Introduction to Computer Science

Géza Horváth

Fifth Lecture

Notation techniques for programming languages

Notation techniques were introduced as simple methods to describe different parts of programming languages. These parts contain terminal and nonterminal symbols. Terminals are given, and nonterminals can be built up from terminals and already defined nonterminals by using simple operations. These operations are the following:

1. Concatenation, when symbols are written after each other.
2. Alternation is a selection from different possibilities.
3. Option is a special selection between a symbol and the empty word.
4. Repetition, when a symbol can be repeated any (≥ 0) number of times.

In this section we introduce two well known techniques, the Backus-Naur form (BNF) and the syntax diagram, but many others have been introduced for a variety of reasons.

Notation techniques – Backus-Naur form

BNF was designed by Peter Naur in 1963 as a simplified version of the notation technique of John Backus. It was used first to describe the programming language ALGOL60. The following table shows the marking of the operations used by BNF.

Operations of the BNF metasyntax.

Definition	**Concatenation**	**Alternation**	**Option**	**Repetition**
::=		|	[ ]	{ }

As you can see, concatenation does not have any special mark, we just write the symbols after each other. We use a terminal symbol as it is, for example, the mark of one as a number is 1. For nonterminals we use their names between angle brackets. We have a special mark to define nonterminal symbols, followed by the description of the nonterminal.

Notation techniques – BNF example

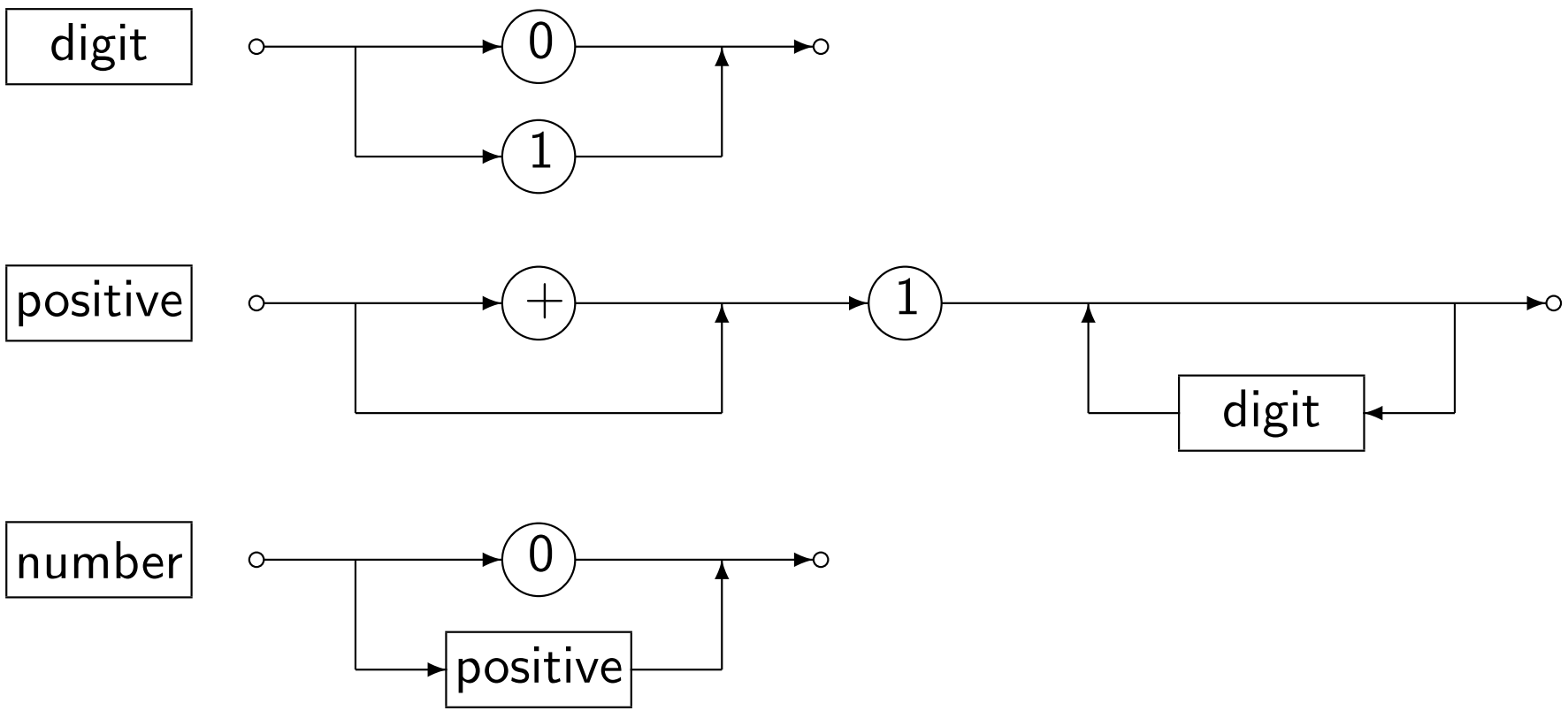
Example

In this example, we describe a non-negative binary number using BNF metasyntax.

$<\text{digit}> ::= 0 \mid 1$
$<\text{positive}> ::= [\ +\ ]\ 1\ \{\ <\text{digit}>\ \}$
$<\text{number}> ::= 0 \mid <\text{positive}>$

Notation techniques – syntax diagram

A syntax diagram is a graphical notation technique. It uses simple graphs, each of them has an entry and an end point. The concatenation, alternation, option and repetition operations are implemented in the structure of the graph.



Chomsky normal form

This normal form was introduced by Noam Chomsky for λ-free context-free languages. Using the Chomsky normal form instead of the universal context-free grammar makes it more simple to store the grammar in the memory of the computer, to calculate using the grammar and to prove theorems about context-free languages.

Chomsky normal form – definitions

Definition

A generative grammar is said to be λ-free grammar if none of its production rules contains the empty word λ on the right hand side.

We have to note that each $\lambda \notin L$ context-free language can be generated by some λ-free context-free grammar.

Definition

The grammar $G = (N, T, S, P)$ is in Chomsky normal form, if all of its production rules has the form:

1. $A \to BC$ *or*
2. $A \to a$,

where $A, B, C \in N$ and $a \in T$.

Chomsky normal form – theorem + proof

First, we have to prove that each λ-free context-free language can be generated by a Chomsky normal form grammar.

Theorem

For each λ-free context-free grammar $G = (N, T, S, P)$ one can give Chomsky normal form grammar $G' = (N', T, S, P')$ such that $L(G) = L(G')$.

Proof. We are going to give a constructive proof of this theorem. We are going to show the necessary steps to construct a Chomsky normal form grammar $G' = (N', T, S, P')$ which is equivalent to the original λ-free context-free grammar $G = (N, T, S, P)$. It can easily be seen that we get equivalent grammars after each step.

1. First of all we create the grammar $G_1 = (N_1, T, S, P_1)$ such that all of its production rules are of the form:
   - $A \to p$, or
   - $A \to a$,

   where $A \in N$, $a \in T$ and $p \in N^+$.

Chomsky normal form – theorem + proof

1. In this step we eliminate each terminal symbol from the production rules whose right-hand side contains more than one letter. To do this we introduce new nonterminal symbols for each such terminal symbol. Let the set of new nonterminals be $N_{new} = \{X \mid a \in T, A \to paq \in P,\ |pq| \neq 0\}$ and let $N_1 = N \cup N_{new}$. Then let the set P_1 be the union of 3 different sets:
   - $\{A \to p \mid A \to p \in P,\ p \in \{T \cup N^+\}\} \subset P_1$ (we keep the rules for which the right hand side contains just one terminal, or contains only nonterminal symbols),
   - $\{X \to a \mid X \in N_{new}\} \subset P_1$ (we add new rules, the right hand side contains the terminal symbol, and the left hand side contains the nonterminal introduced for it),
   - $\{A \to p_0X_1p_1X_2 \dots X_np_n \mid A \to p_0a_1p_1a_2 \dots a_np_n \in P, a_1, a_2, \dots, a_n \in T,\ X_1, X_2, \dots, X_n \in N_{new}, p_0, p_1, \dots, p_n \in N^*, |p_0a_1p_1a_2 \dots a_np_n| \geq 2\} \in P_1$ (here we change each appearance of the terminals to the nonterminal introduced for it in each rule, with right hand side of at least two letters).

Chomsky normal form – theorem + proof

1. Now we have the sets N_1 and P_1 and the grammar $G_1 = (N_1, T, S, P_1)$ satisfies the above conditions. It can be easily shown that $L(G_1) = L(G)$.
2. The next step is to eliminate the long rules. We create the grammar $G_2 = (N_2, T, S, P_2)$ such that all of its production rules are of the form:
   - $A \to B$, or
   - $A \to BC$, or
   - $A \to a$,

   where $A, B, C \in N$ and $a \in T$. To reach our goal, we have to replace the long rules in P_1 with short ones in P_2. For each rule $A \to B_1B_2...B_k \in P_1,\ k \geq 3$ we introduce new nonterminals $Z_1, Z_2, ..., Z_{k-2}$. The set N_2 contains these new nonterminals and the nonterminals contained by the set N_1. In the set P_2 we keep those productions rules from the set P_1 whose right hand side contains at most two letters and instead of each $A \to B_1B_2...B_k \in P_1,\ k \geq 3$ rule we introduce the rules:

Chomsky normal form – theorem + proof

2. $A \to B_1 Z_1$,
   $Z_1 \to B_2 Z_2$,
   $\vdots$
   $Z_{k-3} \to B_{k-2} Z_{k-2}$,
   $Z_{k-2} \to B_{k-1} B_k$.
   The grammar $G_2 = (N_2, T, S, P_2)$ has no long rules and $L(G_2) = L(G_1)$.
3. The third step is to eliminate the rules of the form $A \to B$, where $A, B \in N$.

   First, for each nonterminal letter A let us collect all nonterminal letters $B_1, B_2, ..., B_k$ such that A can be derived from B_i, $1 \leq i \leq k$. Let $U(A) = \{B_1, B_2, ..., B_k\}$ for each nonterminal A. The following formulas make this pocedure simple:

   - $U_1(A) = \{A\}$
   - $U_{i+1}(A) = U_i(A) \cup \{B \mid B \to C \in P_2, C \in U_i(A)\}$
   - if $U_k(A) = U_{k+1}(A)$ then $U(A) = U_k(A)$

Chomsky normal form – theorem + proof

3. When we have set U for each nonterminal letter, we can define set P' with the following formula: $P' = \{B \to p \mid A \to p \in P_2, B \in U(A), p \notin N_2\}$. Then $N' = N_2$, $G' = (N', T, S, P')$ and $L(G') = L(G_2) = L(G_1) = L(G)$.

QED.

See the example on the whiteboard.

Parsing

In formal language theory, parsing – or the so called syntactic analysis – is a process when the parser determines if a given string can be generated by a given grammar. This is very important for compilers and interpreters. For example, it is not too difficult to create a context-free grammar G_P generating all syntactically correct Pascal programs. Then, we can use a parser to decide – about a Pascal program written by a programmer – if the program is in the generated language $L(G_P)$. When the program is in the generated language, it is syntactically correct.

CYK algorithm – description

We have a given Chomsky normal form grammar $G = (N, T, S, P)$ and a word $p = a_1 a_2 \dots a_n$. The Cocke–Younger–Kasami algorithm is a well known method to decide wether $p \in L(G)$ or $p \notin L(G)$. To make our decision, we have to fill out an $n \times n$ size triangular matrix M in the following way: Over the cells of the first line, we write the letters $a_1, a_2, \dots, a_n$, starting from the first letter, one after the other. Then, the cell $M(i, j)$ contains each nonterminal symbol A, if and only if the subword $a_j a_{j+1} \dots a_{j+i-1}$ can be derived from A. (Formally: $A \in M(i, j)$ if and only if $A \Rightarrow^* a_j a_{j+1} \dots a_{j+i-1}$.) This means that the first cell of the first line contains the nonterminal A, if and only if $A \to a_1 \in P$. The cell $M(1, j)$ contains the nonterminal A, if and only if $A \to a_j \in P$. It is also quite easy to fill out the cells of the second line of the matrix. The nonterminal A is in the cell $M(2, j)$, if and only if there exists nonterminals $B, C \in N$ such that $B \in M(1, j)$, $C \in M(1, j + 1)$, and $A \to BC \in P$. From this point the algorithm becomes more complex.

CYK algorithm – description

From the third line, we use the following formula: $A \in M(i,j)$, if and only if there exists nonterminals $B, C \in N$ and integer k such that $B \in M(k,j)$, $C \in M(i-k, j+k)$ and $A \to BC \in P$. This algorithm is finished when the cell $M(n,1)$ is filled out. Remember, the nonterminal A is in the cell $M(i,j)$, if and only if the word $a_j a_{j+1} ... a_{j+i-1}$ can be derived from A. This means that the nonterminal S is in the cell $M(n,1)$, if and only if the word $a_1 a_2 ... a_n$ can be derived from S. So the grammar G generates the word p, if and only if the cell $M(n,1)$ contains the start symbol S. The triangular matrix M for the CYK algorithm:

a_1	a_2	$\dots$	$\dots$	a_{n-1}	a_n
(1,1)	(1,2)	$\dots$	$\dots$	(1,n-1)	(1,n)
(2,1)	(2,2)	$\dots$	$\dots$	(2,n-1)	
$\vdots$	$\vdots$				
$\vdots$	$\vdots$				
(n-1,1)	(n-1,2)				
(n,1)					

CYK algorithm – example

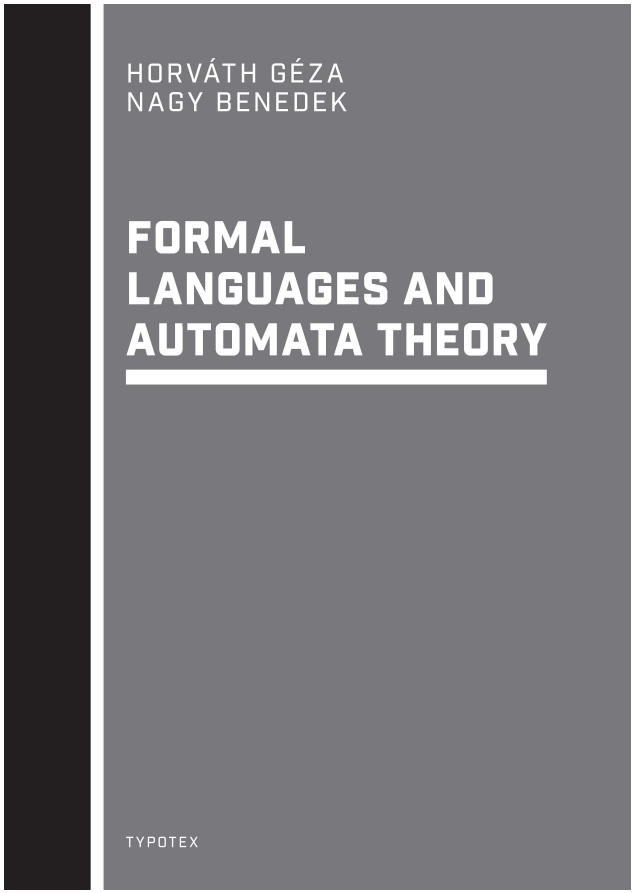
In this example, we use the CYK algorithm to show that the grammar G generates the word *abbaa*.

$G = (\{S, A, B\}, \{a, b\}, S, P)$

$P = \{$
$S \to SA,$
$S \to AB,$
$A \to BS,$
$B \to SA,$
$S \to a,$
$A \to a,$
$B \to b$
$\}$

a	b	b	a	a
S, A	B	B	S, A	S, A
S	-	A	S, B	
-	-	A		
S, B	-			
S, B, A				

References



This work was supported by the construction EFOP-3.4.3-16-2016-00021. The project was supported by the European Union, co-financed by the European Social Fund.