Chapter 4

Lexical and Syntax Analysis

Complexity of Parsing

- Parsing algorithms that work for unambiguous grammar are complex and inefficient, with complexity O(n^3).
  - Too slow.
  - Algorithms usually backed up and reparse part of the sentence being analyzed.
  - Trade generality for efficiency.
    - Algorithms that work for only subsets of the set of all possible grammars O(n).

Recursive-Descent Parsing

- A general form of top-down parsing that may involve backtracking.
  - Backtracking parsers are rarely needed to parse programming languages constructs.

Recursive-Descent Parsing

- Recursive Descent Process
  - There is a subprogram for each nonterminal in the grammar, which can parse sentences that can be generated by that nonterminal
  - EBNF is ideally suited for being the basis for a recursive-descent parser, because EBNF minimizes the number of nonterminals

Recursive-Descent Parsing

- Coding process when there is only one RHS:
  - For each terminal symbol in the RHS, compare it with the next input token; if they match, continue, else there is an error
  - For each nonterminal symbol in the RHS, call its associated parsing subprogram

Recursive-Descent Parsing

- Coding process when A nonterminal that has more than one RHS:
  - The correct RHS is chosen on the basis of the next token of input (the lookahead)
  - The next token is compared with the first token that can be generated by each RHS until a match is found
  - If no match is found, there is a syntax error

Example

- Consider the grammar:

```
<S> ::= c<A>d
<A> ::= ab | a |  abc
```

- Input string: w = cad

LL Grammar Class

- L (left-to-right) L (leftmost derivation).
- What is the problem with the following grammar?

```
<NP> ::= <NP> <PP>
<VP> ::= <VP> <PP>
<S> ::= <S> and <S>
```

  - A left-recursive nonterminal can lead to the parser to recursively expand the same nonterminal over again in exactly the same way, leading to an infinite expansion of trees.

Left-recursive grammars

- A grammar is left-recursive if it contains a nonterminal category that has a derivation that includes itself anywhere along its leftmost branch.
  - Indirect left-recursion

```
<NP> ::= <Det> <Nom>
<Det> ::= <NP> …
```

  - These rules introduce left-recursion into the grammar since there is a derivation for the first element of the *NP*, the *Det*, that has an *NP* as its first constituent.

Eliminating left-recursion

- Weakly equivalent non-left-recursive grammar
  - Rewrite each left-recursive rule

A ? Ab | a ⇒ A *? aA′*
A′ ? bA′ | e

FIRST Set

- Given a string *a* of terminal and nonterminal symbols, FIRST(*a*) is the set of all terminal symbols that can begin any string derived from *a*
- If two different production X ? a_1 and X ? a_2 have the same LHS symbol *(X)* and their RHS have overlapping FIRST sets, then the grammar cannot be parsed using predictive parsing.

Pairwise Disjointness Test

- The other characteristic of grammars that disallows top-down parsing is the lack of pairwise disjointness
  - The inability to determine the correct RHS on the basis of one token of lookahead
  - For each nonterminal, A, in the grammar that has more than one RHS, for each pair of rules, A $\rightarrow \alpha_i$ and A $\rightarrow \alpha_j$, it must be true that
    FIRST(α_i) n FIRST(α_j) = ϕ
- Examples:
  A ® a | aB

Left Factoring

- Left factoring can resolve the previous problem:
  - Original grammar:

```
<S> ::= if <E> then <S> else <S>
<S> ::= if <E> then <S>
```

  - Left factoring the grammar:

```
<S> ::= if <E> then <S> <X>
<X> ::= ε | else <S>
```

LL(1) Grammars: Parsing Table

- A predictive parsing table for the following LL(1) grammar:

<E> ::= <T><E'>
<E'> ::= +<T><E'> | e
<T> ::= <F><T'>
<T'> ::= *<F><T'> | e
<F> ::= (<E>) | id

- A grammar whose parsing table has no multiply-defined entries is said to be LL(1) – left to right, leftmost derivation, one symbol of lookahead -

LL(1) Grammars: Sequence of Moves

- A sequence of moves for the following LL(1) grammar:

<E> ::= <T><E'>
<E'> ::= +<T><E'> | e
<T> ::= <F><T'>
<T'> ::= *<F><T'> | e
<F> ::= (<E>) | id

- With an input *id + id * id*

Bottom-up Parsing

- Attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root(the top).
  - This process can be think as *reducing* a string to the start symbol of a grammar.
  - At each *reduction* step a particular substring matching the RHS of a production is replaced by the symbol on the LHS of that production.
  - If the substring is chosen correctly at each step, a rightmost derivation is traced out in reverse.

Example

- Consider the grammar

<S> ::= a<A><B>e
<A> ::= <A>bc | b
<B> ::= d

- The sentence *abbcde* can be reduced to *S* by the following steps:

abbcde
a<A>bcde
a<A>de
a<A><B>e
<S>

Handles

- A handle of a string:
  - A substring that matches the RHS of a production
  - Reduction to the nonterminal on the LHS represents one step along the reverse of a rightmost derivation.
  - Sometime the leftmost substring that matches the RHS is not a handle because the reduction yields a string that cannot be reduced to the start symbol.

Example: Reduction Table

- Consider the grammar:

  <E> ::= <E>+<E>
  <E> ::= <E>*<E>
  <E> ::= (<E>)
  <E> ::= id

- The input string: $id_1 + id_2 * id_3$
- The sequence of reduction:

Shift-Reduce Parsing

- Two problems with parsing with handles
  - Locate substrings to be reduced in a right-sentential form
  - Determine what production to choose in case there is more than one production with that substring on the RHS
- A shift-reduce parser uses a stack to hold a grammar symbol and an input buffer to hold the string to be parsed.

Shift-Reduce Parsing

- $ is used to mark the bottom of the stack and also the right end of the input.
- Initially, the stack is empty

STACK	INPUT
$	*w$*

- The parser shifts zero or more input symbols onto the stack until a handle *ß* is on top of the stack.
- The parser then reduces *ß* to the left side of the appropriate production.

Shift-Reduce Parsing

- The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty.

STACK	INPUT
$S	*$*

- After that configuration, the parser halts and announces successful completion of parsing.

Shift-Reduce Parsing

- Four possible actions
  1. **Shift**: the next input is shifted onto the top of the stack.
  2. **Reduce:** the parser knows the right end of the handle is at the top of the stack. It must then locate the left end of the handle within the stack and decide with what nonterminal to replace the handle.
  3. **Accept:** successful completion of parsing.
  4. **Error:** a syntax error occurs and an error recovery routine is called.

Summary

- Syntax analysis is a common part of language implementation
- A lexical analyzer is a pattern matcher that isolates small-scale parts of a program
  - Detects syntax errors
  - Produces a parse tree
- A recursive-descent parser is an LL parser
  - EBNF
- Parsing problem for bottom-up parsers: find the substring of current sentential form
- The LR family of shift-reduce parsers is the most common bottom-up parsing approach