LL(1) PARSING

A **top-down** parsing algorithm parses an input string of tokens by tracing out the steps in a leftmost derivation. Such algorithms are called top-down since the implied traversal of the parse tree occurs from the root to the leaves.

Top-down parsers come in two forms: **bactracking parsers** and **predictive parsers**.

A predictive parser attempts to predict the next construction using one or more lookahead tokens.

Two well-known top-down parsing algorithms are called **recursive-descent parsing** and **LL(1) parsing**.

Recursive descent parsing is the most suitable method for a handwritten parser.

The first "L" in LL(1) refers to the fact that it processes the input from left to right.

The second "L" refers to the fact that it traces out a leftmost derivarion for the input string.

The "1" in parentheses means that it uses only one symbol of input to predict the direction of the parse.

We start by considering the following grammar that generates strings of balanced parenthesis:
$S \rightarrow (\ S \ ) \ S \mid \varepsilon$

We will assume that \$ marks the bottom of the stack and the end of the input.

Parsing action of an LL(1) parser:

	Parsing Stack	Input	Action
1	$\$S$	()\$	$S \rightarrow (S)S$
2	$\$S)S($	()\$	match
3	$\$S)S$	)\$	$S \rightarrow \varepsilon$
4	$\$S)$	)\$	match
5	$\$S$	\$	$S \rightarrow \varepsilon$
6	\$	\$	accept

LL(1) parsing table for the grammar

$S \rightarrow (\ S \ ) \ S \mid \varepsilon$

$M[N,T]$	(	)	\$
S	$S \rightarrow (S)S$	$S \rightarrow \varepsilon$	$S \rightarrow \varepsilon$

We denote the table by $M[N, T]$ and N denotes nonterminals and T terminals.

We use this table to decide which decision should be made if a given nonterminal N is at the top of the parsing stack, based on the current input symbol T.

We add production choices to the LL(1) parsing table as follows:

1. If $A \to \alpha$ is a production choice, and there is a derivation $\alpha \Rightarrow^* a\beta$, where a is a token, then we add $A \to \alpha$ to the table entry $M[A, a]$.

2. If $A \to \alpha$ is a production choice, and there are derivations $\alpha \Rightarrow^* \varepsilon$ and $S\$ \Rightarrow^* \beta A a \gamma$, where S is the start symbol and a is a token (or \$), then we add $A \to \alpha$ to the table entry $M[A, a]$.

The idea behind these rules are as follows: In rule 1, given a token a in the input, we wish to select a rule $A \to \alpha$ if α can produce an a for matching.

In rule 2, if A derives the empty string (via $A \to \alpha$), and if a is a token that can legally come after A in a derivation, then we want to select $A \to \alpha$ to make A disappear.

These rules are difficult to implement directly, so we will develop algorithms involving **first** and **follow sets** (concepts that will be defined later) in order to implement these rules.

Definition

A grammar is an **LL(1) grammar** if the associated LL(1) parsing table has at most one production in each table entry.

First Sets

In this lecture we show how to use **First** and **Follow Sets** to construct the LL(1) parsing table $M[N, T]$ for a CFG.

Before we give the precise definitions of First and Follow Sets, we show how to use it in the construction of LL(1) parsing tables.

The LL(1) parsing table $M[N, T]$ is constructed as follows:

Repeat the following two steps for each non-terminal A and each production $A \rightarrow \alpha$:

1. For each token a in First(α), add $A \rightarrow \alpha$ to the entry $M[A, a]$.
2. If ε is in First(α), for each element a of Follow(A) (where a is a token or a is \$), add $A \rightarrow \alpha$ to $M[A, a]$.

Now consider the grammar
$S \rightarrow (\ S \ ) \ S \mid \varepsilon$

In this case we have that
First($(S)S$) = { (}
First(ε) = $\{\varepsilon\}$
Follow(S) = {), \$ }.

Thus according to the above procedure we get the following LL(1) parsing table:

$M[N,T]$	(	)	\$
S	$S \rightarrow (S)S$	$S \rightarrow \varepsilon$	$S \rightarrow \varepsilon$

First Sets

Definition:
Let X be a grammar symbol (a terminal or nonterminal) or ε. Then the set **First(X)** consisting of terminals, and possibly ε, is defined as follows:

1. If X is a terminal or ε, First(X) $= \{X\}$.
2. If X is a n nonterminal, then for each production choice $X \rightarrow X_1 X_2 ... X_n$, First($X$) contains First($X_1$)$-\{\varepsilon\}$. If also for some $i < n$, all the sets First(X_1),...,First(X_i) contains ε, then First(X) contains First(X_{i+1})$-\{\varepsilon\}$. If all the sets First(X_1),...,First(X_n) contains ε, then First(X) also contains ε.

We now define **First(α)** for any string $\alpha = X_1X_2...X_n$ (a string of terminals and nonterminals) as follows:

First(α) contains First(X_1)$-\{\varepsilon\}$.

For each $i = 2, ..., n$, if First(X_k) contains ε for all $k = 1, ..., i-1$, then First(α) contains First(X_i)$-\{\varepsilon\}$.

Finally, if for all $i = 1, ..., n$, First(X_i) contains ε, then First(α) contains ε.

Algorithm for computing First(A) for all nonterminals A

for all nonterminals A **do** First(A):=$\{\}$;
while there are changes to any First(A) **do**
  for each production choice $A \rightarrow X_1...X_n$ **do**
    $k := 1$; Continue:=true;
    while Continue = true **and** $k <= n$ **do**
      add First(X_k)$-\{\varepsilon\}$ to First(A);
      if ε not in First(X_k) **then** Continue:=false;
      $k := k + 1$;
    if Continue = true **then** add ε to First(A);

Simplified algorithm for First Sets in the absence of ε-production

for all nonterminals A **do** First(A):=$\{\}$;
while there are changes to any First(A) **do**
  for each production choice $A \rightarrow X_1...X_n$ **do**
    add First(X_1) to First(A);

Example

Consider the simple integer expression grammar:

$exp \rightarrow exp\ addop\ term \mid term$
$addop \rightarrow + \mid -$
$term \rightarrow term\ mulop\ factor \mid factor$
$mulop \rightarrow *$
$factor \rightarrow (\ exp\ ) \mid \mathbf{number}$

Computation of First sets for nonterminals in the grammar:

Grammar rule	Pass 1	Pass 2	Pass 3
$exp \to exp$ $addop\ term$			
$exp \to term$			First(exp) $= \{(, \mathbf{number}\}$
$addop \to +$	First($addop$) $= \{+\}$		
$addop \to -$	First($addop$) $= \{+, -\}$		
$term \to term$ $mulop\ factor$			
$term \to factor$		First($term$)$=$ $\{(, \mathbf{number}\}$	
$mulop \to *$	First($mulop$) $= \{*\}$		
$factor \to$ $(\ exp\ )$	First($factor$) $= \{\ (\ \}$		
$factor \to$ $\mathbf{number}$	First($factor$) $= \{(, \mathbf{number}\}$		

Thus

First(exp) $= \{(, \mathbf{number}\}$
First($term$) $= \{(, \mathbf{number}\}$
First($factor$) $= \{(, \mathbf{number}\}$
First($addop$) $= \{+, -\}$
First($mulop$) $= \{*\}$

First Sets and Follow Sets

We begin with another example of calculating First Sets for Nonterminals of a CFG.

Example

$statement \to if\text{-}stmt \mid \mathbf{other}$
$if\text{-}stmt \to \mathbf{if}\ (\ exp\ )\ statement\ else\text{-}part$
$else\text{-}part \to \mathbf{else}\ statement \mid \varepsilon$
$exp \to 0 \mid 1$

Computation of First sets for nonterminals in the grammar:

Grammar rule	Pass 1	Pass 2
$statement \to if\text{-}stmt$		First($statement$)$=$ $\{\mathbf{if}, \mathbf{other}\}$
$statement \to \mathbf{other}$	First($statement$) $=$ $\{\mathbf{other}\}$	
$if\text{-}stmt \to \mathbf{if}(\ exp\ )$ $statement\ else\text{-}part$	First($if\text{-}stmt$)$=$ $= \{\mathbf{if}\}$	
$else\text{-}part \to \mathbf{else}$ $statement$	First($else\text{-}part$)$=$ $= \{\mathbf{else}\}$	
$else\text{-}part \to \varepsilon$	First($else\text{-}part$)$=$ $= \{\mathbf{else}, \varepsilon\}$	
$exp \to 0$	First(exp)$=\{0\}$	
$exp \to 1$	First(exp)$=\{0, 1\}$	

Thus

First($statement$) $= \{\mathbf{if}, \mathbf{other}\}$
First($if\text{-}stmt$) $= \{\mathbf{if}\}$
First($else\text{-}part$) $= \{\mathbf{else}, \varepsilon\}$
First(exp) $= \{0, 1\}$

Follow Sets

Given a nonterminal A, the follow set **Follow(A)**, consisting of terminals, and possibly \$, is defined as follows:

1. If A is a start symbol, then \$ is in Follow(A).
2. If there is a production $B \to \alpha A \gamma$, then First(γ)$-\{\varepsilon\}$ is in Follow(A).
3. If there is a production $B \to \alpha A \gamma$ such that ε is in First(γ), then Follow(A) contains Follow(B).

Algorithm for the computation of Follow Sets:

Follow(start-symbol):= {\$};
for all nonterminals $A \neq$ start-symbol **do** Follow(A):={};
while there are changes to any Follow sets **do**
    for each production $A \to X_1 ... X_n$ **do**
        for each X_i that is a nonterminal **do**
            add First($X_{i+1} ... X_n$)$-\{\varepsilon\}$ to Follow(X_i)
            (* Note: if $i = n$, then $X_{i+1} ... X_n = \varepsilon$ *)
            if ε is in First($X_{i+1} ... X_n$) **then**
                add Follow(A) to Follow(X_i)

Example

We consider again the grammar

$exp \to exp\ addop\ term \mid term$
$addop \to + \mid -$
$term \to term\ mulop\ factor \mid factor$
$mulop \to *$
$factor \to (\ exp\ ) \mid \mathbf{number}$

Recall that

First(exp) $= \{(, \mathbf{number}\}$
First($term$) $= \{(, \mathbf{number}\}$
First($factor$) $= \{(, \mathbf{number}\}$
First($addop$) $= \{+, -\}$
First($mulop$) $= \{*\}$

Computation of the Follow sets for the grammar:
We omit the four grammar rule choices that have no possibility of affecting the computation.

Grammar rule	Pass 1	Pass 2
$exp \to exp$ $addop$ $term$	Follow(exp)= $\{\$, +, -\}$ Follow($addop$)= $\{(, \textbf{number}\}$ Follow($term$)= $\{\$, +, -\}$	Follow($term$)= $\{\$, +, -, *,)\}$
$exp \to term$		
$term \to term$ $mulop$ $factor$	Follow($term$)= $\{\$, +, -, *\}$ Follow($mulop$)= $\{(, \textbf{number}\}$ Follow($factor$) = $\{\$, +, -, *\}$	Follow($factor$)= $\{\$, +, -, *,)\}$
$term \to factor$		
$factor \to$ (exp)	Follow(exp) = $\{ \$, +, -,)\}$	

Thus

Follow(exp) $= \{\$, +, -,)\}$
Follow($term$) $= \{\$, +, -, *,)\}$
Follow($factor$) $= \{\$, +, -, *,)\}$
Follow($addop$) $= \{(, \textbf{number}\}$
Follow($mulop$) $= \{(, \textbf{number}\}$

Example

$statement \to if\text{-}stmt \mid \textbf{other}$
$if\text{-}stmt \to \textbf{if}$ (exp) $statement$ $else\text{-}part$
$else\text{-}part \to \textbf{else}$ $statement \mid \varepsilon$
$exp \to 0 \mid 1$

Recall that

First($statement$) $= \{\textbf{if}, \textbf{other}\}$
First($if\text{-}stmt$) $= \{\textbf{if}\}$
First($else\text{-}part$) $= \{\textbf{else}, \varepsilon\}$
First(exp) $= \{0, 1\}$

A calculation as in the previous example shows that

Follow($statement$) $= \{\$, \textbf{else}\}$
Follow($if\text{-}stmt$) $= \{\$, \textbf{else}\}$
Follow($else\text{-}part$) $= \{\$, \textbf{else}\}$
Follow(exp) $= \{\ )\ \}$

Recall that the LL(1) parsing table $M[N,T]$ is constructed as follows:

Repeat the following two steps for each non-terminal A and each production $A \to \alpha$:

1. For each token a in First(α), add $A \to \alpha$ to the entry $M[A,a]$.

2. If ε is in First(α), for each element a of Follow(A) (where a is a token or a is \$), add $A \to \alpha$ to $M[A,a]$.

Using the procedure on the previous slide, we obtain the following table.

$M[N,T]$	**if**	**other**	**else**	0	1	\$
statement	*statement* $\to$ *if-stmt*	*statement* $\to$ **other**				
if-stmt	*if-stmt* $\to$ **if** (*exp*) *statement* *else-part*					
else-part			*else-part* $\to$ **else** *statement* *else-part* $\to \varepsilon$			*else-part* $\to \varepsilon$
exp				*exp* $\to$ 0	*exp* $\to$ 1	

We notice, that as expected, this grammar is not LL(1), since the entry M[*else-part*,**else**] contains two entries, corresponding to the dangling else ambiguity.

We could apply the disambiguating rule that would always prefer the rule that generates the current lookahead token over any other (this corresponds to the most closely nested disambiguating rule), and thus the production
else-part $\to$ **else** *statement*

We now show the LL(1) parsing actions for the string
if (0) **if**(1) **other else other**

We use the following abbreviations:
statement $= S$
if-stmt $= I$
else-part $= L$
exp $= E$
if $=$ **i**
else $=$ **e**
other $=$ **o**

Parsing stack	Input	Action
$\$S$	**i** (0) **i** (1) **o e o**\$	$S \to I$
$\$I$	**i** (0) **i** (1) **o e o**\$	$I \to$ **i** (E) S L
$\$LS)E($**i**	**i** (0) **i** (1) **o e o**\$	match
$\$LS)E($	(0) **i** (1) **o e o**\$	match
$\$LS)E$	0) **i** (1) **o e o**\$	$E \to 0$
$\$LS)0$	0) **i** (1) **o e o**\$	match
$\$LS)$	) **i** (1) **o e o**\$	match
$\$LS$	**i** (1) **o e o**\$	$S \to I$
$\$LI$	**i** (1) **o e o**\$	$I \to$ **i** (E) S L
$\$LLS)E($**i**	**i** (1) **o e o**\$	match
$\$LLS)E($	(1) **o e o**\$	match
$\$LLS)E$	1) **o e o**\$	$E \to 1$
$\$LLS)1$	1) **o e o**\$	match
$\$LLS)$	) **o e o**\$	match
$\$LLS$	**o e o**\$	$S \to$ **o**
$\$LL$**o**	**o e o**\$	match
$\$LL$	**e o**\$	$L \to$ **e** S
$\$LS$**e**	**e o**\$	match
$\$LS$	**o**\$	$S \to$ **o**
$\$L$**o**	**o**\$	match
$\$L$	\$	$L \to \varepsilon$
\$	\$	accept

Left Recursion Removal and Left Factoring

In this lecture we discuss techniques (that sometimes work) to convert a grammar that is not LL(1) into an equivalent grammar that is LL(1).

Consider the following grammar:

$exp \to exp\ addop\ term \mid term$
$addop \to + \mid -$
$term \to term\ mulop\ factor \mid factor$
$mulop \to *$
$factor \to (\ exp\ ) \mid \textbf{number}$

This grammar is not LL(1) since **number** is in First(exp) and in First($term$).

Thus in the entry $M[exp,\ \textbf{number}]$ in the $LL(1)$ parsing table we will have the entries $exp \to exp\ addop\ term$ and $exp \to term$

The problem is the presence of the left recursive rule $exp \to exp\ addop\ term \mid term$.

Thus in order to try to convert this grammar into an LL(1) grammar, we will remove the left recursion from this grammar.

Left Recursion and Right Recursion

Before we look at the technique of removing left recursion from a grammar, we first discuss left recursion in general.

Grammar rules in BNF provide for concatenation and choice but no specific operation equivalent to the $*$ of regular expressions are provided.

We can obtain repetition by using for example rules of the form
$A \to Aa \mid a$
or $A \to aA \mid a$

Both these grammars generate $\{a^n | n \geq 1\}$.

We call the rule $A \to Aa \mid a$ **left recursive** and $A \to aA \mid a$ **right recursive.**

In general, rules of the form
$A \to A\alpha \mid \beta$ are called left recursive and rules of the form
$A \to \alpha A \mid \beta$ right recursive.

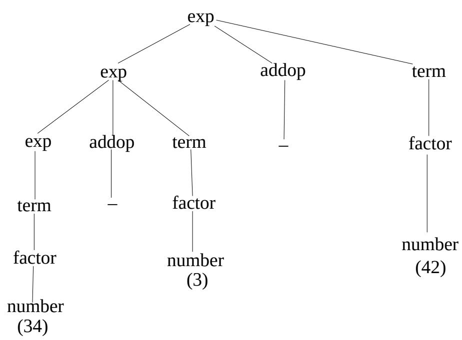
Grammars equivalent to the regular expression a^* are given by
$A \to Aa \mid \varepsilon$
or $A \to aA \mid \varepsilon$

Notice that a left recursive rules make expressions associate on the left.

The parse tree for the expression $34 - 3 - 42$ in the grammar

$exp \rightarrow exp\ addop\ term \mid term$
$addop \rightarrow + \mid -$
$term \rightarrow term\ mulop\ factor \mid factor$
$mulop \rightarrow *$
$factor \rightarrow (\ exp\ ) \mid \textbf{number}$

is for example given by



In EBNF, the left recursive rule

$$exp \rightarrow exp\ addop\ term$$

is written as

$$exp \rightarrow term\ \{\ addop\ term\ \}.$$

In EBNF, in right recursive form, this rule is written as

$exp \rightarrow term\ [\ addop\ exp\ ]$

thus the $addop\ term$ part is considered as an optional construct.

Left Recursion Removal and Left Factoring

In the rule

$$exp \rightarrow exp\ +\ term \mid exp\ -\ term \mid term$$

we have **immediate left recursion** and in

$A \rightarrow B\ a \mid A\ a \mid c$
$B \rightarrow B\ b \mid A\ b \mid d$

we have **indirect left recursion**.

We only consider how to remove immediate left recusion.

Consider again the rule

$exp \rightarrow exp\ addop\ term \mid term$

We rewrite this rule as

$exp \rightarrow term\ exp'$
$exp' \rightarrow addop\ term\ exp' \mid \varepsilon$

to remove the left recursion.

In general if we have productions of the form

$$A \rightarrow A\ \alpha_1 \mid \ldots \mid A\ \alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

we rewrite this as

$A \rightarrow \beta_1\ A' \mid \ldots \mid \beta_m A'$
$A' \rightarrow \alpha_1\ A' \mid \ldots \mid \alpha_n\ A' \mid \varepsilon$

in order to remove the left recursion.

Example

If we remove the left recursion from the rule
$exp \rightarrow exp\ +\ term \mid exp\ -\ term \mid term$
we obtain
$exp \rightarrow term\ exp'$
$exp' \rightarrow +\ term\ exp' \mid -\ term\ exp' \mid \varepsilon$

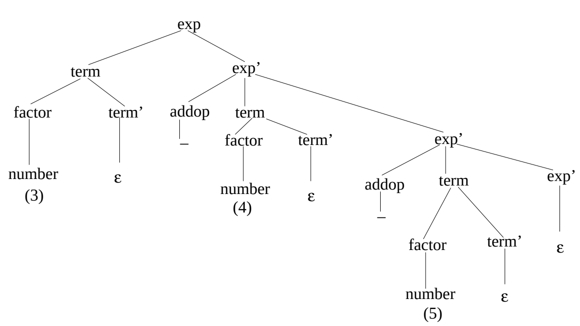
Example

If we remove left recursion from the grammar $exp \rightarrow exp\ addop\ term \mid term$
$addop \rightarrow + \mid -$
$term \rightarrow term\ mulop\ factor \mid factor$
$mulop \rightarrow *$
$factor \rightarrow (\ exp\ ) \mid \mathbf{number}$

we obtain the grammar

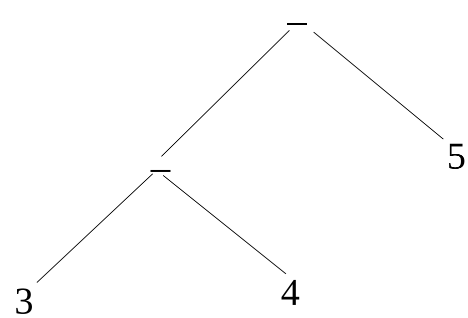
$exp \rightarrow term\ exp'$
$exp' \rightarrow addop\ term\ exp' \mid \varepsilon$
$addop \rightarrow + \mid -$
$term \rightarrow factor\ term'$
$term' \rightarrow mulop\ factor\ term' \mid \varepsilon$
$mulop \rightarrow *$
$factor \rightarrow (\ exp\ ) \mid \mathbf{number}$

Now consider the parse tree for 3 – 4 – 5



This tree no longer expresses the left associativity of subtraction.

Nevertheless, a parser should still construct the appropriate left associative syntax tree. We obtain the syntax tree by removing all the unneccessary information from the parse tree. A parser will usually construct a syntax tree and not a parse tree.



Left Factoring

Left factoring is required when two or more grammar rule choices share a common prefix string, as in the rule
$A \to \alpha\ \beta \mid \alpha\ \gamma$

Obviously, an LL(1) parser cannot distinguish between the production choices in such a situation.

In the following example we have exactly this problem:

$$\begin{aligned} if\text{-}stmt &\to \mathbf{if}\ (\ exp\ )\ statement \\ &\mid \mathbf{if}\ (\ exp\ )\ statement\ \mathbf{else}\ statement \end{aligned}$$

Algorithm for left factoring a grammar:

while there are changes to the grammar **do**
    for each nonterminal A **do**
        let α be a prefix of maximal length that is shared
            by two or more production choices for A
    if $\alpha \neq \varepsilon$ **then**
        let $A \to \alpha_1 \mid ... \mid \alpha_n$ be all the production choices for A
            and suppose that $\alpha_1, ..., \alpha_k$ share α, so that
            $A \to \alpha\beta_1 \mid ... \mid \alpha\beta_k \mid \alpha_{k+1} \mid ... \mid \alpha_n$, the β_j's share
            no common prefix, and $\alpha_{k+1}, ..., \alpha_n$ do not share α.
        replace the rule $A \to \alpha_1 \mid ... \mid \alpha_n$ by the rules:
            $A \to \alpha A' \mid \alpha_{k+1} \mid ... \mid \alpha_n$
            $A' \to \beta_1 \mid ... \mid \beta_k$

Example

Consider the following grammar of if-statements:

$$\begin{aligned} if\text{-}stmt &\to \mathbf{if}\ (\ exp\ )\ statement \\ &\mid \mathbf{if}\ (\ exp\ )\ statement\ \mathbf{else}\ statement \end{aligned}$$

The left factored form of this grammar is

$$\begin{aligned} &if\text{-}stmt \to \mathbf{if}\ (\ exp\ )\ statement\ else\text{-}part \\ &else\text{-}part \to \mathbf{else}\ statement \mid \varepsilon \end{aligned}$$

Example

Here is a typical example where a programming language fails to be LL(1):

$$\begin{aligned} &statement \to assign\text{-}stmt \mid call\text{-}stmt \mid \mathbf{other} \\ &assign\text{-}stmt \to \mathbf{identifier}\ := exp \\ &call\text{-}stmt \to \mathbf{identifier}\ (\ exp\text{-}list\ ) \end{aligned}$$

This grammar is not in a form that can be left factored. We must first replace *assign-stmt* and *call-stmt* by the right-hand sides of their defining productions:

$$\begin{aligned} statement &\to \mathbf{identifier}\ := exp \\ &\mid \mathbf{identifier}\ (\ exp\text{-}list\ ) \\ &\mid \mathbf{other} \end{aligned}$$

Then we left factor to obtain:

$$
\begin{aligned}
&\textit{statement} \rightarrow \textbf{identifier}\ \textit{statement}' \\
&\qquad\qquad \mid \textbf{other} \\
&\textit{statement}' \rightarrow \,:=\, \textit{exp} \mid (\ \textit{exp-list}\ )
\end{aligned}
$$

Note how this obscures the semantics of call and assignment by separating the identifier from the actual call or assign action.

General Remarks

There are of course many more top-down parsing methods than just recursive-descent and LL(1).

Examples of top-down parser generators:

- Antlr generates a recursive-descent parser from an EBNF description. It is part of the Purdue Compiler Construction Tool Set.

- LLGen is a parser generotor for LL(1) grammars.