Top-Down Syntax Analysis

• for grammar with correct properties
  – write set of mutually recursive procedures to analyse
  – one procedure per non terminal in the grammar
• recognise and analyse - checks the output of the LA, call associated procedures
• Often defined using Regular Expressions and Extended BNF

Top-Down Syntax Analysis

• RE
  – Generates a set of strings of terminal symbols.
  • Useful for very simple languages.
  • Typically for identifiers and literals.
    – M\(t\)\(s\) – generates \(\{Me, Ms\}\)
    – ps\(t\)\(s\) – generates \(\{pt, pst, pstt, pstst, \ldots\}\)
    – ba\(n\)\(a\)^n – generates \(\{ba, bana, banana, bannana, \ldots\}\)

Top-Down Syntax Analysis

• EBNF
  – Combination of BNF and RE
  – Production rule is of the form \(N::= X\)
    • \(N\) is a nonterminal symbol and \(X\) is an extended RE
    – RE construction from both terminal and non terminal symbols.
  – Unlike BNF
    • RHS of EBNF may use |, *, ( and )
  – Unlike RE
    • RHS may contain non terminals as well as terminals.
    – Allows for recursive production rules which can generate self embedding statements

Top-Down Syntax Analysis

• In a grammar notation
  – optional alternatives
    \(U \rightarrow A \mid +B \mid -C\) (optional alternatives)
    same as \(U \rightarrow A \mid +B \mid -C \mid \text{null}\) (non optional alternatives)
  – repetition
    \(Y \rightarrow Q \mid +R \mid T \)^* 
    same as \(Y \rightarrow QZ \mid +RZ \mid TZ \mid \text{null}\)

Top-Down Syntax Analysis

• Production rules defining Expression and primary-Expr are mutually recursive
  – Can generate self-embedded expressions

Damien Costello, Dept of Computing & Maths, GMIT
Top-Down Syntax Analysis

- Two problems with this analysis
  - Backtracking
  - Left recursion
- Grammar fragment
  - $B \rightarrow D; B \mid D; S$

Top-Down SA

- Combining portions of alternatives can reduce the incidence of backtracking.
- Example: $B \rightarrow D; B \mid D; S$
  - Analyzer with input of $D; S$ and trying to find a $B$ phrase.
  - $D$ portion successfully analyzed, but $B$ portion will fail.
  - Has to backtrack to analyze as $D; S$.
  - This entails re-analyzing the $D$ part.
  - If analysis is done in opposite order.
- Same problem with $D; B$ input.
- Can remedy this by factoring the grammar.

Top-Down SA

- Left factoring
  - Organize the productions so that common portions at the left end are checked only once.
    - Left factoring
    - There is also right factoring - reduces the grammar, but has no effect on backtracking.
  - So grammar fragment above would become
    - $B \rightarrow D; [B \mid S]$

Top-Down Syntax Analysis

- Useful in the case of if statements
  - Single-command ::= $V$-name := Expression
  - If Expression then single-command
  - If Expression then single-command else single-command

Top-Down SA

- Backtracking may occur because of conflict between different non terminals.
  - If $B \Rightarrow^{*} \alpha$ and $D \Rightarrow^{*} \alpha$.
  - May still backtrack after discovery of alpha.
  - Began to discover a $D$ or an $S$ symbol?
- Want distinct way of choosing correct production to expand next.
  - Replace one of the non terminal with the RHS of its defining production (substitution of nonterminal symbols).

Grammar

- $S \rightarrow A \mid C$
- $A \rightarrow B \mid B + B$
- $B \rightarrow x \mid y$
- $C \rightarrow D \mid D + D$
- $D \rightarrow A \Rightarrow^{A} A$

First Factoring

- $S \rightarrow A \mid C$
- $A \rightarrow B \mid B + B$
- $B \rightarrow x \mid y$
- $C \rightarrow D \mid D + D$
- $D \rightarrow A \Rightarrow^{A} A$

First Replacement

- $S \rightarrow A \mid D \mid A \Rightarrow^{D} D$
- $A \rightarrow B \mid B + B$
- $B \rightarrow x \mid y$
- $C \rightarrow D \mid D + D$
- $D \rightarrow A \Rightarrow^{A} A$

Factored Grammar

- $S \rightarrow A \mid \Rightarrow^{A} A \mid (D \mid D) + D$
- $A \rightarrow B \mid B + B$
- $B \rightarrow x \mid y$
- $C \rightarrow D \mid D + D$
- $D \rightarrow A \Rightarrow^{A} A$

Final grammar will not backtrack when analyzing.
Top-Down SA

• removing left recursion
  – when factored, left recursion will still remain
    • productions where left hand symbol appears at the
      left end of one of the RHS alternatives
    • \( S \rightarrow abc \mid def \mid Srx \)
      – if analyser tries Srx first
    • infinite regression
      – if analyser tries \( abc \) or \( def \) first then
        backtracking may be necessary
    • grammar is not factored properly

• can easily convert to iterative grammar
  – expand left recursive production
    • eventually will include non recursive alternatives
  – production as non recursive alternatives
    followed by right side of recursive element -
    iterative form
    one step \( abc \mid def \mid Srx \)
    two steps \( abcrx \mid defrx \mid Srxrx \)
    three steps \( abcrx \mid defrx \mid Srxrxrx \)
    \( n+1 \) steps \( abcrx(\cdot) \mid defrx(\cdot) \mid Srx(\cdot)^n \)
    iterative production \( S \rightarrow [abc \mid def] \cdot (rx)^* \)

• could change to right recursive type
  – prefer left recursive tree walker
• converting LR productions to iterative eliminates
  direct left recursion
  – possible to have indirect left recursion
    • recursion involves more than one step in the derivation
    • remove in similar way, replacing non terminals with their
      definitions
    • when last one remains, direct recursion is removed as before

• one symbol look ahead and one track analysers
  – grammar fully factored and left recursion removed
    – one symbol look ahead (OSLA) is simplest form of non
      backtracking analyser
  – OSLA
    – analyser decides which path to follow by looking at the
      current lexical item
    – requires OSLA grammar

• OSLA condition stated as
  – if each alternative in every production in the grammar
    starts with a terminal symbol, and if the symbols which
    start competing alternatives are distinct, then the
    grammar is a OSLA grammar
• sample grammar
  \( S \rightarrow i:=E \mid \text{goto } E \mid \text{if } E \text{ then } S \text{ else } S \mid \text{begin } C \text{ } \text{end} \)
  – as soon as current line of analysis fails, analyser can
    produce error report

• error report
  – only indicates fact that current lexitem doesn’t
    match current production
  – unexpected symbol where another was expected
Recursive Descent Parser

- **Grammar**
  
  Sentence ::= Subject Verb Object.
  Subject ::= I | a Noun | the Noun
  Object ::= me | a Noun | the Noun
  Noun ::= cat | mat | rat
  Verb ::= like | is | see | sees

- Bottom-up and top-down are the basis of many parsing algorithms

Recursive Descent Parser

- A recursive descent parser for a grammar G consists of a group of methods \textit{parseN}
  - One for each nonterminal symbol N of G.
  - Task of each method is to parse a single N-phrase.
  - These parse methods cooperate to parse complete sentences.

- The methods are called by \textit{parseSentence} which delegates the parsing of individual parts based on its “knowledge” of how the sentence should be constructed.
- Create a Parser class to contain all the methods, and an instance variable \textit{currentTerminal}
  - Ranges over the terminal symbols of the input string.

```java
public class Parser {
    private TerminalSymbol currentTerminal;
    ...
    private void parseSentence() {
        parseSubject();
        parseVerb();
        parseObject();
        accept('.');
    }
    ...
}
```

- Using the production rule, the sentence consists of a subject, verb, object and . in that order.
Recursive Descent Parser

- parseSubject method checks the format of the subject (3 possibilities)
  - Decides which form by inspecting the currentTerminal.
    - | a Noun | the Noun
    - Decision based on first terminal input

Recursive Descent Parser

Private void parseSubject()
{
if (currentTerminal matches 'I')
  accept('I');
else if (currentTerminal matches 'a') {
  accept('a');
  parseNoun();
}
else if (currentTerminal matches 'the') {
  accept('the');
  parseNoun();
}
else
  report syntactic error
}

Recursive Descent Parser

Private void parseNoun()
{
if (currentTerminal matches 'cat')
  accept('cat');
else if (currentTerminal matches 'mat')
  accept('mat');
else if (currentTerminal matches 'rat')
  accept('rat');
else
  report syntactic error
}

Recursive Descent Parser

- In general, the methods will cooperate as follows;
  - currentTerminal will successively contain each input terminal and is available to all methods.
  - On entry to method parseN, currentTerminal is supposed to contain the first terminal of an N Phrase
  - On exit it contains the input terminal immediately following that n-phrase

Recursive Descent Parser

- Can be systematically developed from a suitable grammar by;
  - Express the grammar in EBNF
    - Single production rule for each nonterminal symbol and perform any necessary grammar transformations
      - left recursion, left factoring.
    - Transcribe each EBNF production rule N::= X to a parsing method parseN whose body is determined by X

Recursive Descent Parser

- On entry to method accept with argument t
  - currentTerminal is supposed to contain the terminal t.
    - On exit, currentTerminal contains the input terminal immediately following t.

- If the production rules are mutually recursive, then the parsing methods will be mutually recursive
  - hence the algorithm is recursive descent.
Recursive Descent Parser

- Make the parser consist of;
  - Private variable currentToken
  - Private parsing methods developed in step 2 above.
  - Private auxiliary methods accept and any others as required
  - Public parse method that calls parseS
    - Where S is the start symbol, having first called the scanner to store the first input token in currentToken.

Error Detection

- can introduce alternative which test for common errors
  - like D<error> in first grammar
  - compiler can produce messages just for those errors
    - provide more information for user
    - <statement> expected but not found type
  - using this, the following errors can be detected
    - declarations used in the middle of statement blocks
    - semicolon inserted between a statement and following else statement
    - too many begin or end symbols in the input

Error Detection

- can
  - abandon compilation
  - delete part of the program so error disappears
  - insert extra symbols into program so that error disappears
- when error encountered
  - compiler should note that program is faulty and not allow it to be executed
  - recovery is difficult
    - fixing or partially fixing may cause other errors

Error Reporting

- messages should always be in plain language and in terms of the source code
- try to quote the item by name
- indicate the position on the line where the error is