

Which indeed associates the else part with the second if-statement

3.5 Extended Notations: EBNF and Syntax Diagrams

3.5.1 EBNF Notation

1. Special Notations for Repetitive Constructs

- Repetition
  - A → A α | β (left recursive), and
  - A → α A | β (right recursive)
  - where α and β are arbitrary strings of terminals and non-terminals, and
  - In the first rule β does not begin with A and
  - In the second β does not end with A
- Notation for repetition as regular expressions use, the asterisk *.
  - A → β α*, and
  - A → α* β
- EBNF opts to use curly brackets {...} to express repetition
  - A → β {α}, and
  - A → {α} β
- The problem with any repetition notation is that it obscures how the parse tree is to be constructed, but, as we have seen, we often do not care.

Examples

- Example: The case of statement sequences
- The grammar as follows, in right recursive form:

\[
\text{stmt-Sequence} \rightarrow \text{stmt} ; \text{stmt-Sequence} | \text{stmt} \\
\text{stmt} \rightarrow s \\
\]

- In EBNF this would appear as

\[
\text{stmt-sequence} \rightarrow \{ \text{stmt} ; \} \text{stmt} \text{ (right recursive form)} \\
\text{stmt-sequence} \rightarrow \text{stmt} \{ ; \text{stmt}\} \text{ (left recursive form)} \\
\]

- A more significant problem occurs when the associativity matters

\[
\text{exp} \rightarrow \text{exp addop term} | \text{term} \\
\text{exp} \rightarrow \text{term} \{ \text{addop term} \} \\
\]

(implicitly left associativity)

\[
\text{exp} \rightarrow \{ \text{term addop} \} \text{ term} \\
\]

(implicitly right associativity)
2. Special Notations for *Optional Constructs*

- Optional constructs are indicated by surrounding them with square brackets \([...]\).
- The grammar rules for if-statements with optional else-parts would be written as follows in EBNF:

```
statement \rightarrow if-stmt | other
if-stmt \rightarrow if \((\text{exp})\) statement [else statement]
```

```
exp \rightarrow 0 | 1
```

- `stmt-sequence \rightarrow stmt; stmt-sequence | stmt` is written as
- `stmt-sequence \rightarrow stmt [ ; stmt-sequence ]`

3.5.2 Syntax Diagrams

- **Syntax Diagrams:**
  - Graphical representations for visually representing EBNF rules.
- **An example:** consider the grammar rule
  
  `factor \rightarrow ( exp ) | number`

- **The syntax diagram:**

![Syntax Diagram]

- Boxes representing terminals and non-terminals.
- Arrowed lines representing sequencing and choices.
- Non-terminal labels for each diagram representing the grammar rule defining that Non-terminal.
- A round or oval box is used to indicate terminals in a diagram.
- A square or rectangular box is used to indicate non-terminals.

- A repetition: `A \rightarrow \{B\}`

![Repetition Diagram]

- An optional: `A \rightarrow [B]`

![Optional Diagram]

**Examples**

- Example: Consider the example of simple arithmetic expressions.

```
exp \rightarrow exp addop term | term
```
addop → + | -
term → term mulop factor | factor
mulop → *
factor → ( exp ) | number
  • This BNF includes associativity and precedence
  • The corresponding EBNF is
exp → term { addop term }
addop → + | -
term → factor { mulop factor }
mulop → *
factor → ( exp ) | number
  • The corresponding syntax diagrams are given as follows:

Examples

- Example: Consider the grammar of simplified if-statements, the BNF

Statement → if-stmt | other
if-stmt → if ( exp ) statement
  | if ( exp ) statement else statement
exp → 0 | 1
  • and the EBNF
statement → if-stmt | other
if-stmt → if ( exp ) statement [ else statement ]
exp → 0 | 1

The corresponding syntax diagrams are given in following figure.