Compiler Theory: Syntax Analysis

Marc Moreno Maza
University of Western Ontario
CS447a – Fall 2004

December 1, 2004

1 Grammars and Parsing

1.1 An introduction to the notion of a grammar

Informally, a GRAMMAR describes the hierarchical structure of programming language constructs. For instance an *if-then-else* statement in C has the form

$$\textbf{if } (\mathit{expression})\ \mathit{statement}\ \textbf{else}\ \mathit{statement}$$

Using the variable *expr* to denote an expression and the variable *stmt* to denote a statement, the above rule can be reformulated as follows in order to express that an *if-then-else* statement is a particular statement

$$\mathit{stmt} \longmapsto \textbf{if } (\mathit{expr})\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt}$$

Such formulation will called a PRODUCTION.

Example 1 *Another example is that of arithmetic expressions over arbitrary variables (i.e. expression involving variables, the four arithmetic operations and exponentiation) which can be defined by the following productions (or grammatical rules):*

$$
\begin{array}{lll}
\text{expr} & \longmapsto & \text{expr binop expr} \\
\text{expr} & \longmapsto & (\text{expr}) \\
\text{expr} & \longmapsto & -\text{expr} \\
\text{expr} & \longmapsto & \mathbf{id} \\
\text{binop} & \longmapsto & + \\
\text{binop} & \longmapsto & - \\
\text{binop} & \longmapsto & * \\
\text{binop} & \longmapsto & / \\
\text{binop} & \longmapsto & \uparrow
\end{array}
$$

where **id** *stands for any variable.*

Remark 1 Let I be a finite set of identifiers and let us consider the alphabet

$$\Sigma \quad = \quad I \cup \{+, -, *, /, \uparrow, (,)\}. \tag{1}$$

Let L be the language consisting of arithmetic expressions over Σ. We shall show that L cannot be recognized by FA.

First we observe that every word $w = w_1 \cdots w_\ell$ of L is a *well parenthesized expression* that is

$$(\forall\, i = 1 \cdots \ell - 1) \quad |\, w_1 \cdots w_i|_(\geq |\, w_1 \cdots w_i|_) \quad \text{and} \quad |\, w|_(= |\, w|_) \tag{2}$$

where $|u|_x$ denotes the number of occurrences of the letter x in the word u. The proof is easy by induction.

Let us assume that L can be recognized by a DFA $\mathcal{A} = (\Sigma, S, s_0, F, \delta)$ with N states and initial state s_0. Let us consider an arithmetic expression a starting with N opening parentheses. There exists N states $s_1, s_2, \ldots, s_N$ such that for every $i = 1 \cdots N - 1$ we have

$$s_{i+1} \;=\; \delta(s_i, x) \quad \text{with} \quad x = (\tag{3}$$

There exists at least two values i_1 and i_2 of the index i in the range $0 \cdots N$ such that $s_{i_1} = s_{i_2}$. Hence by adding $i_2 - i_1$ opening parentheses in front of a we can build another word b recognized by L. However b is clearly not a well parenthesized expression and thus it is not an arithmetic expression. Therefore the language of arithmetic expressions over Σ cannot be recognized by FA.

It follows from Remark 1 that finite automata cannot recognize all languages that we may encounter when building a compiler. The notion of a *grammar* (to be introduced later) solves this difficulty.

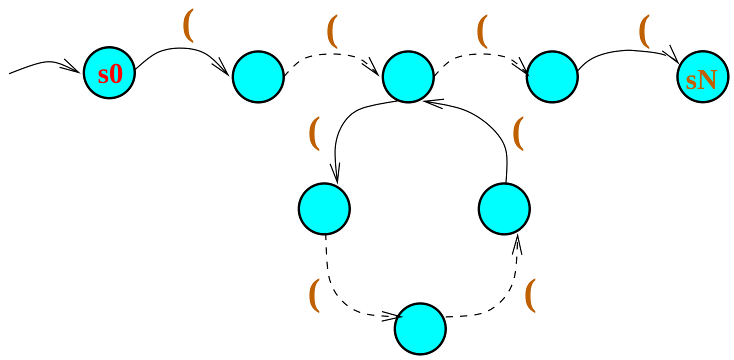

Figure 1: Well parenthesized expressions cannot be recognized by FA.

1.2 The notion of a grammar

<u>A GRAMMAR</u> is quadruple (V_T, V_N, S, P) such that

- V_T is a finite set of symbols called *terminals*,
- V_N is a finite set of symbols called *non-terminals*,
- S is a distinguished non-terminal called *start-symbol*,
- P is a finite set of couples (α, β) called *productions* where
  - α and β belong to $(V_T \cup V_N)^*$ and
  - α does **not** belong $(V_T)^*$.

In other words a production is made of two words over the alphabet $V_T \cup V_N$ such that the first word contains at least one non-terminal symbol. The set $V_T \cup V_N$ is called the set of *grammar symbols*.

Notation 1 Here are some usual notational conventions about grammars.

- It is convenient to denote a production $\alpha \longmapsto \beta$ instead of (α, β).
- By default, the following symbols are *terminals*: lower-case letters, arithmetic operators, punctuation symbols, digits and boldface strings (**if, id**, ...).
- By default, the following symbols are *non-terminals*: upper-case letters early in the alphabet such as $A, B, C, \ldots$, the letter S (for the start-symbol), lower-case italic names such as *stmt, expr*.

- By default, the following symbols are *grammar symbols*: upper-case letters late in the alphabet such as X, Y, Z
- By default, *strings of terminals* are denoted by lower-case letters late in the alphabet such as u, v, w.
- By default, *strings of grammar symbols* are denoted by lower-case Greek letters early in the alphabet such as α, β, γ.
- By default, the start symbol is the left side of the first production.
- If $A \longmapsto \alpha_1$, $A \longmapsto \alpha_2$, $\ldots A \longmapsto \alpha_n$ are all the productions with A as their left side (these productions are called A-productions) then we can write $A \longmapsto \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$. The sequence $\alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$ is called the alternatives of A.

Example 2 *Using the conventions of Notation 1 the grammar of Example 1 can be stated as follows*

$$\begin{array}{lcl} E & \longmapsto & E\ A\ E \mid (E) \mid -E \mid \mathbf{id} \\ A & \longmapsto & + \mid - \mid * \mid / \mid \uparrow \end{array}$$

Remark 2 Grammars offer significant advantages to both language designers and compiler writers. Among them

- A grammar gives a precise and easy to understand syntactic specification of a programming language.
- From certain classes of grammars we can automatically construct an efficient parser that determines if a source program is syntactically well formed.
- Developing a language given by a grammar (adding or removing constructs) is easy.

<u>DERIVATIONS.</u> Let $G = (V_T, V_N, S, P)$ be a grammar and let α and β be two strings of grammar symbols for G. We say that β *derives from* α *in one step* and we write $\alpha \Longrightarrow \beta$ if there exist strings of grammar symbols α_1, α_2 α_3, β_2, such that

- $\alpha = \alpha_1 \alpha_2 \alpha_3$,

- $\beta = \alpha_1 \beta_2 \alpha_3$,
- $\alpha_2 \longmapsto \beta_2$ is a production of G.

The transitive and reflexive closure of the map $(\alpha, \beta) \longmapsto \alpha \Longrightarrow \beta$ over the sets of grammar symbols is denoted by $(\alpha, \beta) \longmapsto \alpha \overset{*}{\Longrightarrow} \beta$. Thus for two grammar symbols α and β we have

$$\alpha \overset{*}{\Longrightarrow} \beta \quad \Longleftrightarrow \quad \begin{cases} \alpha = \beta \\ \text{or} \\ (\exists\, \gamma)\ (\alpha \overset{*}{\Longrightarrow} \gamma \text{ and } \gamma \Longrightarrow \beta) \end{cases} \tag{4}$$

Intuitively, $\alpha \overset{*}{\Longrightarrow} \beta$ means that β derives from α in a finite number of steps. A sequence of strings of grammar symbols $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ is a DERIVATION if we have

$$\alpha_1 \Longrightarrow \alpha_2 \Longrightarrow \cdots \Longrightarrow \alpha_n. \tag{5}$$

THE LANGUAGE GENERATED BY A GRAMMAR. Let $G = (V_T, V_N, S, P)$ be a grammar. The language over V_T generated by G and denoted by $L(G)$ is the set of the words w of ${V_T}^*$ such that $S \overset{*}{\Longrightarrow} w$. The words of $L(G)$ are called the *sentences* of G. More generally any string of grammar symbols α such that $S \overset{*}{\Longrightarrow} \alpha$ is called a *sentential form* of G.

A language L over the alphabet $\Sigma = V_T$ is said *formal* if there exists a grammar G such that L is generated by G. Observe that

- Some languages are not formal like natural languages.
- Two different sequences of derivations-in-one-step can lead to the same sentential form.
- Moreover, two different grammars can generate the same language.

These last two observations are illustrated by Example 3.

Example 3 Let us consider again arithmetic expressions. For simplicity we consider only two operations $+$ and $*$. Our language L can be generated by the grammar G_1:

$$expr \quad \longmapsto \quad expr + expr \mid expr * expr \mid (expr) \mid \mathbf{id}$$

And also by the grammar G_2:

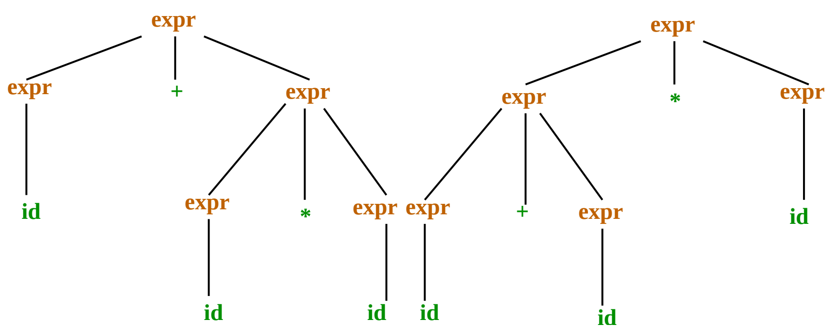

Figure 2: Two parse trees for the same sentence.

$$
\begin{array}{lcl}
expr & \longmapsto & expr + term \mid term \\
term & \longmapsto & term * factor \mid factor \\
factor & \longmapsto & (expr) \mid \mathbf{id}
\end{array}
$$

Consider now the arithmetic expression $\mathbf{id} + \mathbf{id} * \mathbf{id}$. Two different sequences of derivations-in-one-step of G_1 can lead to it:

$$
\begin{array}{lcl}
expr & \Rightarrow & expr + expr \\
 & \Rightarrow & expr + expr * expr \\
 & \Rightarrow & \mathbf{id} + expr * expr \\
 & \Rightarrow & \mathbf{id} + \mathbf{id} * expr \\
 & \Rightarrow & \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{array}
\qquad
\begin{array}{lcl}
expr & \Rightarrow & expr * expr \\
 & \Rightarrow & expr + expr * expr \\
 & \Rightarrow & \mathbf{id} + expr * expr \\
 & \Rightarrow & \mathbf{id} + \mathbf{id} * expr \\
 & \Rightarrow & \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{array}
\tag{6}
$$

This is sketched by the trees of Figure 2, called *parse trees*. With G_2 several derivations-in-one-step can lead to $\mathbf{id} + \mathbf{id} * \mathbf{id}$. However they all correspond to the same tree shown on Figure 3. This notion of a *parse tree* is formalized later.

Non-terminals can be seen as syntactic variables that denote the sets of strings that can be derived from them. They also impose a hierarchical structure on the language that is useful for both syntax analysis and translation. Moreover Example 3 shows that an accurate use of non-terminals can reduce ambiguity.

1.3 Chomsky classification

The grammars of Example 3 have the nice following property: every production has the form $A \longmapsto \alpha$ where A is a non-terminal symbol and α is

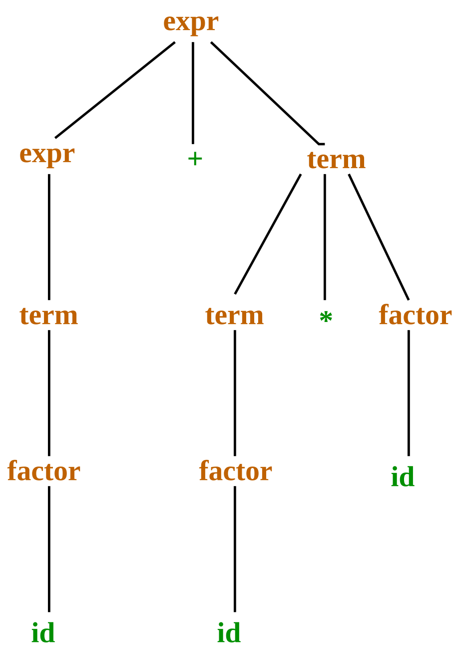

Figure 3: The parse tree of $\mathbf{id} + \mathbf{id} * \mathbf{id}$ with G_2.

a string of grammar symbols. These grammars are called *context-free grammars* and will be studied in the next section. They are one of the classes of the classification of Chomsky that we present now.

<u>TYPE 1.</u> The grammar $G = (V_T, V_N, S, P)$ is of *type* 1 if every production has the form

$$\alpha_1 A \alpha_2 \longmapsto \alpha_1 \beta \alpha_2 \quad \text{where} \quad \begin{cases} A \in V_N \\ \alpha_1, \alpha_2, \beta \in (V_T \cup V_N)^* \end{cases} \tag{7}$$

Such a grammar is also called a *context-sensitive grammar*.

<u>TYPE 2.</u> The grammar $G = (V_T, V_N, S, P)$ is of *type* 2 if every production has the form

$$A \longmapsto \alpha \quad \text{where} \quad \begin{cases} A \in V_N \\ \alpha \in (V_T \cup V_N)^* \end{cases} \tag{8}$$

Such a grammar is also called a *context-free grammar*.

<u>TYPE 3.</u> The grammar $G = (V_T, V_N, S, P)$ is said *right regular* if every production has the form

$$A \longmapsto aB \quad \text{or} \quad A \longmapsto a \quad \text{where} \quad \begin{cases} A, B \in V_N \\ a \in V_T \cup \{\varepsilon\} \end{cases} \tag{9}$$

The grammar $G = (V_T, V_N, S, P)$ is said *left regular* if every production has the form

$$A \longmapsto Ba \quad \text{or} \quad A \longmapsto a \qquad \text{where} \quad \begin{cases} A, B \in V_N \\ a \in V_T \cup \{\varepsilon\} \end{cases} \tag{10}$$

The grammar G is said *regular* or of *type* 3 if it is either right regular or left regular.

<u>TYPE 0.</u> The grammar $G = (V_T, V_N, S, P)$ is of *type* 0 if it is not of type 1, 2 or 3.

<u>THE TYPE OF A LANGUAGE.</u> Let n be an integer in the range $0, 1, 2, 3$. A language L over an alphabet Σ is said of *type* n if it is generated by a grammar of type n and cannot be generated by a grammar of type $n+1$. A language of type 2 is also said *context-free* (or *algebraic*) and a language of type 3 is also said *regular*.

<u>REGULAR LANGUAGES.</u> Let $\mathcal{A} = (\Sigma, S, s_0, F, \delta)$ be a DFA. We define the grammar $G = (\Sigma, S, s_0, P)$ such that

- its terminals are the letters of Σ,
- its non-terminals are the states of the automaton $\mathcal{A}$,
- its start-symbol is the initial state of $\mathcal{A}$,
- its productions are defined as follows: for each $x \in \Sigma$ and each $s \in S$ such that $\delta(s, x)$ is defined we set the rule $s \longmapsto xs'$ where $s' = \delta(s, x)$. Moreover if s' is a final state we set the rule $s' \longmapsto \varepsilon$.

It is not hard to prove that the language recognized by $\mathcal{A}$ is the language generated by G. Observe that G is a regular grammar. Conversly, one can build a FA from a regular grammar such that their associated languages match. Hence we can state the following theorem.

Theorem 1 *A grammar G is regular iff the language generated by G is recognized by finite automata.*

Example 4 Consider $\Sigma = \{a, b\}$ and the language L over Σ denoted by the regular expression $r = (a + b)^* abb$. From r one can easily build the NFA shown on Figure 4. Then applying the subset algorithm one obtains the DFA shown on the same picture. The leads us to the following grammar:

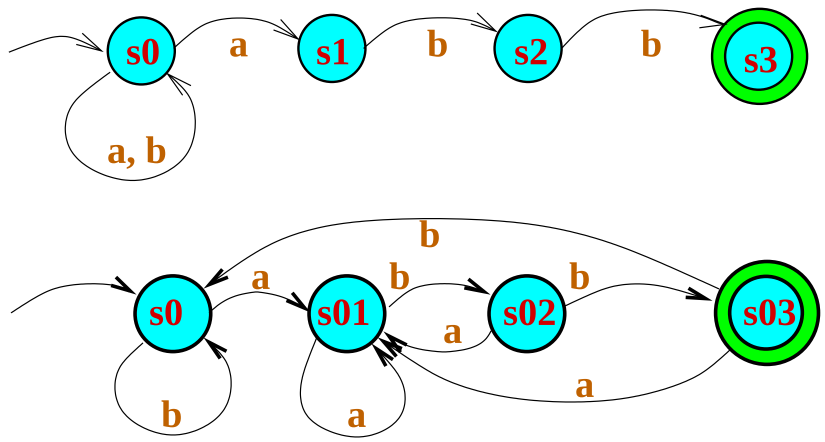

Figure 4: a NFA and a DFA denoted by $(a+b)^*abb$.

.

$$\begin{array}{lcl} s_0 & \longmapsto & as_{01} \mid bs_0 \\ s_{01} & \longmapsto & as_{01} \mid bs_{02} \\ s_{02} & \longmapsto & as_{01} \mid bs_{03} \\ s_{03} & \longmapsto & as_{01} \mid bs_0 \mid \varepsilon \end{array} \tag{11}$$

<u>CONTEXT-FREE LANGUAGES.</u> We start with two examples of context-free languages that we met already. Then we give a *pumping-lemma* for context-free languages as Theorem 2.

Example 5 Consider $\Sigma = \{(,)\}$ and let L be the language of the well parenthesized expressions over Σ. So these expressions are only made of parentheses and satisfy Condition 2 of Remark 1. We consider the grammar $G = (\Sigma, \{S\}, S, P)$ where the set of productions P is given by

$$S \longmapsto (S)S \mid \varepsilon \tag{12}$$

Let L' be the language generated by G. Let us show that L is L'. By induction (on the length of a generated word) it is easy to prove that every word in L' is a well parenthesized expression, so $L' \subseteq L$. Conversly by induction on the length of a well parenthesized expression, it is easy to prove that all such expressions can be generated by G, so $L \subseteq L'$. Hence $L = L'$.

Example 6 Consider $\Sigma = \{a, b\}$ and $L = \{a^n b^n \mid n \in \mathbb{N}\}$. The language L is of type 2. Indeed we already know that L is not recognized by FA and thus it is not of type 3. It is easy to show that L can be generated by the following context-free grammar.

$$S \quad \longmapsto \quad aSb \mid \varepsilon$$

Theorem 2 (Bar-Hillel, Perles, Shamir) *Let $G = (V_T, V_N, S, P)$ be a context-free grammar. There exists an integer N such that for every word m generated by G with length $\mid m \mid \geq N$ there exist words u, v, w, x, y over V_T such that*

$$m \; = \; u\,v\,w\,x\,y \quad \text{with} \quad \begin{cases} v \neq \varepsilon \;\; \text{or} \;\; x \neq \varepsilon \\ \mid v\,w\,x \mid < N \\ (\forall k \in |N)\; u\,v^k\,w\,x^k\,y \in L \end{cases} \tag{13}$$

Example 7 Consider the alphabet $\Sigma = \{a, b, c\} = V_T$ and the language

$$L \;=\; \{a^n b^n c^n \;\mid\; n \in \mathbb{N}\} \tag{14}$$

over Σ. We define $V_N = \{S, T, U, W, X, Y, Z, B\}$. The grammar $G = (V_T, V_N, S, P)$ with P given by

$$\begin{array}{lcl} S & \longmapsto & abc \\ S & \longmapsto & aTBc \\ TB & \longmapsto & UB \\ UB & \longmapsto & UW \\ UW & \longmapsto & BW \\ BW & \longmapsto & BT \\ Tc & \longmapsto & XBcc \\ BX & \longmapsto & BY \\ ZY & \longmapsto & ZB \\ ZB & \longmapsto & XB \\ aX & \longmapsto & aaT \\ aX & \longmapsto & aa \\ B & \longmapsto & b \\ BY & \longmapsto & ZY \end{array} \tag{15}$$

generates the language L. Theorem 2 can be used to prove that L is not a context-free language. Therefore L is a language of type 1.

2 Context-free grammars

2.1 Parse trees

PARSE TREES. Let $G = (V_T, V_N, S, P)$ be a context-free grammar. A parse tree for G is a tree constructed as follows:

- its root is labelled by the start-symbol S of G,
- each internal node is labeled by a non-terminal,
- the leaves are labeled by terminals or ε, the empty-word symbol,
- if an internal node is labeled by a non-terminal A and its children from left to right are labeled by grammar symbols $X_1, \ldots, X_n$ then

$$A \longmapsto X_1 \cdots X_n \tag{16}$$

  is a production of G. Moreover if $n = 1$ we can have $X_1 = \varepsilon$.

Let $\mathcal{T}$ be a parse tree. The string obtained by concatenating the labels of the leaves of $\mathcal{T}$ from left to right id called the YIELD of the parse tree.

Let w be a word of the language generated by G. Clearly there exists at least one parse tree for G whose yield is w. Such a tree is called a parse tree for w w.r.t. G.

LEFTMOST/RIGHTMOST DERIVATIONS. Let $G = (V_T, V_N, S, P)$ be a context-free grammar. The derivation $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ is LEFTMOST if for each α_i (with $i = 1 \cdots n-1$) the leftmost non-terminal is rewritten. Analogously we define rightmost derivations.

Remark 3 A parse tree does not specify the order in which productions are used to rewrite non-terminal symbols by strings of grammar symbols: a given parse tree usually represents several different derivations.

However, to any parse tree we can associate a unique leftmost (or a unique rightmost) derivation. *Top-down parsing* attempts to find a leftmost derivation for the input. In *bottom-up parsing*, we construct a rightmost derivation (in reverse order).

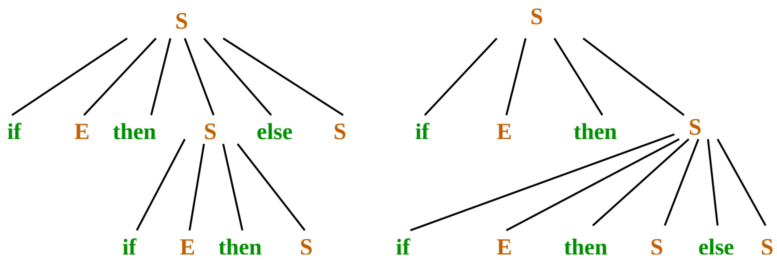

Figure 5: Two parse trees for the same if-then-else statement.

2.2 Elimination of ambiguity.

<u>AMBIGUITY.</u> The context-free grammar $G = (V_T, V_N, S, P)$ is

- *unambiguous* if every sentence of G has a unique parse tree,
- *ambiguous* if is not unambiguous.

A language L over some alphabet Σ is said *inherently ambiguous* if it cannot be generated by an unambiguous grammar.

Example 8 Let us consider $\Sigma = \{a, b, c\}$ and

$$L = \{w \in \Sigma^* \ \mid \ (\mid w|_a = \mid w|_b) \text{ or } (\mid w|_b = \mid w|_c)\}. \tag{17}$$

One can show that L is an inherently ambiguous language.

Example 9 Consider $V_N = \{S, E\}$, $V_T = \{\mathbf{if}, \mathbf{then}, \mathbf{else}\}$ and a grammar $G = (V_T, V_N, S, P)$ such that the following

$$S \ \longmapsto \ \mathbf{if}\ E\ \mathbf{then}\ S \mid \mathbf{if}\ E\ \mathbf{then}\ S\ \mathbf{else}\ S \tag{18}$$

are productions of G. Then the string

$$w \ = \ \mathbf{if}\ E\ \mathbf{then}\ \mathbf{if}\ E\ \mathbf{then}\ S\ \mathbf{else}\ S \tag{19}$$

has two parse trees as shown on Figure 5. In all programming languages with if-then-else statements of this form, the second parse tree is preferred. Hence the general rule is: match each **else** with the previous closest **then**. This disambiguating rule can be incorporated directly into a grammar by using the following observations.

- A statement appearing between a **then** and a **else** must be matched. (Otherwise there will be an ambiguity.)
- Thus statements must split into kinds: *matched* and *unmatched*.
- A *matched statement* is
  - either an if-then-else statement containing no unmatched statements
  - or any statement which is not an if-then-else statement and not an if-then statement.
- Then an *unmatched statement* is
  - an if-then statement (with no else-part)
  - an if-then-else statement where unmatched statements are allowed in the else-part (but not in the then-part).

$$
\begin{array}{lcl}
\mathit{stmt} & \longmapsto & \mathit{matched\text{-}stmt} \mid \mathit{unmatched\text{-}stmt} \\
\mathit{matched\text{-}stmt} & \longmapsto & \mathbf{if}\ \mathit{expr}\ \mathbf{then}\ \mathit{matched\text{-}stmt}\ \mathbf{else}\ \mathit{matched\text{-}stmt} \\
\mathit{matched\text{-}stmt} & \longmapsto & \mathit{non\text{-}alternative\text{-}stmt} \\
\mathit{unmatched\text{-}stmt} & \longmapsto & \mathbf{if}\ \mathit{expr}\ \mathbf{then}\ \mathit{stmt} \\
\mathit{unmatched\text{-}stmt} & \longmapsto & \mathbf{if}\ \mathit{expr}\ \mathbf{then}\ \mathit{matched\text{-}stmt}\ \mathbf{else}\ \mathit{unmatched\text{-}stmt}
\end{array}
$$

2.3 Elimination of left recursion

<u>LEFT RECURSION.</u> Let G be a context-free grammar. A production of G is said *left recursive* if it has the form

$$A \longmapsto A\alpha \tag{20}$$

where A is a nonterminal and α is a string of grammar symbols. A nonterminal A of G is said *left recursive* if there exists a string of grammar symbols α such that we have

$$A \stackrel{*}{\Rightarrow} A\alpha \tag{21}$$

Such derivation is also said *left recursive*. The grammar G is *left recursive* if it has at least one left recursive nonterminal.

Remark 4 Top-down parsing is one of the methods that we will study for generating parse trees. This method cannot handle left recursive grammars. We present now an algorithm for transforming a left recursive grammar G into a grammar G' which is not left recursive and which generates the same language as G.

<u>THE BASIC TRICK</u> is to replace the production

$$A \longmapsto A\alpha \mid \beta \quad \text{by} \quad \left\{ \begin{array}{lll} A & \longmapsto & \beta A' \\ A' & \longmapsto & \alpha A' \mid \varepsilon \end{array} \right. \tag{22}$$

where $\alpha \neq \varepsilon$ is assumed. Observe that this trick eliminates left recursive productions. Applying this trick is not enough to remove all derivations of the form $A \stackrel{*}{\Rightarrow} A\alpha$. However this trick is sufficient for many grammars.

Example 10 The following grammar which generates arithmetic expressions

$$\begin{array}{lll} E & \longmapsto & E+T \mid T \\ T & \longmapsto & T*F \mid F \\ F & \longmapsto & (E) \mid \mathbf{id} \end{array} \tag{23}$$

has two left recursive productions. Applying the above trick leads to

$$\begin{array}{lll} E & \longmapsto & TE' \\ E' & \longmapsto & +TE' \mid \varepsilon \\ T & \longmapsto & FT' \\ T' & \longmapsto & *FT' \mid \varepsilon \\ F & \longmapsto & (E) \mid \mathbf{id} \end{array} \tag{24}$$

<u>THE CASE OF SEVERAL LEFT RECURSIVE A-PRODUCTIONS.</u> Assume that the set of all A-productions has the form

$$A \longmapsto A\alpha_1 \mid A\alpha_2 \mid \cdots A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \beta_n \tag{25}$$

where no β_i begins with an A and where $\alpha_i \neq \varepsilon$ holds for every $i = 1 \cdots m$. Then we repalce these A-productions by

$$\begin{array}{lll} A & \longmapsto & \beta_1 A' \mid \beta_2 A' \mid \cdots \beta_n A' \\ A' & \longmapsto & \alpha_1 A' \mid \alpha_2 A' \mid \cdots \alpha_m A' \mid \varepsilon \end{array} \tag{26}$$

Remark 5 Let us consider the following grammar.

$$\begin{array}{lcl} S & \longmapsto & Aa \mid b \\ A & \longmapsto & Ac \mid Sd \mid \varepsilon \end{array} \tag{27}$$

The nonterminal S is left recursive since we have

$$S \Rightarrow Aa \Rightarrow Sda \tag{28}$$

However there is no left recursive S-productions. We show now how to deal with these left recursive derivations.

<u>Removing all left recursive derivations.</u> Let us assume that nonterminals are numbered: $A_1, A_2, \ldots, A_n$. The goal is to remove any possible circuit for all $1 \leq j < i \leq n$

$$A_i \overset{*}{\Rightarrow} A_j \delta_j \overset{*}{\Rightarrow} A_i \delta_i. \tag{29}$$

The idea is for $i = 1 \cdots n$

- To make sure that every A_i-production does not have a form $A_i \longmapsto A_j \delta$ for some $j < i$.
- To remove any left recursive A_i-production.

The method in more detail:

- remove all left recursive A_1-productions (by the above trick)
- remove A_1 from the right-hand side of each A_2-production of the form $A_2 \longmapsto A_1 \delta$ (by applying all A_1-productions)
- remove all left recursive A_2-productions
- remove A_j from the right-hand side of each A_3-production of the form $A_3 \longmapsto A_j \delta$ for $j = 1, 2$.
- remove all left recursive A_3-productions
- ...
- remove A_j from the right-hand side of each A_i-production of the form $A_i \longmapsto A_j \delta$ (for every j such that $1 \leq j < i \leq n$)

- remove all left recursive A_i-productions
- ...

Observations:

- To remove A_j from the right-hand side of the A_i-production $A_i \longmapsto A_j\delta$ (where $1 \leq j < i \leq n$) replace

  $$A_i \longmapsto A_j\delta \;\text{ by }\; A_i \longmapsto \gamma_1\delta \;|\; \cdots \;|\; \gamma_k\delta \;\text{ where }\; A_j \longmapsto \gamma_1 \;|\; \cdots \;|\; \gamma_k \tag{30}$$

  are all A_j-productions and $\gamma_i \neq \varepsilon$ for $i = 1 \cdots k$.

- This construction may not work if G constains initially a production of the form $A \longmapsto \varepsilon$. See Example 11. However we can always remove such productions. Explain how!

- Also, this construction may fail if the grammar G has a cycle, that is, if there exists a nonterminal A such that $A \overset{*}{\Rightarrow} A$. This can happen with the following grammar fragment

  $$\begin{array}{lcl} A_1 & \longmapsto & A_2 \;|\; \gamma_1 \\ A_2 & \longmapsto & A_1 \;|\; \gamma_2 \end{array} \tag{31}$$

  where

  - γ_1 is a string of grammar symbols not starting with A_1 (what we can assume by application of the basic trick given in Relation 22)
  - γ_2 is a string of grammar symbols not starting with A_1 (what we can assume by application of the A_1-productions).

  Then we can replace the above fragment by

  $$\begin{array}{lcl} A_1 & \longmapsto & \gamma_2 \;|\; \gamma_1 \\ A_2 & \longmapsto & \gamma_1 \;|\; \gamma_2 \end{array} \tag{32}$$

  without changing the generated language. Unfortunately a cycle may be hidden in a more subtle manner. Indeed γ_2 may write $\gamma_{21}A_2\gamma_{22}$ and γ_1 may write $\gamma_{11}A_1\gamma_{12}$ with $\gamma_{21} \overset{*}{\Rightarrow} \varepsilon$, $\gamma_{22} \overset{*}{\Rightarrow} \varepsilon$, $\gamma_{11} \overset{*}{\Rightarrow} \varepsilon$ and $\gamma_{12} \overset{*}{\Rightarrow} \varepsilon$. However this may only happen if there are productions of the form $A \longmapsto \varepsilon$.

- In conclusion, the above construction for removing all left recursive derivations should be performed after
  - removing all productions of the form $A \longmapsto \varepsilon$,
  - removing cycles, that is derivations of the form $A \stackrel{*}{\Rightarrow} A$.

  Now observe that the basic trick given in Relation 22 introduces productions of the form $A \longmapsto \varepsilon$. However one can show that the above construction cannot reintroduce cycles.

Example 11 Consider the following grammar.

$$\begin{array}{lcl} A_3 & \longmapsto & A_2A_1 \\ A_2 & \longmapsto & \varepsilon \\ A_1 & \longmapsto & A_2A_1 \end{array} \tag{33}$$

Applying the previous algorithm for removing all possible left-recursive derivations fail on this grammar, because of the ε-production.

Example 12 Consider again the following grammar.

$$\begin{array}{lcl} S & \longmapsto & Aa \mid b \\ A & \longmapsto & Ac \mid Sd \mid \varepsilon \end{array} \tag{34}$$

We order the nonterminals: $S < A$. There is no left recursive S-productions. So the next step is to remove S from the right-hand side of the A-productions of the form $A \longmapsto S\alpha$. Hence we obtain

$$\begin{array}{lcl} S & \longmapsto & Aa \mid b \\ A & \longmapsto & Ac \mid Aad \mid bd \mid \varepsilon \end{array} \tag{35}$$

Then we remove the left recursive A-productions.

$$\begin{array}{lcl} S & \longmapsto & Aa \mid b \\ A & \longmapsto & bdA' \mid A' \\ A' & \longmapsto & cA' \mid adA' \mid \varepsilon \end{array} \tag{36}$$

2.4 Left factoring

Left factoring is another useful grammar transformation used in parsing. The general ideal is to replace the productions

$$A \longmapsto \alpha\beta_1 \mid \cdots \alpha\beta_n \mid \gamma \quad \text{by} \quad \begin{cases} A & \longmapsto \alpha A' \mid \gamma \\ A' & \longmapsto \beta_1 \mid \cdots \mid \beta_n \end{cases} \tag{37}$$

where

- A, A' are nonterminals,
- $\alpha, \beta_1, \ldots, \beta_n$ are strings of symbols with $\alpha \neq \varepsilon$,
- γ represents all alternatives for A-productions that do not start with α.

Example 13 Let us consider the following grammar:

$$\begin{array}{lcl} S & \longmapsto & \textbf{if } E \textbf{ then } S \mid \textbf{if } E \textbf{ then } S \textbf{ else } S \mid a \\ E & \longmapsto & b \end{array} \tag{38}$$

By left factoring we obtain

$$\begin{array}{lcl} S & \longmapsto & \textbf{if } E \textbf{ then } SS' \mid a \\ S' & \longmapsto & \textbf{else } S \mid \varepsilon \\ E & \longmapsto & b \end{array} \tag{39}$$

3 Parsing

Parsing is the process of determining if a string of tokens can be generated by a grammar. It is helpful to think of a parse tree being constructed (even though a compiler may not actually construct such tree).

<u>Efficiency considerations.</u>

- A parser can be constructed from any gammar.
- However grammars used in practice have a special form.
- For any context-free grammar there is a parser that takes at most $\mathcal{O}(n^3)$ time to parse a string of n tokens.

- But in fact there are linear algorithms to parse all languages that arise in practice.

- Programming language parsers almost always make a single left-to-right scan over the input, looking ahead one token at a time

METHODS. Most parsing methods fall into one of the two following classes

Top-down parsing.

- Construction starts at the root and proceeds towards the leaves.
- Efficient top-down parsers can be constructed by hands.

Bottom-up parsing.

- Construction starts at the leaves and proceeds towards the root.
- Bottom-up parsers can handle a larger class of grammars.
- Software for generating parsers tend to produce bottom-up parsers.

3.1 Top-down parsing

The construction of the parse tree relies on the following principles.

(1) Create a tree reduced to the start-symbol of the grammar. The current token being scanned is called the *lookahead symbol*. Initially, the lookahead symbol is the leftmost token of the input string.

(2) Repeat

  (2.1) If the leftmost leaf L of the current tree is labeled by a nonterminal A then

    (2.1.1) select an A-production $A \longmapsto \alpha$ and

    (2.1.2) create a parse tree for $A \longmapsto \alpha$ rooted at L.

    (2.1.3) Advance to the leftmost leaf in the tree.

  (2.2) Otherwise if the terminal at node L matches the lookahead symbol then

    (2.2.1) Advance to the next leftmost leaf in the tree.

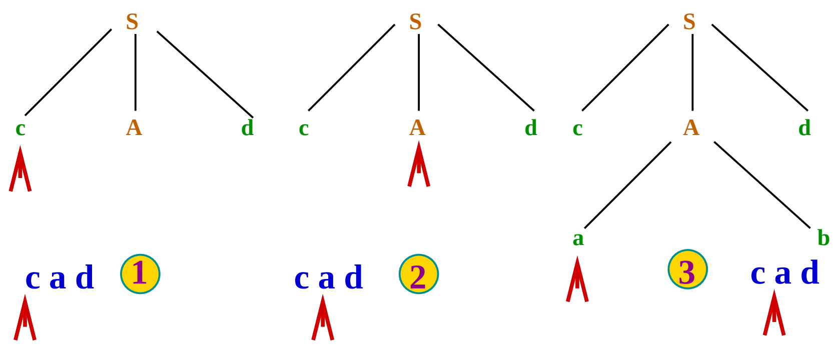

Figure 6: Sketch of the top-down parsing of $w = cad$ before conflict.

(2.2.2) Advance the lookahead symbol in the input string to the next token.

(2.3) Otherwise

(2.3.1) Back-track in the tree to the last nonterminal A for which several A-productions could be chosen.

(2.3.2) Choose another A-production and peforms steps (2.1.2) and (2.1.3).

(2.3.3) Adjust the lookahead symbol accordingly.

(3) Until the input string is read completely or the back-tracking process fails.

Example 14 Consider the following grammar G (with terminals a, b, c, d and nonterminals S, A)

$$\begin{array}{lcl} S & \longmapsto & cAd \\ A & \longmapsto & ab \mid a \end{array}$$

and the input string $w = cad$. Figures 6 and 7 show the building of a parse tree for w w.r.t. G.

Remark 6 A left-recursive grammar can cause an infinite loop: when we try to expand the nonterminal A we may be led to expand A without having consumed any input. Fortunetely, any left-recursive grammar G can be replaced by a grammar without left-recursion and generating the same language as G.

However avoiding backtraking requires more advanced techniques to be described in the next section.

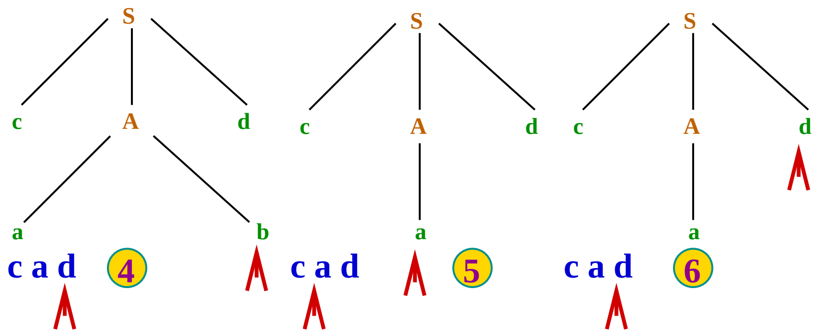

Figure 7: Sketch of the top-down parsing of $w = cad$: conflct, back-track and termination.

3.2 Predictive parsing

<u>The idea.</u> Predictive parsing relies on information about what first symbols can be generated by the right side of a production. The lookahead symbol guides the selection of the production $A \longmapsto \alpha$ to be used:

- if α starts with a token, then the production can be used when the lookahead symbol matches this token
- if α starts with a nonterminal B, then the production can be used if the lookahead symbol can be generated from B.

<u>The method.</u>

(1) For each nonterminal A for each production $A \longmapsto \alpha$ let $\mathrm{FIRST}(\alpha)$ be the subset of $V_T \cup \{\varepsilon\}$ defined by the following rules

- for every $a \in V_T$ we have

$$a \in \mathrm{FIRST}(\alpha) \quad \Longleftrightarrow \quad (\exists \beta \in (V_T \cup V_N)^*) \;\; \alpha \overset{*}{\Longrightarrow} a\beta \tag{40}$$

- $\varepsilon \in \mathrm{FIRST}(\alpha) \quad \Longleftrightarrow \quad \alpha \overset{*}{\Longrightarrow} \varepsilon$

In other words the token a belongs to $\mathrm{FIRST}(\alpha)$ iff there exists a string γ deriving from α such that a is the first symbol of γ. Moreover ε belongs to $\mathrm{FIRST}(\alpha)$ iff ε derives from α.

(2) Consider the following procedure in pseudo-code of Algorithm 1.

Algorithm 1

```
procedure match(a: token)
  if lookahead = a
    then lookahead = nexttoken
    else error
```

So *match*(a) moves the cursor *lookahead* one symbol forward iff *lookahead* points to a. Otherwise an error is produced.

(3) A procedure *proc_A* (given by Algorithm 2) is associated with every nonterminal A.

Algorithm 2

```
procedure proc_A
  for every A ⟼ α repeat
    if lookahead ∈ FIRST(α)
      then
        let α = X_1 X_2 ··· X_ℓ where each X_i is a grammar symbol
        for i = 1 ··· ℓ repeat
          if terminal?(X_i) then match(X_i) else proc_X_i
        return
  if A ⟼ ε then return else error
```

In other words, given a nonterminal A the call *proc_A* chooses an A-production $A \longmapsto \alpha$ such that *lookahead* belomgs to $\mathrm{FIRST}(\alpha)$. If such a production exists then for each symbol X in α (reading α from left to right)

- if X is a terminal, then call *match*(X)
- if X is a nonterminal, then call *proc_X*

If no A-production $A \longmapsto \alpha$ satisfies *lookahead* $\in \mathrm{FIRST}(\alpha)$ then either $A \longmapsto \varepsilon$ is a production and the procedure terminates normally, or an error is produced.

(4) Initially *lookahead* is the first symol of the input string and we run *procS* where S is the start symbol.

Observations:

- In Algorithm 2 **return** is an escape statement without error whereas **error** is an interruption with error.
- The grammar must have no left recursive derivations, otherwise the parsing may lead to an infinite loop.
- Back-tracking is avoided provided that for every nonterminal A and for every string of symbols α, β

$$\left\{ \begin{array}{lcl} A & \longmapsto & \alpha \\ A & \longmapsto & \beta \end{array} \right. \quad \Rightarrow \quad FIRST(\alpha) \cap FIRST(\beta) \ = \ \emptyset. \qquad (41)$$

- The last statement of Algorithm 2 means that the A-production $A \longmapsto \varepsilon$ **can be chosen** if the lookahead symbol is not in any of the $\mathrm{FIRST}(\alpha)$ for $A \longmapsto \alpha$ with $\alpha \neq \varepsilon$.

Hint: You should understand the production $A \longmapsto \varepsilon$ as a way to say that the nonterminal A can be canceled (if no other A-production is convenient to use).

Example 15 Consider the following grammar (with terminals a, b, c, d and nonterminals S, A)

$$\begin{array}{lcl} S & \longmapsto & cAd \\ A & \longmapsto & ab \mid a \end{array}$$

Here we have $\mathrm{FIRST}(cAd) = \{c\}$, $\mathrm{FIRST}(ab) = \{a\}$ and $\mathrm{FIRST}(a) = \{a\}$. Unfortunately, if we have $lookahead = a$ we cannot tell each A-production to use. So left factoring would be needed to process this example further (what we will not do).

Example 16 The following grammar generates a subset of the types in the PASCAL language.

$$\begin{array}{lcl} type & \longmapsto & simple \mid \uparrow\mathbf{id} \mid \mathbf{array}[simple] \ \mathbf{of} \ type \\ simple & \longmapsto & \mathbf{integer} \mid \mathbf{char} \mid \mathbf{num\ dotdot\ num} \end{array}$$

where

- all bold names and the characters $\uparrow$,, [,] are tokens,

- *type* and *simple* are nonterminals,
- **dotdot** stands for $\cdots$
- and **num** for a type of *small integers*.

Here's a string of tokens generated by the above grammar.

$$\textbf{array}\ [\ \textbf{num dotdot num}\ ]\ \textbf{of integer}$$

We give now the FIRST sets of every left side of the above productions.

$$\begin{array}{rcl}
\mathrm{FIRST}(\mathit{simple}) & = & \{\ \textbf{integer}, \textbf{char}, \textbf{num}\ \} \\
\mathrm{FIRST}(\uparrow\textbf{id}) & = & \{\ \uparrow\ \} \\
\mathrm{FIRST}(\textbf{array}[\mathit{simple}]\ \textbf{of}\ \mathit{type}) & = & \{\ \textbf{array}\ \} \\
\mathrm{FIRST}(\textbf{integer}) & = & \{\ \textbf{integer}\ \} \\
\mathrm{FIRST}(\textbf{char}) & = & \{\ \textbf{char}\ \} \\
\mathrm{FIRST}(\textbf{num dotdot num}) & = & \{\ \textbf{num}\ \}
\end{array}$$

Algorithms 3 and 4 define the procedures *proc_simple* and *proc_type*.

Algorithm 3

```
procedure proc_type
  if lookahead ∈ { integer, char, num } then
    proc_simple
  else
    if lookahead = ↑ then
        match(↑);
        match(id)
    else
        if lookahead = array then
          match(array);
          match([);
          proc_simple;
          match(]);
          match(of);
          proc_type
        else
          error
```

Algorithm 4

```
procedure proc_simple
  if lookahead = integer then
    match(integer)
  else
    if lookahead = char then
      match(char)
    else
      if lookahead = num then
        match(num);
        match(dotdot);
        match(num)
      else
        error
```

3.3 Non-recursive implementation of predictive parsing

THE IDEA. Predictive parsing can be performed using a pushdown stack, avoiding recursive calls.

- Initially the stack holds just the start symbol of the grammar.
- At each step a symbol X is popped from the stack:
  - if X is a terminal symbol then it is matched with *lookahead* and *lookahead* is advanced,
  - if X is a nonterminal, then using *lookahead* and a *parsing table* (implementing the FIRST sets) a production is chosen and its right hand side is pushed onto the stack.
- This process goes on until the stack and the input string become empty. It is useful to have an *end_of_stack* and an *end_of_input* symbols. We denote them both by $.

Figure 8 shows the structure of non-recursive predictive parsers. For each nonterminal A and each token a the entry $M[A, a]$ of the parsing table contains either an A-production generating sentences starting with a or an *error-entry*.

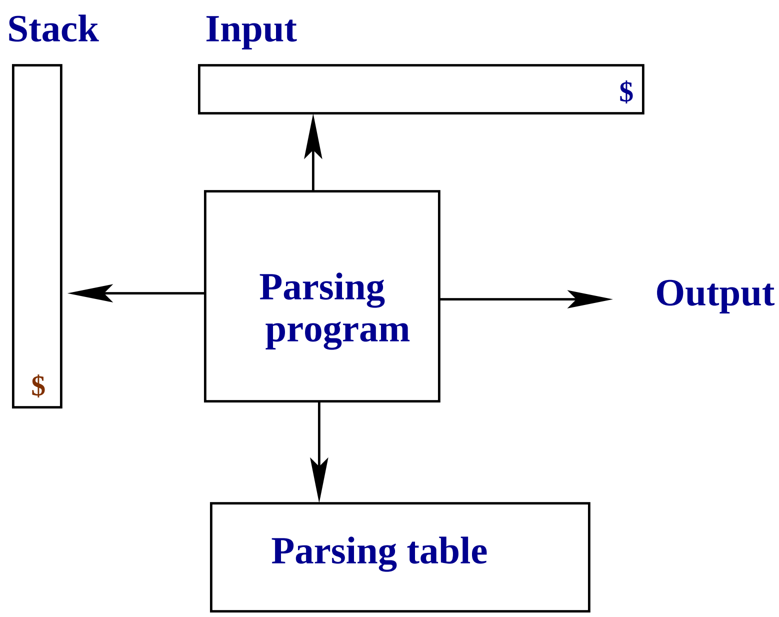

Figure 8: The structure of non-recursive predictive parsers.

Example 17 Consider the grammar G given by:

$$
\begin{array}{lcl}
S & \longmapsto & aAa \mid BAa \mid \varepsilon \\
A & \longmapsto & cA \mid bA \mid \varepsilon \\
B & \longmapsto & b
\end{array}
$$

The parsing table is:

	a	b	c	$\$$
S	$S \longmapsto aAa$	$S \longmapsto BAa$		$S \longmapsto \varepsilon$
A	$A \longmapsto \varepsilon$	$A \longmapsto bA$	$A \longmapsto cA$	
B		$B \longmapsto b$		

where the empty slots correspond to error-entries. Consider the parsing of the word $w = bcba$.

Stack	Remaining input	action
$\$S$	$bcba\$$	choose $S \longmapsto BAa$
$\$aAB$	$bcba\$$	choose $B \longmapsto b$
$\$aAb$	$bcba\$$	match b
$\$aA$	$cba\$$	choose $A \longmapsto cA$
$\$aAc$	$cba\$$	match c
$\$aA$	$ba\$$	choose $A \longmapsto bA$
$\$aAb$	$ba\$$	match b
$\$aA$	$a\$$	choose $A \longmapsto \varepsilon$
$\$a$	$a\$$	match a

THE ALGORITHM.

Algorithm 5

Initial configuration: the stack contains $\boxed{\$S}$ where S is the start-symbol and the input string is $\boxed{w\$}$ where w is a word over the tokens.

Exit: with an error iff w is not generated by the grammar.

Output: one by one the production that generates w from S.

set ip to the first symbol of $w\$$
repeat
  let X be the top of the stack;
  let a be the symbol pointed by ip;
  if $X \in V_T$ **or** $X = \$$ **then**
    if $X = a$ **then**
      pop the stack;
      advance ip;
    else error
  else
    let $X \longmapsto Y_1Y_2 \cdots Y_k$ be $M[X, a]$;
    if no such production **then error**;
    pop X from the stack;
    push $Y_k, Y_{k-1}, \cdots, Y_1$ onto the stack;
    output $X \longmapsto Y_1Y_2 \cdots Y_k$;
until $X = \$$

<u>COMPUTING THE FIRST SETS.</u> Recall that for any string α of symbols the set $\mathrm{FIRST}(\alpha)$ satisfies the following conditions for every terminal a and every string of symbols β

- $\mathrm{FIRST}(\alpha) \subseteq V_T \cup \{\varepsilon\}$
- $\alpha \stackrel{*}{\Longrightarrow} a\beta$ iff $a \in \mathrm{FIRST}(\alpha)$
- $\alpha \stackrel{*}{\Longrightarrow} \varepsilon$ iff $\varepsilon \in \mathrm{FIRST}(\alpha)$.

For a symbol $X \in V_T \cup V_N$ the set $\mathrm{FIRST}(X)$ can be computed as follows

Algorithm 6

Input: $X \in V_T \cup V_N$.

Output: $\mathrm{FIRST}(X)$

```
if X ∈ V_T then
      FIRST(X) := {X}
else
      if X ⟼ ε then
         FIRST(X) := {ε}
      else
         FIRST(X) := ∅
for every X ⟶ α with α ≠ ε repeat
      let α = X_1 X_2 ··· X_ℓ where each X_i is a grammar symbol
      for i = 1 ··· ℓ repeat
            FIRST(X) := (FIRST(X_i) \{ε}) ∪ FIRST(X)
      until ε ∉ FIRST(X_i)
      if ε ∈ ⋂_{1≤i≤ℓ} FIRST(X_i) then
            FIRST(X) := FIRST(X) ∪ {ε}
```

Comments about the computation of $\mathrm{FIRST}(X)$ with Algorithm 6.

- The case where X is a terminal symbol is trivial.
- Assume from now on that X is a nonterminal. The case where $X \longmapsto \varepsilon$ follows immediately from the specifications of $\mathrm{FIRST}(X)$.

- Assume that there is a production of the form $X \longmapsto X_1X_2\cdots X_k$ where $X_1, X_2, \ldots X_k$ are grammar symbols. If $\varepsilon \notin \mathrm{FIRST}(X_1)$ then the first letter of a word generated from $X_1X_2\cdots X_k$ is the first letter of a word generated from X_1 and thus $\mathrm{FIRST}(X) = \mathrm{FIRST}(X_1)$. If $\varepsilon \in \mathrm{FIRST}(X_1)$ but $\varepsilon \notin \mathrm{FIRST}(X_2)$ then the first letter of a word generated from $X_1X_2\cdots X_k$ is either the first letter of a word generated from X_1 or the first letter of a word generated from X_2. Etc ... This explains the nested `for` loop.

- The last **if** statement tells that if ε belongs to all $\mathrm{FIRST}(X_i)$ then ε must be in $\mathrm{FIRST}(X)$. This statement is necessary since the nested `for` loop **cannot** add ε to $\mathrm{FIRST}(X)$ even if ε belongs to all $\mathrm{FIRST}(X_i)$.

Algorithm 7

Input: $X = X_1X_2\cdots X_n$ where every $X_i \in V_T \cup V_N$.

Output: $\mathrm{FIRST}(X_1X_2\cdots X_n)$.

$\mathrm{FIRST}(X) := \emptyset$
for $i = 1\cdots n$ **repeat**
  $\mathrm{FIRST}(X) := (\mathrm{FIRST}(X_i) \setminus\{\varepsilon\}) \cup \mathrm{FIRST}(X)$
until $\varepsilon \notin \mathrm{FIRST}(X_i)$
if $\varepsilon \in \bigcap_{1\leq i\leq n}\mathrm{FIRST}(X_i)$ **then**
  $\mathrm{FIRST}(X) := \mathrm{FIRST}(X) \cup \{\varepsilon\}$

The principle of Algorithm 7 is similar to that of Algorithm 6.

- If $\varepsilon \notin \mathrm{FIRST}(X_1)$ then the first letter of a word generated from $X_1X_2\cdots X_k$ must be the first letter of a word generated from X_1.

- If $\varepsilon \in \mathrm{FIRST}(X_1)$ but $\varepsilon \notin \mathrm{FIRST}(X_2)$ then the first letter of a word generated from $X_1X_2\cdots X_k$ is either the first letter of a word generated from X_1 or the first letter of a word generated from X_2.

- Etc ...

- ε belongs to $\mathrm{FIRST}(X_1X_2\cdots X_k)$ iff it belongs to each $\mathrm{FIRST}(X_i)$.

<u>THE FOLLOW SETS.</u> In order to give an algorithm for building the parsing table we define for every $A \in V_N$

$$\text{FOLLOW}(A) \;=\; \left\{ a \in V_T \cup \{\$\} \;\mid\; S\$ \overset{*}{\Longrightarrow} \alpha A a r \;\mid\; \left\{ \begin{array}{l} \alpha \in (V_T \cup V_N)^* \\ r \in (V_T \cup V_N)^* \cup \{\$\,\varepsilon\} \end{array} \right. \right\} \tag{42}$$

In other words FOLLOW(A) is the set of the terminals that can appear immediately to the right of the nonterminal A in some sentential form. Moreover \$ belongs to FOLLOW(A) if A is the rightmost symbol in some sentential form.

<u>COMPUTING THE FOLLOW SETS.</u>

Algorithm 8

Input: $G = (V_T, V_N, S, P)$.

Output: FOLLOW(A) for $A \in V_N$.

```
for every A ∈ V_N repeat FOLLOW(A) := ∅
FOLLOW(S) := {$}
done := false
while done = false repeat
    building := false
    for every A ∈ V_N repeat
        for every B ∈ V_N repeat
            for every (A ⟼ αBβ) ∈ P repeat
                n := | FOLLOW(B) |
                FOLLOW(B) := FIRST(β) \ {ε} ∪ FOLLOW(B)
                if β ⟹* ε then
                    FOLLOW(B) := FOLLOW(B) ∪ FOLLOW(A)
                if | FOLLOW(B) | > n then
                    building := true
            for every (A ⟼ αB) ∈ P repeat
                n := | FOLLOW(B) |
                FOLLOW(B) := FOLLOW(B) ∪ FOLLOW(A)
                if | FOLLOW(B) | > n then
                    building := true
    if building = false then
        done := true
```

The FOLLOW sets of all nonterminal symbols are computed *together* by the following process:

- Initially all these sets are empty, except FOLLOW(S).
- There exist three rules that can increase FOLLOW(B) for a given nonterminal B. These rules are:
  1. if $(A \longmapsto \alpha B\beta)$ then $\text{FOLLOW}(B) := \text{FIRST}(\beta)\backslash\{\varepsilon\} \cup \text{FOLLOW}(B)$
  2. if $(A \longmapsto \alpha B)$ then $\text{FOLLOW}(B) := \text{FOLLOW}(B) \cup \text{FOLLOW}(A)$
  3. if $(A \longmapsto \alpha B\beta)$ and $\beta \overset{*}{\Longrightarrow} \varepsilon$ then $\text{FOLLOW}(B) := \text{FOLLOW}(B)$ $\cup\ \text{FOLLOW}(A)$
- Let us call a *pass* the fact of trying to apply each rule to each nonterminal.
- Then the algorithm scheme can be stated as follows: **repeat** perform a pass **until** this pass does not change any of the FOLLOW sets.
- To implement this process in Algorithm 8 we use two boolean auxiliary variables:
  - *done* which becomes **true** when a pass could not increase any of the FOLLOW sets.
  - *building* which becomes **true** during a pass if at least one FOLLOW set could be increased.
- In the pseudo-code of Algorithm 8 the value $\mid \text{FOLLOW}(B) \mid$ denotes the number of elements of the set FOLLOW(B).
- One could wonder whether Algorithm 8 could run forever. But it is easy to see that the process has to stop. Indeed, each FOLLOW set contains at most $t+1$ symbols where t is the number of terminals. So the number of passes is at most $n(t+1)$ where n is the number of nonterminals (since each successful pass increases at least by one the number of elements of at least one FOLLOW set).

Example 18 Consider the following grammar (with terminals a, b, c and nonterminals S, A)

$$
\begin{array}{lcl}
S & \longmapsto & aAa \mid bAa \mid \varepsilon \\
A & \longmapsto & cA \mid bA \mid \varepsilon
\end{array}
$$

The FIRST sets of the right side of the productions are given by

$$
\begin{array}{lcl}
\text{FIRST}(aAa) & = & \{a\} \\
\text{FIRST}(bAa) & = & \{b\} \\
\text{FIRST}(cA) & = & \{c\} \\
\text{FIRST}(bA) & = & \{b\} \\
\text{FIRST}(\varepsilon) & = & \{\varepsilon\}
\end{array}
$$

The FOLLOW sets of the left side of the productions are given by

$$
\begin{array}{lcl}
\text{FOLLOW}(S) & = & \{\$\} \\
\text{FOLLOW}(A) & = & \{a\}
\end{array}
$$

Computing the parsing table.

Algorithm 9

Input: $G = (V_T, V_N, S, P)$.

Output: Parsing table M.

```
for every A ∈ V_N repeat
    for every a ∈ V_T ∪ {$} repeat
      M[A, a] := ∅
for every (A ⟼ α) ∈ P repeat
    for every a ∈ FIRST(α) repeat
        if a≠ε then
            M[A, a] := M[A, a] ∪ {A ⟼ α}
    if ε ∈ FIRST(α) then
        for every b ∈ FOLLOW(A) repeat
            M[A, b] := M[A, b] ∪ {A ⟼ α}
for every A ∈ V_N repeat
    for every a ∈ V_T ∪ {$} repeat
        if M[A, a] = ∅ then
            M[A, a] := error
```

Algorithm 9 consists of three main steps.

- The first **for** loop which initializes each entry of the parsing table M to the empty set.
- The second **for** loop which fills the parsing table M by using the following rules.
  1. If $A \longmapsto \alpha$ with $\alpha \neq \varepsilon$ is a production and if a is a terminal such that $a \in \mathrm{FIRST}(\alpha)$ then the production $A \longmapsto \alpha$ is added to $M[A, a]$.
  2. If $\varepsilon \in \mathrm{FIRST}(\alpha)$ (which means $\alpha \stackrel{*}{\Longrightarrow} \varepsilon$) then the production $A \longmapsto \alpha$ is added to $M[A, b]$ for every $b \in \mathrm{FOLLOW}(A)$. In particular if $\varepsilon \in \mathrm{FIRST}(\alpha)$ and $\$ \in \mathrm{FOLLOW}(A)$ then the production $A \longmapsto \alpha$ is added to $M[A, \$]$.
- The third **for** loop which sets to error every empty entry of the parsing table M.

Example 19 Consider the following grammar (with terminals $+, *, (,), \mathbf{id}$ and nonterminals T, E, E', F)

$$\begin{array}{lcl}
E & \longmapsto & TE' \\
E' & \longmapsto & +TE' \mid \varepsilon \\
T & \longmapsto & FT' \\
T' & \longmapsto & *FT' \mid \varepsilon \\
F & \longmapsto & (E) \mid \mathbf{id}
\end{array}$$

The FIRST sets of the right side (and some of their prefixes) of the productions are given by

$$\begin{array}{lcl}
\mathrm{FIRST}(F) & = & \{(, \mathbf{id}\} \\
\mathrm{FIRST}(T) & = & \{(, \mathbf{id}\} \\
\mathrm{FIRST}(T') & = & \{*, \varepsilon\} \\
\mathrm{FIRST}(E) & = & \{(, \mathbf{id}\} \\
\mathrm{FIRST}(E') & = & \{+, \varepsilon\} \\
\mathrm{FIRST}(TE') & = & \{(, \mathbf{id}\} \\
\mathrm{FIRST}(FT') & = & \{(, \mathbf{id}\} \\
\mathrm{FIRST}((E)) & = & \{(\} \\
\mathrm{FIRST}(+TE') & = & \{+\} \\
\mathrm{FIRST}(*FT') & = & \{*\}
\end{array}$$

The FOLLOW sets of the left side of the productions are given by

$$
\begin{array}{lcl}
\text{FOLLOW}(E) & = & \{), \$\} \\
\text{FOLLOW}(E') & = & \{), \$\} \\
\text{FOLLOW}(T) & = & \{+,), \$\} \\
\text{FOLLOW}(T') & = & \{+,), \$\} \\
\text{FOLLOW}(F) & = & \{+, *,), \$\}
\end{array}
$$

	id	$+$	$*$	(	)	\$
E	$E \longmapsto TE'$			$E \longmapsto TE'$		
E'		$E' \longmapsto +TE'$			$E' \longmapsto \varepsilon$	$E' \longmapsto \varepsilon$
T	$T \longmapsto FT'$			$T \longmapsto FT'$		
T'		$T' \longmapsto \varepsilon$	$T' \longmapsto *FT'$		$T' \longmapsto \varepsilon$	$T' \longmapsto \varepsilon$
F	$F \longmapsto \mathbf{id}$			$F \longmapsto (E)$		

Example 20 Consider the following grammar (with terminals a, b, e, i, t and nonterminals S, S', E)

$$
\begin{array}{lcl}
S & \longmapsto & i\,E\,t\,S\,S' \;\mid\; a \\
S' & \longmapsto & e\,S \;\mid\; \varepsilon \\
E & \longmapsto & b
\end{array}
$$

The FIRST sets of the right side of the productions are given by

$$
\begin{array}{lcl}
\text{FIRST}(i\,E\,t\,S\,S') & = & \{i\} \\
\text{FIRST}(a) & = & \{a\} \\
\text{FIRST}(e\,S) & = & \{e\} \\
\text{FIRST}(\varepsilon) & = & \{\varepsilon\} \\
\text{FIRST}(b) & = & \{b\}
\end{array}
$$

Le's sketch the computation of the FOLLOW sets. Recall that the rules are:

$$
\begin{array}{lcl}
\text{if} & A \longmapsto \alpha B\beta & \text{then} \quad \text{FOLLOW}(B) := \text{FIRST}(\beta) \setminus \{\varepsilon\} \cup \text{FOLLOW}(B) \\
\text{if} & A \longmapsto \alpha B & \text{then} \quad \text{FOLLOW}(B) := \text{FOLLOW}(B) \cup \text{FOLLOW}(A) \\
\text{if} & \left\{ \begin{array}{l} A \longmapsto \alpha B\beta \\ \beta \stackrel{*}{\Longrightarrow} \varepsilon \end{array} \right. & \text{then} \quad \text{FOLLOW}(B) := \text{FOLLOW}(B) \cup \text{FOLLOW}(A)
\end{array}
\tag{43}
$$

	Initialization	$S \longmapsto iEtSS'$	$S \longmapsto iEtSS'$
FOLLOW(S)	$\{\$\}$	$\{e, \$\}$	$\{e, \$\}$
FOLLOW(S')	$\emptyset$	$\emptyset$	$\{e, \$\}$
FOLLOW(E)	$\emptyset$	$\{t\}$	$\{t\}$

This leads to the following parsing table.

	a	b	e	i	$\$$	t
S	$S \longmapsto a$			$S \longmapsto iEtSS'$		
S'			$S' \longmapsto \varepsilon$ $S' \longmapsto eS$		$S' \longmapsto \varepsilon$	
E		$E \longmapsto b$				

The entry $M[S', e]$ contains both

- $S' \longmapsto eS$ (since $e \in \mathrm{FIRST}(e\,S)$)
- $S' \longmapsto \varepsilon$ (since $\varepsilon \in \mathrm{FIRST}(\varepsilon)$ and since $e \in \mathrm{FOLLOW}(S')$)

We know that this grammar is ambiguous and this ambiguity is shown by this choice of productions when an e (**else**) is seen.

Error recovery in predictive parsing. A predictive parser attempts to match the nonterminals and the terminals in the stack with the remaining input. Therefore two types of conflicts can occur.

T-conflict. A terminal appearing on top of the stack does not match the following input token.

N-conflict. For a nonterminal B on top of the stack and the *lookahead* token b the entry $M[B, b]$ of the parsing table is empty.

Panic-mode recovery is based on the idea of skipping symbols on the input string until a token in a selected set of *synchronizing tokens* appears. These *synchronizing sets* should be chosen such that the parser recovers quickly from errors that are likely to occur in practice. Here are some strategies for the above conflict cases.

T-conflict. Skip (= ignore and advance) the token in the input string. Hence the synchronizing set conists here of all other tokens.

N-conflict. Possible solutions.

1. Skip (= ignore and advance) the token b in the input string.

2. If $M[B, b]$ is a blank entry labeled *synch* then skip (= pop and ignore) the nonterminal B. (Strictly speaking and according to the above definition, this is not a panic-mode recovery, but quite close in the spirit.)
3. Skip tokens from the input string until an element of FIRST(B) is reached, then continue parsing normally. So the synchronizing set here is FIRST(B).
4. Skip tokens from the input string until an element of FOLLOW(B) is reached, then skip B and continue parsing normally. (Again this is a variation of panic-mode recovery.)
5. Delimiters such as ; in C can be added to the two previous synchronizing sets.

3.4 $LL(1)$ Grammars

$LL(1)$ GRAMMARS AND LANGUAGES. A context-free grammar $G = (V_T, V_N, S, P)$ whose parsing table has no multiple entries is said to be $LL(1)$. In the name $LL(1)$,

- the first L stands for scanning the input from **l**eft to right,
- the second L stands for producing a **l**eftmost derivation,
- and the 1 stands for using **one** input symbol of lookahead at each step to make parsing action decision.

A language is said to be $LL(1)$ if it can be generated by a $LL(1)$ grmmar. It can be shown that $LL(1)$ grammars are

- not ambiguous and
- not left-recursive.

Moreover we have the following theorem to characterize $LL(1)$ grammars and show their importance in practice.

Theorem 3 *A context-free grammar $G = (V_T, V_N, S, P)$ is $LL(1)$ if and if only if for every nonterminal A and every strings of symbols α, β such that $\alpha \neq \beta$ and $A \longmapsto \alpha \mid \beta$ we have*

1. FIRST$(\alpha) \cap$ FIRST$(\beta) = \emptyset$,
2. *if* $\alpha \overset{*}{\Longrightarrow} \varepsilon$ *then* FIRST$(\beta) \cap$ FOLLOW$(A) = \emptyset$.

$LL(k)$ PARSING TABLES. The notion of FIRST sets can be generalized to describe the first k tokens of a string (with $k \geq 1$). This leads to $LL(k)$ parsing tables where

- the rows are labelled by non-terminals and
- we have a column for every sequence of k terminals.

This is rarely done in practice since these tables are so large.

$LL(k)$ GRAMMARS. Grammars parsable with $LL(k)$ parsing tables are called *$LL(k)$ grammars*. The $LL(k)$ grammars have the following properties.

- If G is a $LL(k)$ grammar then G is also a $LL(k+1)$ grammar (for every $k \geq 1$).
- For every $k \geq 1$, if G is a $LL(k)$ grammar the G is not ambiguous.

Remark 7 Here are some practical observations on the use of $LL(1)$ grammars and predictive parsing.

- The main difficulty in using predictive parsing is in writing a $LL(1)$ grammar for the source language.
- The parser of a compiler often uses
  - predictive parsing methods for control constructs and
  - bottom-up parsing methods for expressions.

4 YACC – Yet Another Compiler Compiler

YACC generates a syntax analyzer (parser) for a given grammar.

4.1 Structure of a YACC source program

A YACC source program is structurally similar to a LEX one.

```
declarations
%%
rules
%%
routines
```

- The declaration section may be empty. Moreover, if the routines section is omitted, the second %% mark may be omitted also.
- Blanks, tabs, and newlines are ignored except that they may not appear in names.

THE DECLARATIONS SECTION may contain the following items.

- Declarations of tokens. Yacc requires token names to be declared as such using the keyword `%token`.
- Declaration of the start symbol using the keyword `%start`
- C declarations: included files, global variables, types.
- C code between `%{` and `%}`.

RULES SECTION.

A rule has the form:

```
nonterminal : sentential form
            | sentential form
            .................
            | sentential form
            ;
```

Actions may be associated with rules and are executed when the associated sentential form is matched.

LEX-YACC INTERACTION

`yyparse()` calls `yylex()` when it needs a new token.

LEX	YACC
`return(TOKEN)`	%token TOKEN
	TOKEN is used in production

The external variable `yylval`

- is used in a LEX source program to return values of lexemes,
- `yylval` is assumed to be integer if you take no other action.
- Changes related to `yylval` must be made
  - in the definitions section of YACC specification
    * by adding new types in the following way

      ```
      %union {
          (type fieldname)
          (type fieldname)
          ...............
      }
      ```

    * and defining which token and non-terminals will use these types

      ```
      %token <fieldname> token
      %type <fieldname> non-terminal
      ```

  - in LEX specification by using the fieldnames in the assignment as follows

    ```
    yylval.fieldname = ...........
    ```

If you need a record type, then add it in the union. Example:

```
%union {
   struct s {
      double fvalue;
      int ivalue;
   } t;
}
```

- in the LEX specification use the record name and record field in assignments:

```
yylval.t.ivalue = ......
```

- in the YACC rules specification use the record field only in the assignment:

  ```
  $1.ivalue = ......
  ```

  assuming that `$1` has the appropriate type, whatever it denotes.

4.2 A first simple example

4.2.1 The LEX file

```
%{
#include "y.tab.h"
extern int yylval;
#include <stdlib.h>
%}

PLUS            [\+]
MINUS           [\-]
TIMES           [\*]
DIVIDE          [/]
DIGIT           [0-9]
NUMBER          [0-9]+
WS              [ \t]*
LP              "("
RP              ")"
RET             [\n]

%%

{WS}            {
                /* eat up white space */
                }
{NUMBER}        {
                yylval = atoi( yytext );
                return TOK_NUMBER;
```

```
                }
{PLUS}          {
                return TOK_PLUS;
                }
{TIMES}         {
                return TOK_TIMES;
                }
{MINUS}         {
                return TOK_MINUS;
                }
{DIVIDE}        {
                return TOK_DIVIDE;
                }
{LP}            {
                return TOK_LP;
                }
{RP}            {
                return TOK_RP;
                }
.               {
                }
{RET}           {
                return yytext[0];
                }
```

4.2.2 The YACC corresponding file

```
%{
#include <ctype.h>
#include <stdio.h>

void yyerror(const char *str)
{
  fprintf(stderr,"error: %s\n",str);
}

int yywrap()
{
```

```
  return 1;
}

int main()
{
  printf("> ");
  yyparse();
  return 0;
}


%}

%token TOK_NUMBER TOK_PLUS TOK_TIMES TOK_MINUS TOK_DIVIDE TOK_LP TOK_RP

%%
line        : line expr '\n' {printf("%d\n", $2); printf("> "); }
    | /* empty word */
;

expr        : expr TOK_PLUS term          {$$ = $1 + $3; }
            | term                  {$$ = $1; }


term        : factor TOK_TIMES term        {$$ = $1 * $3; }
            | factor              {$$ = $1; }


factor      : TOK_LP expr TOK_RP        {$$ = $2; }
            | TOK_NUMBER           {$$ = $1; }
             ;
%%
```

4.2.3 Constructing the parser without Makefile

```
[moreno@iguanodon yacc]$ ls
bin  Makefile  ps  README  src  test
```

```
[moreno@iguanodon yacc]$ yacc -d src/calculator.y
[moreno@iguanodon yacc]$ ls
bin  Makefile  ps  README  src  test  y.tab.c  y.tab.h
[moreno@iguanodon yacc]$ flex src/calculator.l
[moreno@iguanodon yacc]$ ls
bin  lex.yy.c  Makefile  ps  README  src  test  y.tab.c  y.tab.h
[moreno@iguanodon yacc]$ gcc lex.yy.c y.tab.c -Wall -I./src -ll -o bin/calculator
lex.yy.c:1030: warning: ‘yyunput’ defined but not used
src/calculator.y: In function ‘main’:
src/calculator.y:18: warning: implicit declaration of function ‘yyparse’
y.tab.c: In function ‘yyparse’:
y.tab.c:275: warning: implicit declaration of function ‘yylex’
[moreno@iguanodon yacc]$ ls bin/
calculator
[moreno@iguanodon yacc]$ ./bin/calculator
> 2 + 7
9
> (6 - 5) * (11 + 4)
error: syntax error
```

4.3 A detailed example

4.3.1 The file `type.h`

```
// tNodeType: enumarate type describing the type of the node
typedef enum {t_number, t_id, t_plus, t_times} tNodeType;

// definition of the Node
struct _tNode {
  tNodeType type;
  int number;
  char id;
  struct _tNode *left, *right;
};

// struct _Node and tNode are identical
typedef struct _tNode tNode;
```

4.3.2 The file `expressionTree.l`

```
%{
#include "type.h"
#include "y.tab.h"

//extern int yylval;
#include <stdlib.h>
%}

PLUS            [\+]
TIMES           [\*]
DIGIT           [0-9]
NUMBER          [0-9]+
ID              [a-zA-Z]
WS              [ \t]*
LP              "("
RP              ")"
RET             [\n]

%%

{WS}            {
                /* eat up white space */
                }
{NUMBER}        {
                yylval.intvalue = atoi( yytext );
                return TOK_NUMBER;
                }
{ID}            {
                yylval.charvalue = yytext[0];
                return TOK_ID;
                }
{PLUS}          {
                return TOK_PLUS;
                }
```

```
{TIMES}         {
                return TOK_TIMES;
                }
{LP}            {
                return TOK_LP;
                }
{RP}            {
                return TOK_RP;
                }
.               {
                }
{RET}           {
                return yytext[0];
                }
```

4.3.3 The file expressionTree.y

```
%{
#include <ctype.h>
#include <stdio.h>
#include <type.h>

tNode *buildNumber(int number) {
  tNode *ptr;
  ptr = (tNode*)malloc(sizeof(tNode));
  (*ptr).type = t_number;
  (*ptr).number = number;
  return ( ptr );
}


tNode *buildId(char id) {
  tNode *ptr;
  ptr = (tNode*)malloc(sizeof(tNode));
  ptr->type = t_id;
```

```
  ptr->id = id;
  return ( ptr );
}

tNode *buildOperator(tNodeType t, tNode *l, tNode *r) {
  tNode *ptr;
  ptr = (tNode*)malloc(sizeof(tNode));
  ptr->type = t;
  ptr->left = l;
  ptr->right = r;
  return ( ptr );
}

void printTree(tNode *node) {
  switch (node->type) {
    case t_number  : { printf("%d", node->number); break; }

    case t_id :      { printf("%c", node->id); break; }

    case t_times :   {
       printf("(* ");
       printTree(node->left);
       printf(" ");
       printTree(node->right);
       printf(")");
       break;
    }

    case t_plus : {
       printf("(+ ");
       printTree(node->left);
       printf(" ");
       printTree(node->right);
       printf(")");
       break;
    }
  }
```

```
}

void freeTree(tNode *node) {
  if ((node->type == t_number)||(node->type == t_id)) {
    free(node);
  } else {
    freeTree(node->left);
    freeTree(node->right);
    free(node);
  }
}

void yyerror(const char *str)
{
  fprintf(stderr,"error: %s\n",str);
}

int yywrap()
{
  return 1;
}

int main()
{
  yyparse();
  return 0;
}


%}


%union {
  int intvalue;
  char charvalue;
  tNode *nodevalue;
};
```

```
%token TOK_NUMBER TOK_ID TOK_PLUS TOK_TIMES  TOK_LP TOK_RP

%type <intvalue> TOK_NUMBER
%type <charvalue> TOK_ID
%type <nodevalue> expr term factor

%%
line        : line expr '\n' {
         printTree( $2 );
                 printf("\n");
                 freeTree( $2 );
            }
    | /* empty word */
            ;

expr        : expr TOK_PLUS term
              { $$ = buildOperator(t_plus, $1, $3 ); }
            | term
              {$$ = $1; } ;


term        : factor TOK_TIMES term
              {$$ = buildOperator(t_times, $1, $3); }
            | factor
      {$$ = $1; } ;


factor      : TOK_LP expr TOK_RP   {$$ = $2; }
    | TOK_ID               {$$ = buildId( $1 ); }
            | TOK_NUMBER           {$$ = buildNumber( $1 ); }
            ;
%%
```

4.3.4 The Makefile

```
# Makefile $CS447ROOT/yacc
# Marc Moreno Maza
```

```
# Dcembre 2002

SHELL = bash

# Files location
TOPDIR = $(PWD)
PSDIR = $(TOPDIR)/ps
SRCDIR = $(TOPDIR)/src
BINDIR = $(TOPDIR)/bin
TESTDIR = $(TOPDIR)/test

# Compilers, commands  and options
YACC = yacc
YACCO = -d
LEX = flex
LEXO =
CC = gcc
CCO = -Wall -I./src -ll -o
PP = a2ps
PPO = -q -o
MV = mv
MVO = -f
RM = rm
RMO = -f

# All files that require building
SRCS = calculator

all:  clean bin ps

bin: $(BINTARS)

test: $(TESTTARGS) clean

ps: $(PSTARGS)

clean:
@(cd $(BINDIR); $(RM) $(RMO) *)
```

```
@(cd $(SRCDIR); $(RM) $(RMO) *~)
@(cd $(TESTDIR); $(RM) $(RMO) *~)
@(cd $(TOPDIR); $(RM) $(RMO) *.c)
@(cd $(TOPDIR); $(RM) $(RMO) *.h)
@(cd $(PSDIR); $(RM) $(RMO) *)
@(cd $(TOPDIR); $(RM) $(RMO) *~)

distrib:

%.ps:
@($(PP) $(PPO) $(PSDIR)/`basename $*`.ps $(SRCDIR)/`basename $*`.l)

pdf:

html:

%.out:
@($(YACC) $(YACCO) $(SRCDIR)/$*.y)
@($(LEX) $(LEXO) $(SRCDIR)/$*.l)
@($(CC) $(TOPDIR)/lex.yy.c  $(TOPDIR)/y.tab.c  $(CCO) $(BINDIR)/$*)

.DEFAULT :
@ echo "------------------------------------------------------------"
@ echo "Building $< in $(BINDIR)"
@($(YACC) $(YACCO) $(SRCDIR)/$<.y)
@($(LEX) $(LEXO) $(SRCDIR)/$<.l)
@($(CC) $(TOPDIR)/lex.yy.c  $(TOPDIR)/y.tab.c  $(CCO) $(BINDIR)/$<)
@(cd $(TOPDIR); $(RM) $(RMO) lex.yy.c y.tab.c y.tab.h)
@ echo "------------------------------------------------------------"
```

4.3.5 A session

```
[moreno@iguanodon yacc]$ ls
bin  Makefile  ps  src  test
[moreno@iguanodon yacc]$ make expressionTree
------------------------------------------------------------
Building expressionTree in /home/moreno/src/Courses/CS447/yacc/bin
```

```
lex.yy.c:1026: warning: ‘yyunput’ defined but not used
/home/moreno/src/Courses/CS447/yacc/src/expressionTree.y: In function ‘main’:
/home/moreno/src/Courses/CS447/yacc/src/expressionTree.y:81: warning: implicit declaration of function ‘yyparse’
y.tab.c: In function ‘yyparse’:
y.tab.c:341: warning: implicit declaration of function ‘yylex’
------------------------------------------------------------
[moreno@iguanodon yacc]$ ./bin/expressionTree
1 + 2
(+ 1 2)
(1 + 2) * (x + y)
(* (+ 1 2) (+ x y))
1 + 2 + 3 + 4
(+ (+ (+ 1 2) 3) 4)
a*b*c + b*c*d + c*d*e
(+ (+ (* a (* b c)) (* b (* c d))) (* c (* d e)))
```

5 Parsing (II)

5.1 Bottom-up parsing

<u>KEY IDEAS.</u>

- The weakness of top-down $LL(k)$ parsing techniques is that they must predict which production to use, having seen only the first k tokens in the right side.
- The more powerful techniques of bottom-up $LR(k)$ parsing is able to postpone the decision until it has seen
  - input tokens corresponding to the *entire* right side of the production
  - and k lookahead tokens beyond

Example 21 Consider the grammar (with terminals a, b, c, d, e and nonterminals S, A, B)

$$\begin{array}{lcl} S & \longmapsto & aABe \\ A & \longmapsto & Abc \ \mid \ b \\ B & \longmapsto & d \end{array}$$

Consider the input word $w = abbcde$. We reduce w to S by repeating the following algorithm.

1. Choose a string β such that
   - β is the leftmost factor of w such that
   - there exists a production $A \longmapsto \beta$.
2. Replace β by A in w.

Warning! This algorithm is a first approach. (True algorithms for botton-up parsing do not follow exactly this scheme.) Well, our *first-approach* algorithms leads to the following sequence of transformations in the first column of the table below. The second column gives the production that could be used in order to rewrite the current word *backwards*. The last column gives the chosen production according to the above algorithm.

Current string	Qualifying productions	Chosen production
$abbcde$	$A \longmapsto b$ $B \longmapsto d$	 $A \longmapsto b$
$aAbcde$	$A \longmapsto Abc$ $A \longmapsto b$ $B \longmapsto d$	 $A \longmapsto Abc$
$aAde$	$B \longmapsto d$	$B \longmapsto d$
$aABe$	$S \longmapsto aABe$	$S \longmapsto aABe$
S		

Observe that we have

$$S \stackrel{rm}{\Longrightarrow} aABe \stackrel{rm}{\Longrightarrow} aAde \stackrel{rm}{\Longrightarrow} aAbcde \stackrel{rm}{\Longrightarrow} abbcde \tag{44}$$

where $\alpha \stackrel{rm}{\Longrightarrow} \beta$ means that β derives from α by a rightmost derivation.

Bottom-up parsing constructs a parse tree for an input string beginning at the leaves and working up towards the root.

- To do so, bottom-up parsing tries to find a rightmost derivation of a given string backwards.
- Bottom-up parsing is also called *shift-and-reduce* parsing where
  - *shift* means read the next token,
  - *reduce* means that a substring β matching the right side of a production $A \longmapsto \beta$ is replaced by A.

Handle. Let $G = (V_T, V_N, S, P)$ be a context-free grammar. Let n be a positive integer, let w, β be two strings of symbols and let A be a nonterminal such that $A \longmapsto \beta$ is a production of G. We say that $(A \longmapsto \beta, n)$ is a *handle* for w if there exists a string of symbols γ such that

$$S \stackrel{rm}{\Longrightarrow}^{*} \alpha A \gamma \stackrel{rm}{\Longrightarrow} \alpha\beta\gamma = w \tag{45}$$

and such that $n = \mid \alpha \mid +1$.

More informally, a handle of w is a substring β that matches a right side of a production $A \longmapsto \beta$ such that the reduction of the handle corresponds to a step of the reverse of a rightmost derivation.

Theorem 4 *Let* $G = (V_T, V_N, S, P)$ *be a context-free grammar. If* G *is unambiguous then for each string of symbols* w *such that* $S \overset{*}{\Longrightarrow} w$ *there exists exactly one handle.*

Example 22 Consider the grammar

$$S \quad \longmapsto \quad aSb \mid \varepsilon$$

- The handle of $aabb$ is $(S \longmapsto \varepsilon, 3)$
- The handle of $aaSbb$ is $(S \longmapsto aSb, 2)$
- The handle of aSb is $(S \longmapsto aSb, 1)$

Example 23 Consider the grammar

$$\begin{array}{lcl} S & \longmapsto & AC \mid BC \\ A & \longmapsto & a \\ B & \longmapsto & a \\ C & \longmapsto & c \end{array}$$

The string ac has handles $(A \longmapsto a, 1)$ and $(B \longmapsto a, 1)$. So the grammar is ambiguous.

Example 24 Consider the grammar

$$\begin{array}{lcl} S & \longmapsto & AB \mid A \\ A & \longmapsto & a \\ B & \longmapsto & b \end{array}$$

N.B. In the sentential form Ab the handle is $(B \longmapsto b, 2)$ and not $(S \longmapsto A, 1)$.

<u>Handle pruning</u> is the general approach used in shift-and-reduce parsing. The implementation of handle pruning involves the following data-structures:

- ***a stack*** to hold the grammar symbols,
- ***an input buffer*** containing the remaining input,
- ***a table*** to decide handles.

Initially the situation is

Stack	Input
\$	$w\$$

where w is the sentence to parse. The possible actions of a shift-and-reduce parser are:

- ***shift*** where the next symbol is shifted onto the top of the stack

- ***reduce*** where the parser

  - knows that the right end of the handle is at the top of the stack,
  - locates the left end of the handle within the stack and
  - decide with what nonterminal to replace the handle.

- ***accept*** where the parser announces successful completion of parsing.

- ***error*** where the parser discovers that a syntax error has occured and calls an error recovery routine.

The final configuration must be

Stack	Input
$\$S$	\$

Example 25 Consider the grammar G

$$
\begin{array}{lcl}
E & \longmapsto & E + E \\
E & \longmapsto & E * E \\
E & \longmapsto & (E) \\
E & \longmapsto & \mathbf{id}
\end{array}
$$

A rigthmost derivation of the sentence $w = \mathbf{id} + \mathbf{id} * \mathbf{id}$ is

$$
\begin{array}{lcl}
S & \stackrel{rm}{\Longrightarrow} & E + E \\
 & \stackrel{rm}{\Longrightarrow} & E + E * E \\
 & \stackrel{rm}{\Longrightarrow} & E + E * \mathbf{id} \\
 & \stackrel{rm}{\Longrightarrow} & E + \mathbf{id} * \mathbf{id} \\
 & \stackrel{rm}{\Longrightarrow} & \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{array}
\tag{46}
$$

Let us sketch the actions a shift-and-reduce parser might take in parsing w to obtain the above rigthmost derivation. (Another shift-and-reduce parser might obtain another rigthmost derivation of w w.r.t. G.)

Stack	Input	Action to take
$\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	shift
$\$\mathbf{id}$	$+\mathbf{id} * \mathbf{id}\$$	reduce with $E \longrightarrow \mathbf{id}$
$\$E$	$+\mathbf{id} * \mathbf{id}\$$	shift
$\$E+$	$\mathbf{id} * \mathbf{id}\$$	shift
$\$E + \mathbf{id}$	$*\mathbf{id}\$$	reduce with $E \longrightarrow \mathbf{id}$
$\$E + E$	$*\mathbf{id}\$$	shift
$\$E + E*$	$\mathbf{id}\$$	shift
$\$E + E * \mathbf{id}$	$\$$	reduce with $E \longrightarrow \mathbf{id}$
$\$E + E * E$	$\$$	reduce with $E \longrightarrow E * E$
$\$E + E$	$\$$	reduce with $E \longrightarrow E + E$
$\$E$	$\$$	accept

Of course the decisions (especially for the 4th shift) made in the above table are not explained at this point.

Handle pruning can be implemented using a stack because of the following fact.

Proposition 1 *Let $G = (V_T, V_N, S, P)$ be an unambiguous context-free grammar. In a shift-and-reduce parser for G implemented with a stack, the right end of the handle will always appear on top the stack.*

PROOF $\triangleright$ This fact becomes clear when considering the possible forms of two successive steps in any rightmost derivation. These two steps can be of the form (type 1)

$$S \overset{rm}{\Longrightarrow}^{*} \alpha\, A\, z \overset{rm}{\Longrightarrow}^{*} \alpha\, \beta\, B\, y\, z \overset{rm}{\Longrightarrow}^{*} \alpha\, \beta\, \gamma\, y\, z \tag{47}$$

or the form (type 2)

$$S \overset{rm}{\Longrightarrow}^{*} \alpha\, B\, x\, A\, z \overset{rm}{\Longrightarrow}^{*} \alpha\, B\, x\, y\, z \overset{rm}{\Longrightarrow}^{*} \alpha\, \gamma\, x\, y\, z \tag{48}$$

- In type 1, first the rightmost nonterminal A is replaced by the right side of the production $A \longrightarrow \beta\, B\, y$ and then the rightmost nonterminal B is replaced by the right side of the production $B \longrightarrow \gamma$.
- In type 2, the rightmost nonterminal A is replaced first by use of the rule $A \longrightarrow y$ (where y is a string of one or more terminals) and then B is replaced by using $B \longrightarrow \gamma$.

We proceed by induction. Let us assume that the handle $B \longrightarrow \gamma$ appears on top of the stack. Then let us show that the handle $A \longrightarrow \beta\, B\, y$ in type 1 (and the handle $A \longrightarrow y$ in type 2) appears on top also. Let us consider type 1 in reverse, where a shift-and-reduce parser has just reached the following configuration.

Stack	Input
$\alpha\,\beta\,\gamma$	$y\,z$

The parser now reduces the handle.

Stack	Input
$\alpha\,\beta\,B$	$y\,z$

Since B is the rightmost nonterminal, the right end of the handle of $w = \alpha\,\beta\,Byz$ cannot be to the left of B. Therefore the parser can shift y.

Stack	Input
$\alpha\,\beta\,B\,y$	z

For case 2 in reverse, assume the following configuration has been reached

Stack	Input
$\alpha\gamma$	$x\,y\,z$

Reduction leads to

Stack	Input
$\alpha\,B$	$x\,y\,z$

Again B is the rightmost nonterminal so it needs to be involved.
$\lhd$

5.2 *LR* Parsers

<u>Key ideas.</u>

- The LR parsing method is the most general non-backtrackingshift-reduce parsing method known.
- LR parsers can parse a strictly larger class of grammars than (top-down) predictive parsers.

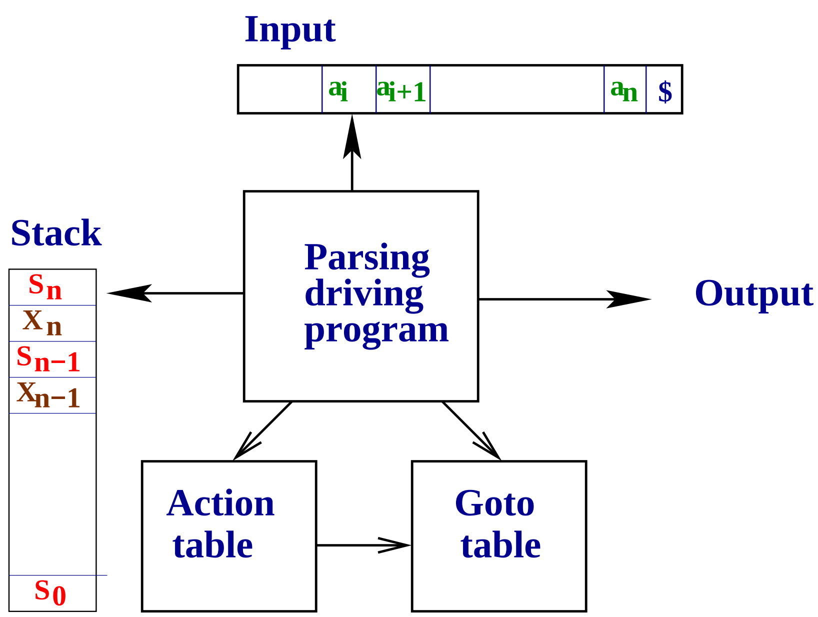

Figure 9: Model of an LR parser.

- LR parsers can usually recognize all programming language construct that can be specified by context-free grammars.
- LR parsers detect errors fast.
- Drawback: it is too much work to construct an LR parser by hand.
- Fortunately, we can use an LR parser generator such as YACC.

<u>THE *LR* PARSING PRINCIPLE</u>

An LR-Parser uses

- states to *memorize* information during the parsing process,
- an *action* table to make decision (such as *shift* or *reduce*) and to compute states
- a *goto* table to compute states

These states are embedded with grammar symbols in a stack. More, precisely, after eahc iteration, the stack stores a word of the form $s_0 X_1 s_1 X_2 \cdots s_{m-1} X_m s_m$ where s_0 is the end-of-stack symbol and s_m is the state on top of the stack.

For a state s and a terminal a, the entry action$[s, a]$ has one of the four following forms

- shift s' where s' is a state,
- reduce $A \longmapsto \beta$,
- accept,
- error.

Algorithm 10

Input: A LR-Parser for an unambiguous context-free grammar G over an alphabet Σ and a word $w \in \Sigma^*$.

Output: An error if $w \notin L(G)$ or a rightmost derivation for w otherwise.

Set the cursor to the rightmost symbol of $w\$$
push the initial state s_0 on top of the empty stack
repeat
  let s be the state on top of the stack
  let a be the current pointed symbol in $w\$$
  if action$[s, a]$ = shift s' **then**
    push a on top of the stack
    push s' on top of the stack
    advance the cursor to the next symbol on the right in $w\$$
  else if action$[s, a]$ = reduce $A \longmapsto \beta$ **then**
    pop $2 \mid \beta \mid$ symbols of the stack
    let s' be the state on top of the stack
    push A on top of the stack
    push goto$[s', A]$ on top of the stack
    output $A \longmapsto \beta$
  else if action$[s, a]$ = accept **then**
    return
  else
    error