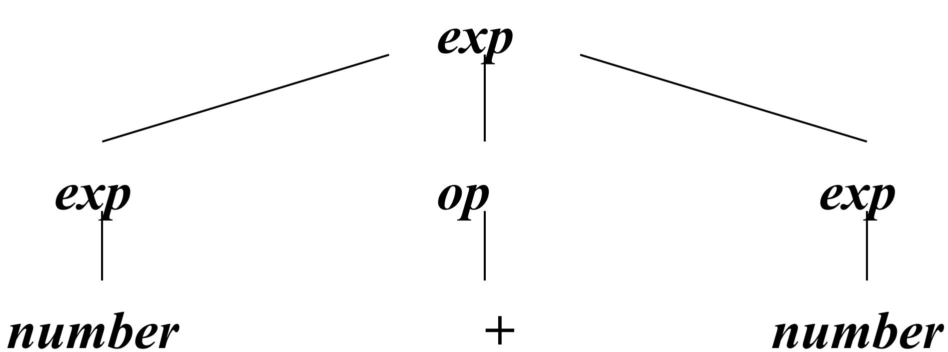
COMPILER CONSTRUCTION

Concept of Top-Down Parsing(1)

- It parses an input string of tokens by ***tracing out the steps in a leftmost derivation***.

- The example:
  - number + number,   and corresponds to the parse tree



Concept of Top-Down Parsing(2)

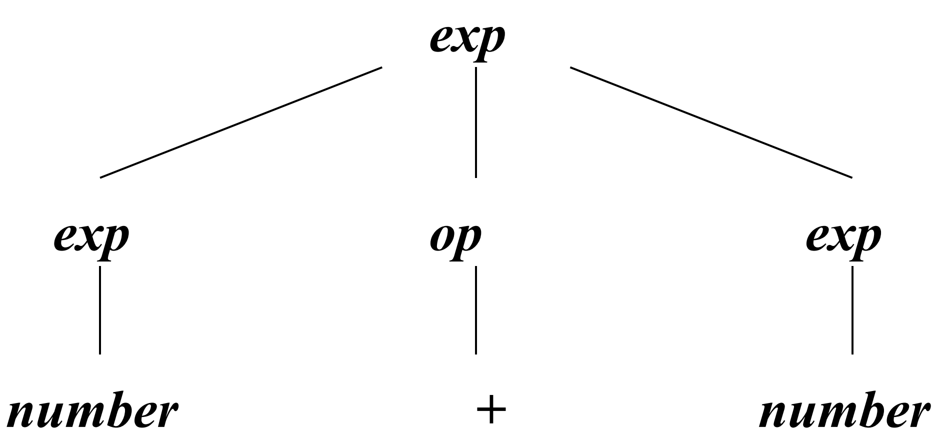
The above parse tree corresponds to the leftmost derivations:

(1) *exp* => *exp op exp*

(2) => ***number*** *op exp*

(3) => ***number*** + *exp*

(4) => ***number*** + ***number***



Two kinds of Top-Down parsing algorithms

- ***Recursive-descent parsing***:
  - is quite versatile and suitable for a handwritten parser.
- ***LL*(1) *parsing***:
  - The first "*L*" 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 derivation** for the input string;
  - The number "1" means that it uses only one symbol of input to predict the direction of the parse.

Contents

4. Top-Down Parsing

PART ONE

4.1 Top-Down Parsing by Recursive-Descent

4.1.1 The Basic Method of Recursive-Descent

The idea of Recursive-Descent Parsing

- View the grammar rule for a non-terminal *A* as a **definition for a procedure to recognize *A***

- The Expression Grammar:

  exp → *exp addop term* | *term*

  addop → + | **-**

  term → *term mulop factor* | *factor*

  mulop →*****

  factor → **(***exp***)** | **number**

A recursive-descent procedure that recognizes a *factor*

procedure *factor*
begin
  case *token* of
  (: *match*(**(**);
    exp;
    match(**)**);
  number:
    match (**number**);
  else *error*;
  end case;
end *factor*

- The *token* keeps the next token in the input (one symbol of look-ahead)
- The *match* procedure matches the next token with its parameter, advances the input if it succeeds, and declares error if it does not

Rule: *factor* → **(**exp**)** | **number**

Match Procedure

```
procedure match( expectedToken);
begin
  if token = expectedToken then
    getToken;
  else
    error;
  end if;
end match
```

Requiring the Use of EBNF

- The corresponding EBNF is

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

  $addop \rightarrow$ + | -

  $term \rightarrow factor$ { $mulop$ $factor$ }

  $mulop \rightarrow$ *

  $factor \rightarrow$ (exp) | **number**

- Write recursive-decent procedures for the remaining rules in the expression grammar is not as easy for *factor* (left recursion causes infinite loop)

4.1.2 Repetition and Choice: Using EBNF

An Example

- The grammar rule for an if-statement:

  If-stmt → **if (** *exp* **)** *statement*
  | **if (** *exp* **)** *statement* **else** *statement*

- Could not immediately distinguish the two choices because both start with the token **if**
- Put off the decision until we see the token **else** in the input

The EBNF of the if-statement

- *If-stmt* → **if (** *exp* **)** *statement* [**else** *statement*]

Square brackets of the EBNF are translated into a test in the code for *ifstmt*.

```
if token = else then
  match (else);
  statement;
endif;
```

- Notes
  - EBNF notation is designed to **mirror closely** the actual code of a recursive-descent parser
  - So a grammar should always be **translated into EBNF** if recursive-descent is to be used.

```
procedure ifstmt;
    begin
      match( if );
      match( ( );
      exp;
      match( ) );
      statement;
      if token = else then
        match (else);
        statement;
      end if;
    end ifstmt;
```

EBNF for Simple Arithmetic Grammar(1)

- The EBNF rule for $exp \rightarrow exp\ addop\ term \mid term$

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

The curly bracket expressing repetition can be **translated into a loop** in the code for *exp*:

procedure *exp*;
begin
  term;
  while *token* = + or *token* = - **do**
    match(*token*);
    term;
  end while;
end *exp*;

EBNF for Simple Arithmetic Grammar(2)

- The EBNF rule for *term*:
  term → *factor* {*mulop factor*}

Becomes the code:

```
procedure term;
begin
   factor;
   while token = * do
    match(token);
    factor;
   end while;
end term;
```

4.1.3 Further Decision Problems

More formal methods to deal with complex situation

(1) It may be difficult to convert a grammar in BNF into EBNF form;

(2) It is difficult to decide when to use the choice $A \rightarrow \alpha$ and the choice $A \rightarrow \beta$;

if both α and β begin with non-terminals. Such a decision problem requires the computation of the **First Set**.

More formal methods to deal with complex situation

(3) It may be necessary to know what token legally coming from the non-terminal *A*, **in writing the code for an ε-production: $A \rightarrow \varepsilon$.** Such tokens indicate *A* may disappear at this point in the parse. This set is called the **Follow Set** of *A*.

(4) It requires computing the First and Follow sets in order to **detect the errors as early as possible**. Such as ")3-2)", the parse will descend from *exp* to *term* to *factor* before an error is reported.

4.2 LL(1) Parsing

4.2.1 The Basic Method of LL(1) Parsing

Main idea

- LL(1) Parsing uses an **explicit stack** rather than recursive calls to perform a parse
- An example:
  - a simple grammar for the strings of balanced parentheses:

    $S \rightarrow \mathbf{(}S\mathbf{)}S \mid \varepsilon$
- The following table shows the actions of a top-down parser given this grammar and the string "()"

Table of Actions

Steps	Parsing Stack	Input	Action
1	\$S	( ) \$	S→(S) S
2	\$S)S(	( ) \$	match
3	\$S)S	)\$	S→ ε
4	\$S)	)\$	match
5	\$S	\$	S→ ε
6	\$	\$	accept

General Schematic

- A top-down parser begins by **pushing the start symbol** onto the stack
- It accepts an input string if, after a series of actions, the stack and the input become empty
- A general schematic for a successful top-down parse:

\$ StartSymbol	Inputstring\$	
…	…	//one of the two actions
…	…	//one of the two actions
\$	\$	accept

Two Actions

- **The two actions**
  - **Generate**: Replace a non-terminal *A* at the top of the stack by a string α (**in reverse**) using a grammar rule *A* → α
  - **Match**: Match a token on the top of the stack with the current input token
- The list of generating actions in the above table:

$$\begin{array}{ll} S \Rightarrow (S)S & [S \rightarrow (S)\ S] \\ \Rightarrow ()S & [S \rightarrow \varepsilon] \\ \Rightarrow () & [S \rightarrow \varepsilon] \end{array}$$

Which corresponds **precisely** to the steps in a leftmost derivation of string "()".

4.2.2 The LL(1) Parsing Table and Algorithm

Purpose and Example of LL(1) Table

- **Purpose of the LL(1) Parsing Table:**
  - **To express the possible rule choices for a non-terminal *A* when *A* is at the top of parsing stack based on the next input token (the look-ahead).**
- **The LL(1) Parsing table for the following simple grammar:**

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

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

The General Definition of Table

- **The table is a two-dimensional array indexed by non-terminals and terminals**
- **Contain production choices to use at the appropriate parsing step called $M[N,T]$**
    - **N is the set of non-terminals of the grammar**
    - **T is the set of terminals or tokens (including \$)**
- **Any entrances remaining empty**
    - **Represent potential errors**

Table-Constructing Rule

- The table-constructing rule
  - If $A \rightarrow \alpha$ is a production choice, **and there is a derivation $\alpha \stackrel{*}{\Rightarrow} a\beta$**, where ***a*** is a token, then add $A \rightarrow \alpha$ to the table entry $M[A, a]$;
  - If $A \rightarrow \alpha$ is a production choice, and **there are derivations $\alpha \stackrel{*}{\Rightarrow} \varepsilon$ and $S\$ \stackrel{*}{\Rightarrow} \beta A a \gamma$**, where S is the start symbol and ***a*** is a token (or **$**), then add $A \rightarrow \alpha$ to the table entry $M[A, a]$;

A Table-Constructing Case

- The constructing-process of the following table
  - For the production : $S \rightarrow (S)S$, $\alpha = (S)S$, where $a = ($, this choice will be added to the entry $M[S, (]$;
  - Since $S\$ \Rightarrow (S)S\$$, rule 2 applies with $\alpha = \varepsilon$, $\beta = ($, $A = S$, $a =)$, and $\gamma = S\$$, so add the choice $S \rightarrow \varepsilon$ to $M[S,)]$
  - Since $S\$ \Rightarrow S\$$, $S \rightarrow \varepsilon$ is also added to $M[S, \$]$.

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

Rule 1: $\alpha \Rightarrow^* a\beta$
i.e., $a \in \text{first}(\alpha)$

Rule 2: $S\$ \Rightarrow^* \beta A a \gamma$
i.e., $a \in \text{follow}(A)$

Properties of LL(1) Grammar

- **Definition of LL(1) Grammar**
  - **A grammar is an LL(1) grammar if the associated LL(1) parsing table has at most one production in each table entry**
- **An LL(1) grammar cannot be ambiguous**

A Parsing Algorithm Using the LL(1) Parsing Table

(* assumes $ marks the bottom of the stack and the end of the input *)

push the start symbol onto the top of the parsing stack;
while the top of the parsing stack ≠ $ and
the next input token ≠ $ do

if *the top of the parsing stack is terminal a and the next input token is a*
then (* match *)
pop the parsing stack;
advance the input;

A Parsing Algorithm Using the LL(1) Parsing Table

else if ***the top of the parsing stack is non-terminal A***
and ***the next input token is terminal a***
and ***parsing table entry M[A,a] contains production*** $A \rightarrow X_1 X_2 \ldots X_n$
then (* generate *)
pop the parsing stack;
for $i := n$ **downto 1 do**
push X_i **onto the parsing stack;**
else error;
if the top of the parsing stack = $ and the next input token = $
then accept
else error

Example: If-Statements

- **The LL(1) parsing table for simplified grammar of if-statements:**

statement → if-stmt | other

if-stmt → if (exp) statement else-part

else-part → else statement | ε

exp → 0 | 1

M[N,T]	If	Other	Else	0	1	$
Statement	**Statement → if-stmt**	**Statement → other**				
If-stmt	**If-stmt → if (exp) statement else-part**					
Else-part			**Else-part → else statement** **Else-part → ε**			**Else-part → ε**
Exp				Exp → **0**	Exp → **1**	

Notice for Example: If-Statement

- **The entry M[else-part, else] contains two entries, i.e. *the dangling else ambiguity*.**
- **Disambiguating rule: *always prefer the rule that generates the current look-ahead token over any other,* and thus the production**

  Else-part → else statement

  over

  Else-part →ε
- **With this modification, the above table will become unambiguous**

The parsing based LL(1) Table

- The parsing actions for the string:

  If (0) <u>if (1) other else other</u>

- (for conciseness, statement= S, if-stmt=I, else-part=L, exp=E, if=i, else=e, other=o)

Steps	Parsing Stack	Input	Action
1	$S	i(0)i(1)oeo$	S→I
2	$I	i(0)i(1)oeo$	I→i(E)SL
3	$LS)E(i	i(0)i(1)oeo$	Match
4	$ LS)E(	(0)i(1)oeo $	Match
5	$ LS)E	0)i(1)oeo $	E→0
6	$ LS)0	0)i(1)oeo $	Match
7	$ LS)	)i(1)oeo $	Match
8	$ LS	i(1)oeo $	S→I
9	$ LI	i(1)oeo $	I→i(E)SL
10	$ LLS)E(i	i(1)oeo $	Match
11	$ LLS)E(	(1)oeo $	Match
12	$ LLS)E	1)oeo $	E→1
13	$ LLS)1	1)oeo $	Match
14	$ LLS)	)oeo $	match
15	$ LLS	oeo $	S→o
16	$ LLo	oeo $	match
17	$ LL	eo $	L→eS
18	$ LSe	eo $	Match
19	$ LS	o $	S→o
20	$ Lo	o $	match
21	$ L	$	L→ ε
22	$	$	accept

4.2.3 Left Recursion Removal and Left Factoring

Left Recursion Removal

- **Left recursion is commonly used to make operations left associative:**

  exp → exp addop term | term

- **Immediate left recursion:**

  The left recursion occurs only within the production of a single non-terminal.

  exp → exp + term | exp - term |term

- **Indirect left recursion:**

  Never occur in actual programming language grammars, but be included for completeness.

  A → Bb |…

  B → Aa |…

CASE 1: Simple Immediate Left Recursion

- $A \rightarrow A\alpha \mid \beta$

  Where α and β are strings of terminals and non-terminals;
  β does not begin with *A*.

- **The grammar will generate the strings of the form** $\beta\alpha^{n}$

- ***We rewrite this grammar rule into two rules:***

  $A \rightarrow \beta A'$

  To generate β first;

  $A' \rightarrow \alpha A' \mid \varepsilon$

  To generate the repetition of α, using right recursion.

Example

- **exp → exp addop term | term**
- **To rewrite this grammar to remove left recursion, we obtain (α = addop term and β = term)**

exp → term exp'

exp' → addop term exp' | ε

$A \rightarrow \beta A'$

To generate β first;

$A' \rightarrow \alpha A' \mid \varepsilon$

To generate the repetition of α,

CASE2: General Immediate Left Recursion

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$

Where none of $\beta_1, \ldots, \beta_m$ begin with A.

The solution is similar to the simple case:

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon$

Example

- **exp → exp <u>+ term</u> | exp <u>- term</u> | <u>term</u>**

- **Remove the left recursion as follows:**

  exp → term exp’

  exp’ → + term exp’ | - term exp’ | ε

Notice

- Left recursion removal does not change the language, but
  - Change the grammar and the parse tree
- This change causes a complication for the parser

Example

Simple arithmetic expression grammar

exp → exp addop term | term

addop → +|-

term → term mulop factor | factor

mulop →*

factor →(exp) | number

After removal of the left recursion

exp → term exp'

exp'→ addop term exp' | ε

addop → +|-

term → factor term'

term' → mulop factor term' | ε

mulop →*

factor →(exp) | number

The LL(1) parsing table for the new grammar

M[N,T]	(	number	)	+	-	*	$
Exp	exp → term exp'	exp → term exp'					
Exp'			exp' → ε	exp' → addop term exp'	exp' → addop term exp'		exp' → ε
Addop				addop → +	addop → −		
Term	term → factor term'	term → factor term'					
Term'			term' → ε	term' → ε	term' → ε	term' → mulop factor term'	term' → ε
Mulop						mulop →*	
factor	factor → (expr)	factor → number					

Left Factoring

- Left factoring is required when two or more grammar rule choices share a common prefix string, as in the rule

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

- Example:

stmt-sequence→stmt; stmt-sequence | stmt

stmt→s

- An LL(1) parser cannot distinguish between the production choices in such a situation

- The solution in this simple case is to "factor" the α out on the right and rewrite the rule as two rules:

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta | \gamma$$

Algorithm for Left Factoring

While there are changes to the grammar do

For each non-terminal 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 $A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$ be all the production choices for A

and suppose that $\alpha_1, \alpha_2, \ldots, \alpha_k$ share α, so that $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \ldots | \alpha\beta_k | \alpha_{k+1} | \ldots | \alpha_n$, the β_j's share no common prefix, and $\alpha_{k+1}, \ldots, \alpha_n$ do not share α

Replace the rule $A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$ by the rules

$A \rightarrow \alpha A' | \alpha_{k+1} | \ldots | \alpha_n$

$A' \rightarrow \beta_1 | \beta_2 | \ldots | \beta_k$

Example 4.4

- **Consider the grammar for statement sequences, written in right recursive form:**

  Stmt-sequence→stmt; stmt-sequence | stmt

  Stmt→s

- **Left Factored as follows:**

  Stmt-sequence→stmt stmt-seq'

  Stmt-seq'→; stmt-sequence | ε