Top Down Analysis

Top down constriction of a parse tree:

- Start with the root, labelled by the starting symbol
- Repeatedly perform the following steps:
  1. At internal node N, labelled with non-terminal A
     - Select one of the production rules for A
     - Construct children at N for the symbols in the right part of this production rule
  2. Find the next node to construct a subtree
     - Typically the leftmost unexpanded non-terminal of the current tree

During the construction of the parse tree the current terminal of the input that is being scanned is called the lookahead symbol

Our aim during top-down parsing - to construct the parse tree, such that the string generated by the parse tree matches the input string

For a match to occur - the starting symbol stmt must derive a string that starts with for

When a node in the parse tree:

- Is labelled with a terminal
- Matches the lookahead symbol

Then

- The lookahead becomes the next terminal in the input
- We consider the next child in the parse tree

When a node in the parse tree is labelled with a non-terminal then:

- We repeat by selecting one of its production rules
- Special case: $A \rightarrow \epsilon$ - we choose it when nothing else can be used

In general:

- Many possibilities for a production at a non-terminal
- The selection of one of them may involve trial and error

A selected production is unsuitable if after using this production, we can't complete the tree to match the whole input string

If a selected production is unsuitable, backtrack and try another production until we

- either match the input string
- or we report error (input string not in the language)

1 Recursive descent parsing

- A top-down parsing method, using recursive procedures to process the input
- One procedure for each non-terminal of the grammar
- In general it requires backtracking until it finds the correct production rule
- Very easy and intuitive to write code for it

1.1 Predictive parsing

- Special case of recursive-descent parsing
- It uniquely determines the steps of each procedure, therefore no backtracking is required
- Runs in linear time
- Not all grammars can be parsed by predictive parsing

LL(k) grammars:

- A recursive de4cent parser can uniquely determine the next production rule, just by looking to the next k tokens of the input
- Subclass of context free grammars

Predictive parsing is only possible for LL(k) grammars LL(k) grammars exclude:

- All ambiguous grammars
- All grammars containing left recursion

1.2 Elimination of left recursion

A grammar is left recursive if for some non-terminal A and some string $\alpha : A \xrightarrow{+} A\alpha$

Immediate left recursion:: $A \rightarrow A\alpha|\beta$
This can be rewritten as

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A'|\epsilon$$

Left recursion in two or more steps:

$$S \rightarrow A\alpha|\beta$$
$$A \rightarrow Sd|c|\epsilon$$

In this case it can occur that $S \rightarrow A\alpha \rightarrow Sd\alpha$

A left recursive grammar can lead a recursive-descent parser into an infinite loop, e.g.

$$A \rightarrow A\alpha|\beta$$

Expanding A using the production rule $A \rightarrow A\alpha$ will lead to backtracking forever

1.3 Algorithm for eliminating left recursion

Take the following production with **immediate** left recursion

$$A \rightarrow A\alpha_1|A\alpha_2|...|A\alpha_m|\beta_1|\beta_2|...|\beta_n$$

where no β_i begins with A

This can then be rewritten as:

$$A \rightarrow \beta_1 A'|\beta_2 A'|...|\beta_n A'$$
$$A' \rightarrow \alpha_1 A'|\alpha_2 A'|...|\alpha_m A'|\epsilon$$

```
1)  arrange the nonterminals in some order A_1, A_2, ..., A_n.
2)  for ( each i from 1 to n ) {
3)      for ( each j from 1 to i - 1 ) {
4)          replace each production of the form A_i → A_j γ by the
              productions A_i → δ_1 γ | δ_2 γ | ··· | δ_k γ, where
              A_j → δ_1 | δ_2 | ··· | δ_k are all current A_j-productions
5)      }
6)      eliminate the immediate left recursion among the A_i-productions
7)  }
```

2 Left factoring

Many times:

- Two alternative production rules have the same prefix
- The choice between them is not clear

Example: $A \rightarrow \alpha\beta_1|\alpha\beta_2$

- Suppose the input begins with a string derived from α
- Which transition to choose?
- We need to read more symbols from the input - this grammar is not an LL(1) grammar

We can turn this example into an LL(1) grammar in the following way

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1|\beta_2$$

2.1 Algorithm for left factoring

For each non-terminal A

- Find the longest prefix α common to two or more of its alternative productions

$$A \rightarrow \alpha\beta_1|\alpha\beta_2|...|\alpha\beta_n|\gamma$$

- Replace these productions by the following

$$A \rightarrow \alpha A'|\gamma$$
$$A' \rightarrow \beta_1|\beta_2|...|\beta_n$$

Repeat until no two alternative productions have a common prefix

3 LL(k) grammars

When we consider a non-terminal:

- How do we know which production rule to use?
- The predictive parser looks ahead to the rest of the input and its decision is forced

LL(k) grammar:

- L: left-to-right scan of the input tokens
- L: leftmost derivation of the input string
- k: look k tokens ahead in the input string

We must first:

- Eliminate ambiguities
- Eliminate left recursions
- Perform left factoring

3.1 LL(1) grammars

Interesting case:

- LL(1) grammars - read only one token ahead in the input
- More efficient parsing than other LL(k) grammars

How can we uniquely determine the next production in an LL(1) grammar?

We do these with the help of two functions (sets)

- FIRST(α), where α is any string of grammar symbols
- FOLLOW(A), where A is any non-terminal

And a predictive parsing table that associates terminals (of the grammar) with non-terminals (of the input)

3.1.1 FIRST

The set FIRST(α), for a string α, contains:

- Every first terminal of a string derived from α
- i.e. if $\alpha \overset{*}{\Rightarrow} c\gamma$, then $c \in FIRST(\alpha)$
- also, if $\alpha \overset{*}{\Rightarrow} \epsilon$, then $\epsilon \in FIRST(\alpha)$

Example usage:

- Consider the two productions $A \to \gamma_1$ and $A \to \gamma_2$
- Suppose that FIRST(γ_1) and FIRST(γ_2) are disjoint
- let the next input symbol be a
- if a belongs to FIRST(γ_1), then choose $A \to \gamma_1$
- if a belongs to FIRST(γ_2), then choose $A \to \gamma_2$
- Otherwise report error

3.1.2 Follow

The set FOLLOW(A), for a non-terminal contains:

- Every terminal a that can appear immediately to the right of A in some string derived from S
- i.e. there exists a derivation $S \overset{*}{\Rightarrow} \alpha A a \beta$ where α and β are two strings

In addition if A can be the rightmost symbol in some string derived from S, then \$ is in FOLLOW(A) (where \$ is the end of file symbol)

4 LL(1) Grammars

Definition: LL(1) Grammar

A Grammar G is an LL(1) grammar iff for every two productions $A \to \alpha|\beta$:

1. There is no terminal a such that both α and β derive a string starting with a
2. At most one of α and β can derive the empty string ϵ
3. If $\beta \overset{*}{\Rightarrow} \epsilon$ then α does not derive any string beginning with a terminal in FOLLOW(A) and vice versa

Therefore an efficient algorithm to determine whether a grammar G is LL(1) grammar by computing all needed sets FIRST() and FOLLOW()

A language L is called an LL(1) language is L can be generated by an LL(1) grammar

If any language L is given by a non-LL(1) grammar L may still be an LL(1) language by a different grammar

5 Non-recursive predictive parsing

We saw predictive parsing via recursive calls of procedures

We can ”mimic” leftmost derivation:

- Explicitly maintain a stack
- ”table driven predictive parsing”

If w is the input that has been matched so far, then the stack holds a sequence α of grammar symbols (terminals and non-terminals), such that

$$S \stackrel{*}{\Rightarrow} w\alpha$$

where S is the start symbol of the grammar

Given a parsing table M and input w:

- Initialise a stack containing S (with bottom symbol $)
- Repeat until the stack contains only $:
  - Let the next input character be c
  - If the top of the stack is a terminal t, then:
    * If c and t don’t match, report an error
    * Otherwise match c and t - consume c from input and pop t from stack
  - If the top of the stack is a non-terminal A, then:
    * If $M[A, c]$ is undefined, report an error
    * Otherwise replace the top of the stack with $M[A, c]$