3.3 THE PARSING PROBLEM

Problem:

"Given a context-free grammar G and string x, is $x \in L(G)$?"

Solution method = *parsing algorithm*.

Many alternative methods, especially when G is of some limited (practical) form.

Derivations and parse trees

Let $\gamma \in V^*$ be a sentence derivation of the grammar $G = (V, \Sigma, P, S)$.

The sequence of direct derivations

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \cdots \Rightarrow \gamma_n = \gamma$$

from the initial symbol S to the string γ is called the *derivation* of γ in G.

The *length* of a derivation is the number of direct derivations in it (previously n).

Example: derivations of the sentence $a + a$ in grammar G_{expr}:

$$\begin{array}{lllllllll}
\text{(i)} & E & \Rightarrow & E+T & \Rightarrow & T+T & \Rightarrow & F+T \\
 & & \Rightarrow & a+T & \Rightarrow & a+F & \Rightarrow & a+a \\
\text{(ii)} & E & \Rightarrow & E+T & \Rightarrow & E+F & \Rightarrow & T+F \\
 & & \Rightarrow & F+F & \Rightarrow & F+a & \Rightarrow & a+a \\
\text{(iii)} & E & \Rightarrow & E+T & \Rightarrow & E+F & \Rightarrow & E+a \\
 & & \Rightarrow & T+a & \Rightarrow & F+a & \Rightarrow & a+a.
\end{array}$$

A derivation $\gamma \Rightarrow^* \gamma'$ is a *left derivation*, denoted by

$$\gamma \underset{\text{lm}}{\Rightarrow}^* \gamma',$$

if in each derivation the production has been applied to the left-most nonterminal in the string (see (i) above).

A *right derivation* is defined similarly (see (iii) above) and denoted by

$$\gamma \underset{\text{rm}}{\Rightarrow}^* \gamma'$$

Direct left and right derivations are denoted by $\gamma \underset{\text{lm}}{\Rightarrow} \gamma'$ and $\gamma \underset{\text{rm}}{\Rightarrow} \gamma'$.

Let $G = (V, \Sigma, P, S)$ be a context-free grammar

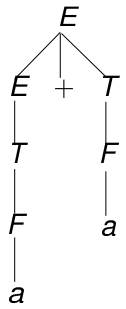
A *parse tree* in G is an ordered tree with the following properties:

(i) the vertices of the tree are labelled with elements of $V \cup \{\varepsilon\}$ such that the internal vertices are labelled with non-terminals in $N = V - \Sigma$ and the root vertex is labelled with the initial symbol S;

(ii) if A is the name of an internal vertex and $X_1, \ldots, X_k$ are the names of its descendants taken in order, then
$A \to X_1 \ldots X_k$ is a production of G.

The *yield* τ of a parse tree is the string obtained by catenating the names of its leaf vertices in pre-order ("from left to right").

Example. Parse tree of the sentence $a + a$ in grammar G_{expr}:



Derivation of the sentence:

$$\begin{aligned} E &\Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \\ &\Rightarrow a+T \Rightarrow a+F \Rightarrow a+a \end{aligned}$$

Constructing a parse tree that corresponds to the derivation

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \cdots \Rightarrow \gamma_n = \gamma :$$

(i) the root vertex is labelled with S; if $n = 0$, the tree has no other vertices; otherwise

(ii) If the production $S \to X_1 X_2 \ldots X_k$ has been applied in the first step, the root will have k descendants, whose names from left to right are $X_1, X_2, \ldots, X_k$;

(iii) If the production $X_i \to Y_1 Y_2 \ldots Y_l$ has been used in the next step, then the ith descendant of the root will have l descendants, whose names from left to right are $Y_1, Y_2, \ldots, Y_l$; etc.

We observe that if τ is the parse tree that corresponds to some derivation $S \Rightarrow^* \gamma$ then τ yields γ.

Let τ be a parse tree in G that yields the string of terminals x.

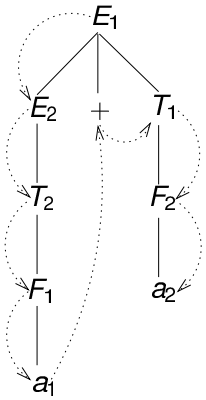
The left derivation of x can be obtained from τ by visiting the vertices of the tree in pre-order ("top down, left to right") and by expanding the non-terminals in order in the manner suggested by the tree.

The right derivation can be obtained in reverse preorder ("top down, right to left").

If a parse tree is formed from a given left derivation $S \underset{\text{lm}}{\Rightarrow}^* x$ and again a left derivation is derived from the resulting parse tree, we obtain the original left derivation; similarly for right derivations.

Example. Forming the left derivation of $a + a$ from the parse tree.

Parse tree:



Vertices in pre-order:

$E_1 E_2 T_2 F_1 a_1 + T_1 F_2 a_2$

Left derivation:

$$\begin{aligned} E &\underset{\text{lm}}{\Rightarrow} E+T \underset{\text{lm}}{\Rightarrow} T+T \underset{\text{lm}}{\Rightarrow} F+T \\ &\underset{\text{lm}}{\Rightarrow} a+T \underset{\text{lm}}{\Rightarrow} a+F \underset{\text{lm}}{\Rightarrow} a+a \end{aligned}$$

Theorem 3.3 Let $G = (V, \Sigma, P, S)$ be a context-free grammar. Then:

(i) every sentence derivation γ of G corresponds to a parse tree τ in G that yields γ;

(ii) every parse tree τ in G that yields the string of terminals x has unique corresponding left and right derivations $S \underset{\text{lm}}{\Rightarrow}^* x$ and $S \underset{\text{rm}}{\Rightarrow}^* x$.

Corollary 3.4 Every sentence in G has a left and a right derivation.

That is: the parse trees, left and right derivations produced by a context-free grammar are in one-to-one correspondance with each other.

In addition to solving the decision problem "Is $x \in L(G)$?" it is usually expected that one of these representations of the parsing is produced.

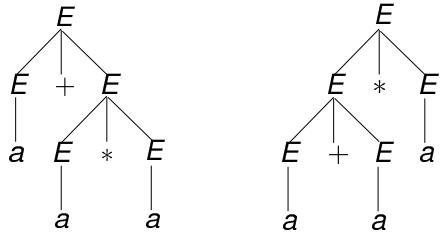
Ambiguity of a grammar

A sentence may have several derivations in a grammar.

Example. Examine the grammar of simple arithmetic expressions:

$$G'_{\text{expr}} = \{E \to E + E,\ E \to E * E,\ E \to a,\ E \to (E)\}.$$

There are two ways to parse $a + a * a$ in this grammar:



A context-free grammar G is *ambiguous*, if some sentence $x \in G$ has two different parse trees in G. Otherwise the grammar is *unambiguous*.

Ambiguity is usually undesirable, as it means that the given sentence has two alternative "interpretations."

If all context-free grammars that produce the language L are ambiguous, L is said to be *structurally ambiguous*.

For example G'_{expr} is ambiguous while G_{expr} and G_{match} are unambiguous. The language $L_{\text{expr}} = L(G'_{\text{expr}})$ is not structurally ambiguous, since it also has a unique grammar G_{expr}. For example, the language

$$\{a^i b^j c^k \mid i = j \text{ or } j = k\}$$

is structurally ambiguous (proof omitted).

3.4 Recursive descent parsing

A (inefficient in the general case!) method of looking for a left derivation (parse tree) in G for a given sentence x is to start from the initial symbol of G and to generate systematically all possible left derivations (parse trees) while matching the terminals of the sentence derivation (leaves of the tree) to the symbols in x. If an incompatibility is encountered, the last production choice is cancelled and the next alternative is tried.

Such a parsing method is called *recursive descent parsing*, since the algorithm tries to derive x from the initial symbol by recursively trying out all combinations and matching the structure with the given sentence.

Example. Examine the grammar G:

$$\begin{aligned} E &\rightarrow T+E \mid T-E \mid T \\ T &\rightarrow a \mid (E). \end{aligned}$$

Recursive descent parsing of $a-a$ in G:

$$\begin{aligned} E &\Rightarrow T+E \Rightarrow a+T && \text{[contradiction; return]} \\ &\Rightarrow (E)+T && \text{[contradiction; return]} \\ &\Rightarrow T-E \Rightarrow a-E \Rightarrow a-T+E \Rightarrow a-a+E && \\ & && \text{[contradiction; return]} \\ &\Rightarrow T-E \Rightarrow a-E \Rightarrow a-T+E \Rightarrow a-(E)+E && \\ & && \text{[contradiction; return]} \\ & \Rightarrow a-T-E \Rightarrow a-a-E && \\ & && \text{[contradiction; return]} \\ & \Rightarrow a-T-E \Rightarrow a-(E)-E && \\ & && \text{[contradiction; return]} \\ & \Rightarrow a-T \Rightarrow a-a && \text{[OK!]} \end{aligned}$$

The recursive descent method can be much more efficient, if the grammar has the property that at each point in parsing the *next symbol* in the input stream determines uniquely which production must be chosen. Such grammars are *LL(1)-type* grammars.

Let us "factor" the productions of the nonterminal E to obtain the equivalent grammar G':

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +E \mid -E \mid \varepsilon \\ T &\rightarrow a \mid (E). \end{aligned}$$

Parsing $a-a$ in G' (the input symbol that determines the production used is written above the derivation arrow):

$$E \underset{\text{lm}}{\Rightarrow} TE' \overset{a}{\underset{\text{lm}}{\Rightarrow}} aE' \overset{-}{\underset{\text{lm}}{\Rightarrow}} a-E \underset{\text{lm}}{\Rightarrow} a-TE' \overset{a}{\underset{\text{lm}}{\Rightarrow}} a-aE' \overset{\varepsilon}{\underset{\text{lm}}{\Rightarrow}} a-a.$$

It is easy to write a program to parse an LL(1)-type grammar as recursive procedures. For example the following collection of C functions can be written on the basis of G. The program outputs the productions used for the left derivation in order.

```
#include <stdio.h>

int next;
void E(void); void Eprime(void); void T(void);

void E(void)
{
  printf("E  -> TE'\n");
  T(); Eprime();
}
```



```
void Eprime(void)
{
  if (next == '+') {
    printf("E' -> +E\n");
    next = getchar();
    E();
  }
  else if (next == '-') {
    printf("E' -> -E\n");
    next = getchar();
    E();
  }
  else
    printf("E' -> \n");
}
```

```
void T(void)
{
  if (next == 'a') {
    printf("T  -> a\n");
    next = getchar();
  }
  else if (next == '(') {
    printf("T  -> (E)\n");
    next = getchar();
    E();
    if (next != ')')
      ERROR(") expected.");
    next = getchar();
  }
  else ERROR("T cannot start with this.");
}
```



```
void ERROR(char *msg)
{
  printf("%s\n",msg); exit(1);
}

int main(void)
{
  next = getchar();
  E();
  exit(0);
}
```


For example in parsing `a-(a+a)` the program outputs the following lines:

```
E → TE'
T → a
E' → -E
E → TE'
T → (E)
E → TE'
T → a
E' → +E
E → TE'
T → a
E' →
E' →
```

The output corresponds to the left derivation

$$\begin{aligned} E &\Rightarrow TE' \Rightarrow aE' \Rightarrow a-E \Rightarrow a-TE' \\ &\Rightarrow a-(E)E' \Rightarrow a-(TE')E' \\ &\Rightarrow a-(aE')E' \Rightarrow a-(a+E)E' \\ &\Rightarrow a-(a+TE')E' \Rightarrow a-(a+aE')E' \\ &\Rightarrow a-(a+a)E' \Rightarrow a-(a+a). \end{aligned}$$

3.5 Attribute grammars

A way of adding simple description of semantics to context-free grammars.

Every vertex of the parse tree named with the symbol X is seen as a "data record" of "type" X. The "fields" in a record of type X are called the *attributes* of X and denoted with $X.s$, $X.t$ etc. It is thought that every vertex of type X has distinct *instances* of the attributes of X.

Rules for evaluating the attributes are added to the productions $A \to X_1 \dots X_k$ in the grammar. The rules indicate how the values of the attributes in a vertex of the parse tree are determined from the attribute values of its parent and descendants.

In principle the rules can be any functions that only take as arguments locally available information. More exactly: The rules attached to the production $A \to X_1 \dots X_k$ may only mention attributes of the symbols $A, X_1, \dots, X_k$.

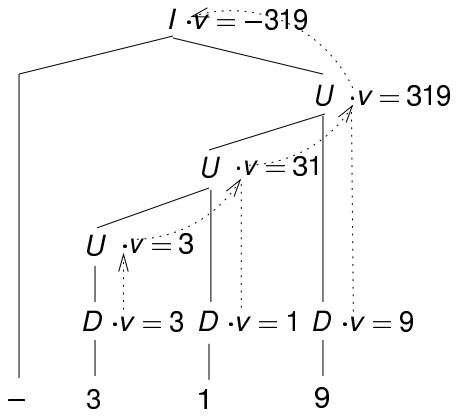
Example. Evaluating signed integers (grammar + attribute evaluation rules).

An attribute instance $X.v$ is added to every nonterminal of type X in the parse tree. The value of $X.v$ will be the value of the string of numbers derived from X; especially the attribute v of the root vertex will have the value of the whole string of numbers that is the yield of the parse tree.

Productions:			*Evaluation rules:*		
I	$\to$	$+U$	$I.v$	$:=$	$U.v$
I	$\to$	$-U$	$I.v$	$:=$	$-U.v$
I	$\to$	U	$I.v$	$:=$	$U.v$
U	$\to$	D	$U.v$	$:=$	$D.v$
U	$\to$	UD	$U_1.v$	$:=$	$10 * U_2.v + D.v$
D	$\to$	0	$D.v$	$:=$	0
$\dots$					
D	$\to$	9	$D.v$	$:=$	9

In the evaluation rule for $U \to UD$ the different instances of the nonterminal U are identified by subscripts.

With these rules the attributed parse tree of the sentence “-319” is the following:



An attribute t of an attribute grammar is *synthetic*, if the evaluation rule attached to each production $A \to X_1 \dots X_k$ is of the form

$$A.t := f(A, X_1, \dots, X_k).$$

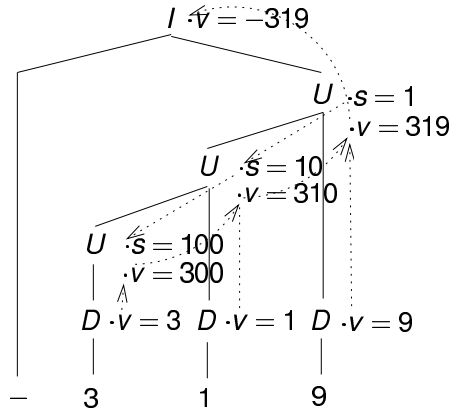
Then the value of every instance of the t attribute only depends on the value of the attributes of the corresponding vertex and its descendants. Other kinds of attributes are *hereditary*.

In describing attribute semantics mainly synthetic attributes are used, since they can be evaluated in one pass from the leaves to the root of the tree. It is also possible to use hereditary attributes — as long as the dependancy graph of the attribute instances contains no cycles.

Example: evaluating integers by using a hereditary "position factor" attribute s and a synthetic "value" attribute v:

Productions:			*Evaluation rules:*			
I	$\rightarrow$	$+U$	$U.s$	$:=$	1, $I.v$	$:=$ $U.v$
I	$\rightarrow$	$-U$	$U.s$	$:=$	1, $I.v$	$:=$ $-U.v$
I	$\rightarrow$	U	$U.s$	$:=$	1, $I.v$	$:=$ $U.v$
U	$\rightarrow$	D			$U.v$	$:=$ $(D.v) * (U.s)$
U	$\rightarrow$	UD	$U_2.s$	$:=$	$10 * (U_1.s)$,	
			$U_1.v$	$:=$	$U_2.v + (D.v) * (U_1.s)$	
D	$\rightarrow$	0			$D.v$	$:=$ 0
$\vdots$						
D	$\rightarrow$	9			$D.v$	$:=$ 9

By this position factor technique one can obtain the following attributed parse tree for the sentence "-319":



The values of the attribute instances can often be calculated directly in the parsing routines without constructing the parse tree explicitly.

Example. A program that transforms arithmetic expressions given as input to *postfix* notation.

To the usual grammar for producing simple arithmetic expressions a synthetic, string-valued attribute pf is added; the value of the attribute instance $X.pf$ attached to each nonterminal X is the post-fix presentation of the subexpression rooted at X.

Productions:			*Evaluation rules:*		
E	$\rightarrow$	$T + E$	$E_1.pf$	$:=$	$(T.pf)\hat{}(E_2.pf)\hat{}('+')$
E	$\rightarrow$	T	$E.pf$	$:=$	$T.pf$
T	$\rightarrow$	$F * T$	$T_1.pf$	$:=$	$(F.pf)\hat{}(T_2.pf)\hat{}('*')$
T	$\rightarrow$	F	$T.pf$	$:=$	$F.pf$
F	$\rightarrow$	a	$F.pf$	$:=$	$'a'$
F	$\rightarrow$	(E)	$F.pf$	$:=$	$E.pf$

A recursive descent parser, that evaluates the values of the attribute instances directly while parsing:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXLEN 80          /* max. expression length */

int next;
char *pf;                  /* resulting expression   */
void E(char *); void T(char *); void F(char *);

void ERROR(char *msg)
{
  printf("%s\n",msg); exit(1);
}
```

```
/* Productions: E -> T+E | T  */
void E(char *pf)
{
  char *pf1, *pf2;
  pf1 = (char *) malloc(MAXLEN+1);
  pf2 = (char *) malloc(MAXLEN+1);
  T(pf1);                  /* pf1  = T.pf    */
  if (next == '+') {
    next = getchar();
    E(pf2);                /* pf2  = E(2).pf */
    strcpy(pf,strcat(pf1,strcat(pf2,"+")));
                           /* E.pf = pf1^pf2^('+') */
  }
  else strcpy(pf,pf1);     /* E.pf = T.pf    */
  free(pf1); free(pf2);
}
```



```
/* Productions: T -> F*T | F  */
void T(char *pf)
{
  char *pf1, *pf2;
  pf1 = (char *) malloc(MAXLEN+1);
  pf2 = (char *) malloc(MAXLEN+1);
  F(pf1);                  /* pf1  = F.pf    */
  if (next == '*') {
    next = getchar();
    T(pf2);                /* pf2  = T(2).pf */
    strcpy(pf,strcat(pf1,strcat(pf2,"*")));
                           /* T.pf = pf1^pf2^('*') */
  }
  else strcpy(pf,pf1);     /* T.pf = F.pf    */
  free(pf1); free(pf2);
}
```



```
/* Productions: F -> a | (E)  */
void F(char *pf)
{
  if (next == 'a') {
    strcpy(pf,"a");        /* F.pf = 'a'     */
    next = getchar();
  }
  else if (next == '(') {
    next = getchar();
    E(pf);                 /* F.pf = E.pf    */
    if (next != ')')
      ERROR(") expected.");
    next = getchar();
  }
  else ERROR("F cannot start with this.");
}
```



```
int main(void)
{
  next = getchar();
  pf = (char *) malloc(MAXLEN+1);
  E(pf);
  printf("%s\n", pf);
  free(pf);
}
```