Zpracování textových souborů

Obsah:

- Užitečné funkce v ANSI C
- Konečné automaty
- Gramatiky

Užitečné funkce

Knihovna <stdlib.h>

- atof() converts a string to a double
- atoi() converts a string to an integer
- atol() converts a string to a long
- strtod() converts a string to a double
- strtol() converts a string to a long
- strtoul() converts a string to a long

Užitečné funkce ANSI C

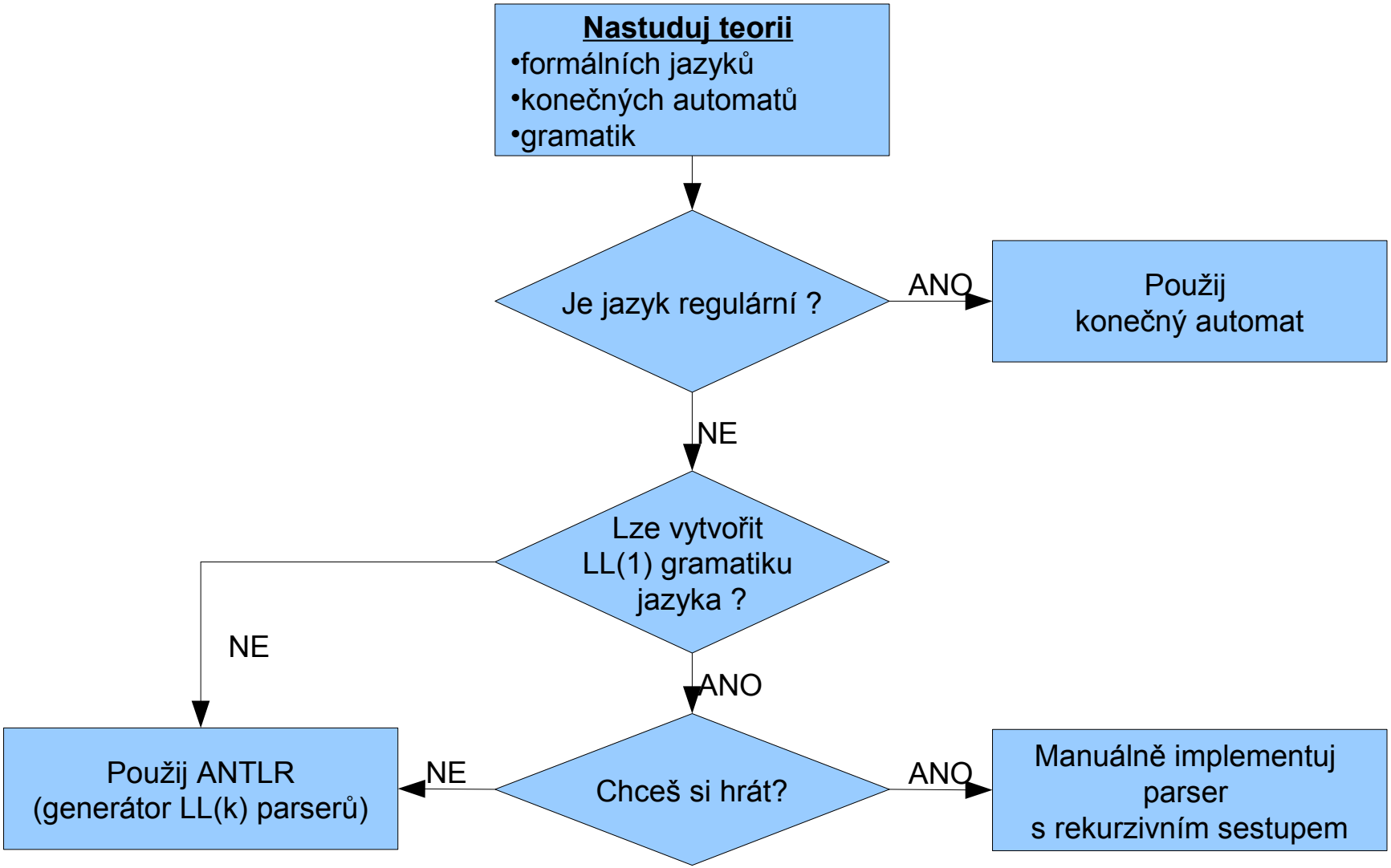
- Při zpracování textových souborů budete nejčastěji využívat funkce z knihoven <stdlib.h>, <string.h> a <ctype.h>

- Knihovna <ctype.h> obsahuje užitečné funkce pro práci se znaky:
  - isalnum() true if alphanumeric
  - isalpha() true if alphabetic
  - iscntrl() true if control character
  - isdigit() true if digit
  - isgraph() true if a graphical character
  - islower() true if lowercase
  - isprint() true if a printing character
  - ispunct() true if punctuation
  - isspace() true if space
  - isupper() true if uppercase character
  - isxdigit() true if a hexidecimal character
  - tolower() converts a character to lowercase
  - toupper() converts a character to uppercase

Jak na složitější textové struktury?

- Pro mnoho dnes používaných strukturovaných jazyků jsou k dispozici knihovny s hotovými parsery, např.: XML (knihovna expat), HTML (HTML-tidy), ...
- Pokud potřebujeme parsovat formát, pro který ještě parser nikdo nenapsal, pak máme k dispozici pouze uvdené základní funkce ANSI C. Co s tím?
- Musíme se naučit implementovat parsery „na zelené louce“ !

Jak implementovat parser ?

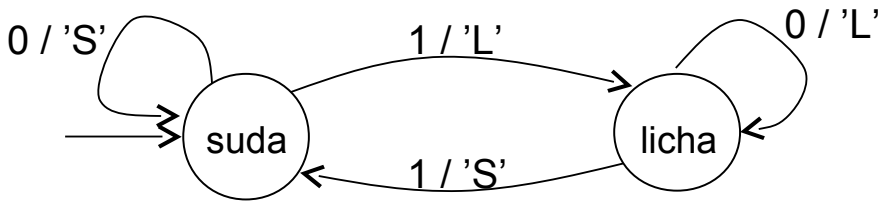


Formální a neformální metody pro zpracování jazyků

- Historie: praxe i teorie vznikala zároveň
  - **1936:** Turingův stroj
  - **Začátek 50. let:** první překladače (implementované jako „Babylonská věž brouka Pytlíka“)
  - **1955:** lingvista Noam **Chomsky** v rámci své PhD disertace uveřejňuje publikaci „*Logical Structure of Linguistic Theory*“, ve které poprvé zavádí formalismus „generující gramatika
  - **1957**: Chomsky publikuje monografii „*Syntactic Structures*“, ve které jsou shrnuty výsledky jeho PhD a post-PhD práce.
  - **1957**: FORTRAN team (IBM) pod vedením **Johna Backus**-e dokončil první kompletní překladač.
  - **1959/1960**: Backus a Naur spolu vytváří notaci **BNF**, která je ekvivalentní bezkontextové gramatice a dodnes se používá pro specifikaci syntaxe počítačových jazyků. Je vysoce pravděpodobné, že Backus v té době ještě práci Chomského neznal
  - **60. léta** – Mealy a Moore definují svoje konečné automaty

Konečný automat

- Konečný automat je jednoduchý, ale účinný nástroj, pomocí kterého můžeme analyzovat regulární jazyky (viz slajdy o regulárních gramatikách)
- Konečný automat je abstrakce s formálně definovanou sémantikou, díky níž je např. možno použít pro ověření funkce navrženého algoritmu některé metody automatické verifikace.
- „Co to teda vlastně je“?
  Na obrázku níže máte příklad jednoduchého Mealyho konečného automatu s výstupem:



- Kolečka „suda“ a „licha“ jsou stavy automatu. Šipka zleva do stavu „suda“ znamená, že se jedná o počáteční stav. Ostatní šipky znázorňují přechodovou funkci neboli přechody z jednoho stavu do druhého, výrazy nad šipkami pak určují podmínku, ze které se přechod stane, a výstupní symbol, který přitom vygenerován na výstup. Např. šipka ze stavu 1/ ‘L’ znamená, že při příjmu symbolu „1“ ve stavu „suda“ přejde automat do stavu „licha“ a na výstup vygeneruje znak ‘L’

Formální definice

Konečný automat je pětice

$KA = (Q, \Sigma, O, \delta, \lambda)$

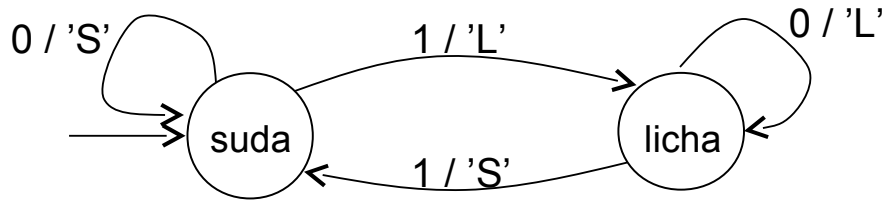
kde

- Q je konečná, neprázdná, množina stavů
- Σ je konečná neprázdná množina vstupních prvků - symbolů (vstupní abeceda)
- O je konečná neprázdná množina výstupních symbolů (výstupní abeceda)
- δ (přechodová funkce) je zobrazení $\delta: Q \times \Sigma \rightarrow Q$. Přechod je opět určen stavem ve kterém se automat nachází a symbolem, který je čten na vstupu

$\forall$ λ (výstupní funkce) je zobrazení $Q \times \Sigma \rightarrow O$ (Mealy) nebo $Q \rightarrow O$ (Moore)

Příklad

- Jednoduchý automat, který kontroluje paritu ve vstupním souboru binárních čísel – pokud je parita sudá, generuje na výstup znak ‘S’, pokud je lichá, znak ‘L’:



Implementace v C:

```
enum {suda, licha} stav;
while((c=getchar())!=EOF)
   swich (stav) {
        case suda: if (c=='1') {putchar('L'); stav=licha;}
                   else if (c=='0') putchar('S');
                   break;
        case licha: if (c=='1') {putchar('S'); stav=suda;}
                    else if (c=='0') putchar('L');
                    break;
   }
}
```

Další typy automatů

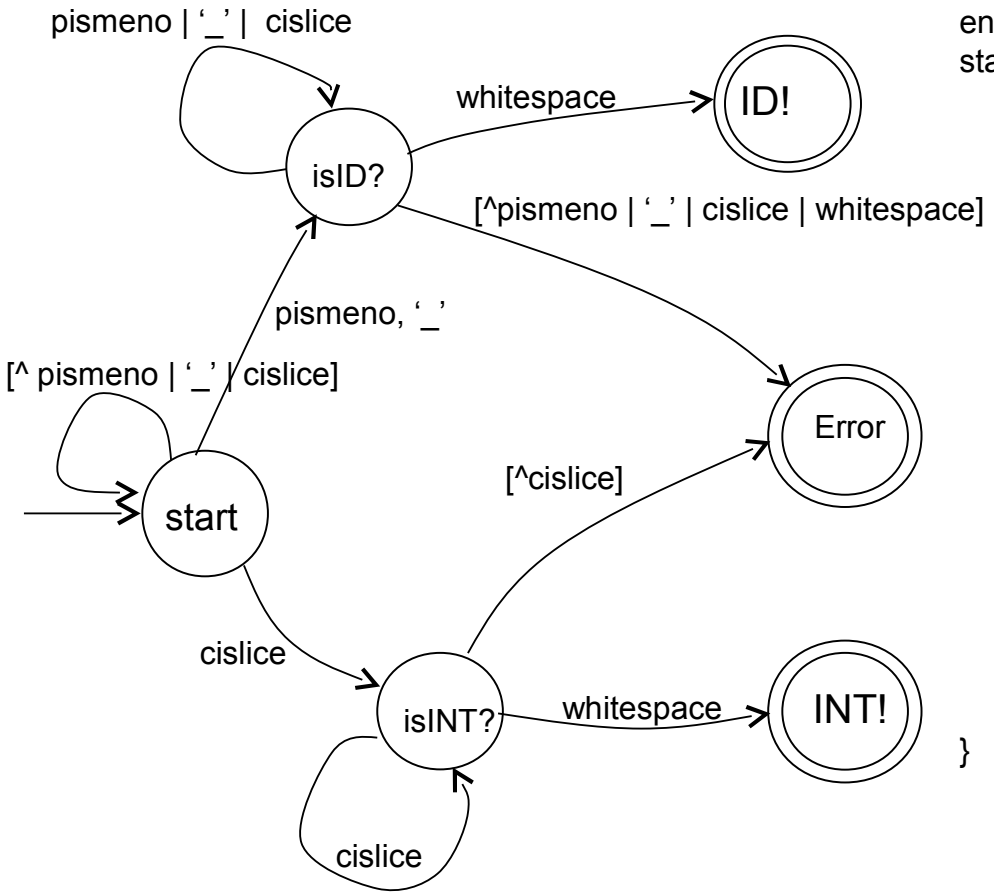
Automat typu *akceptor* – nemá výstupní abecedu O ani výstupní funkci λ. Definuje se jako pětice $A = (Q, \Sigma, \delta, q_0, F)$, kde:

- Q je konečná, neprázdná, množina stavů
- Σ je konečná neprázdná množina vstupních symbolů (vstupní abeceda)
- δ (přechodová funkce) je zobrazení $\delta: Q \times \Sigma \rightarrow Q$. Přechod je určen stavem ve kterém se automat nachází a symbolem, který přichází na vstup (nebo který je čten na vstupu)
- q_0 je počáteční (iniciální) stav ($q_0 \in Q$)
- F je množina koncových stavů ($F \subseteq Q$)

Automat typu akceptor

- Automat typu akceptor je v podstatě pouze zjednodušeným Mealyho (popř. Moorovým) automatem – negeneruje na výstupu žádné textové řetězce, jeho výstupem je pouze konečný stav, ve kterém skončí analýza vstupního řetězce
- akceptor se používá např.
  - ke kontrole správnosti syntaxe regulárních jazyků
  - V překladačích je akceptor využit jako lexikální analyzátor neboli „tokenizer“ - tj. procedura, která kouskuje vstupní soubor na jednotlivé tokeny (z hlediska gramatiky jde o terminální symboly např. „poznámka“, „číslo“, „identifikátor“ atd.)
- Příklad: rozlišení identifikátoru a celočíselné konstanty v textovém souboru (identifikátor musí začínat písmenem nebo podtržítkem a pokračuje libovolným řetězcem písmen a číslic)

Příklad – rozlišení identifikátorů a integer konstant



Implementace v C:

```
enum {start, isID, isINT, id, int, error, eof} stav;
stav lexAnalyzer(FILE * soubor) {
    while ((c=fgetc(soubor))!=EOF)
      switch (stav)  {
            case start:
                if (isalpha(c) || c=='_') stav=isID;
                else if (isdigit(c)) stav=isINT;
                break;
            case isID:
                if (isspace(c)) return id;
                else
                  if (!(isspace(c) || isalphanum(c) || c=='_'))
                    return error;
                break;
            case isINT:
                if (isspace(c)) return id;
                else (!isdigit(c)) return error;
      }
    if (c==EOF) return eof;
}
```

BNF - Backus-Naur Form

- Historie BNF
  - 1959 - John Backus (IBM) navrhl „Backus Normal Form“ jako formální notaci pro zápis syntaxe programovacího jazyka
  - 1960: Peter Naur, dánský pionýr informatiky, pomohl publikovat Backusovu notaci
  - Výsledek = Backus-Naurova Forma, sám Naur ale odmítá tento název a propaguje „Backus Normal Form“
  - BNF se dodnes používá k definici syntaxe počítačových jazyků: viz syntaxe command-line příkazů, YACC, Bison, ANTLR...

Backus-Naur Form

- BNF je sada derivační pravidel s následující syntaxí: <symbol> ::= __expression__
- **<symbol>** je **nonterminální symbol**,
- **__expression__** se skládá z jedné nebo více sekvencí symbolů. Sekvence v rámci jednoho pravidla oddělujeme znakem | (choice - volba)
- Význam pravidla: <symbol> lze nahradit jednou ze sekvencí
- Symboly, které se v BNF neobjeví na levé straně, jsou **terminální symboly**

BNF - příklad

- Syntaxe BNF pravidel v BNF :-):

<syntax> ::= <rule> | <rule> <syntax>
<rule> ::= <opt-whitespace> "<" <rule-name> ">" <opt-whitespace> "::=" <opt-whitespace> <expression> <line-end>
<opt-whitespace> ::= *ε* <opt-whitespace> | ""
<expression> ::= <list> | <list> "|" <expression>
<line-end> ::= <opt-whitespace> <EOL> | <line-end> <line-end>
<list> ::= <term> | <term> <opt-whitespace> <list>
<term> ::= <literal> | "<" <rule-name> ">"
<literal> ::= "" <text> "" | "'" <text> "'"

(Viz wikipedia)

EBNF

- EBNF – Extended BNF – vyvinul Niklaus Wirth, později bylo standardizováno jako EBNF standard ISO-14977. Zavádí následující speciální znaky:
  - definition =
  - concatenation ,
  - termination ;
  - separation |
  - option [...]
  - repetition { ... }
  - grouping (...)
  - double quotation marks " ... "
  - single quotation marks ' ... '
  - comment (* ... *)
  - special sequence ? ... ?
  - exception -

EBNF

- Terminály musí být v uvozovkách
- Ostré závorky <> u nonterminálů mohou být vypuštěny
- Každé pravidlo musí končit středníkem
- Příklad: jednoduchý Pascalovský jazyk, který obsahuje pouze příkazy přiřazení

```
(* a simple program in EBNF − Wikipedia *)
program = 'PROGRAM' , white space , identifier , white space ,
 'BEGIN' , white space ,
{ assignment , ";" , white space } ,
 'END.' ;
identifier = alphabetic character , { alphabetic character | digit } ;
number = [ "-" ] , digit , { digit } ;
string = "" , { all characters − "" } , "" ;
assignment = identifier , ":=" , ( number | identifier | string ) ;
alphabetic character = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N"| "O" |
"P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
white space = ? white space characters ? ;
all characters = ? all visible characters ? ;
```

Gramatiky

- Gramatika definuje syntaktickou strukturu jazyka
- V Chomského teorii gramatik je „Generující gramatika“ formalismem, který podle daných pravidel generuje nějaký jazyk.
- Generující gramatika je čtveřice

$$G=(N, T, P, S)$$

kde:

- ***N*** je množina nonterminálů (používají se k označení syntaktických celků
- ***T*** je množina terminálních symbolů, neboli abeceda, nad kterou je jazyk definován
- ***S*** je počáteční symbol gramatiky
- ***P*** je konečná množina přepisovacích pravidel

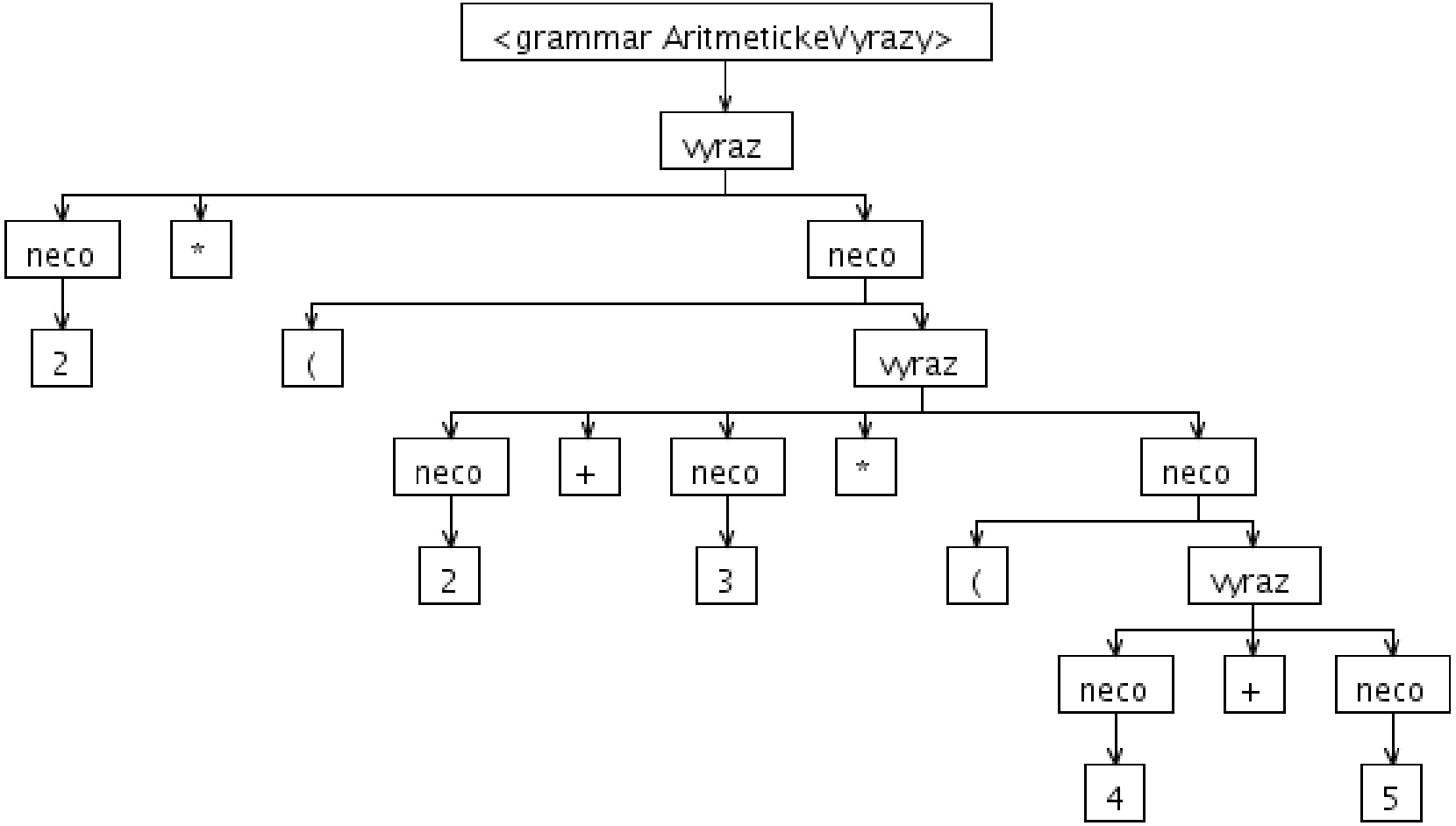
Příklad

- Jednoduchá gramatika, generující jazyk aritmetických výrazů
  - N={výraz, něco}
  - T={číslo, '+', '-', '*', '/', '(', ')'}
  - P={
    výraz→něco+výraz, výraz→něco-výraz, výraz→něco*výraz,
    výraz→něco/výraz,
    výraz→ něco,
    něco→(výraz),
    něco→číslo
    }
  - S={vyraz}
- Tato gramatika generuje např. „věty“:
  - číslo+číslo
  - číslo*(číslo-číslo)
  - číslo+číslo+číslo+číslo
  - ...

Derivační strom

- „Derivací“ nazýváme použití jednoho pravidla při generování věty gramatikou
- Derivační strom je grafické znázornění konstrukce jedné určité věty podle pravidel dané gramatiky
- Vrcholem stromu je startovací symbol
- Listy čtené zleva doprava tvoří větu vzniklou derivacemi dle konkrétních pravidel gramatiky
- Levý derivační strom sestavíme systematickým přepisováním nejlevějšího neterminálu.
- Pravý derivační strom sestavíme systematickým přepisováním nejpravějšího neterminálu
- Pokud se oba stromy rovnají, pak je daná derivace deterministická (v každém okamžiku můžeme použít pouze 1 pravidlo)

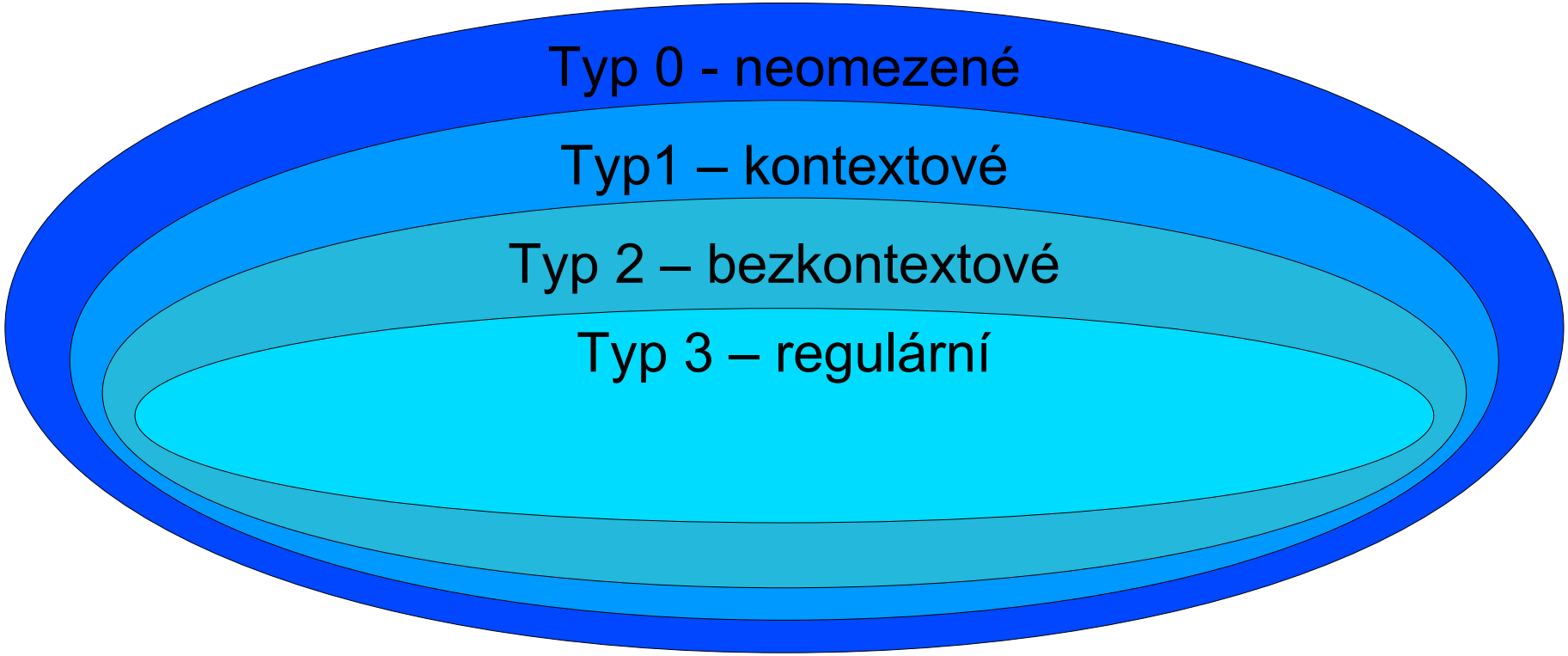
Příklad derivačního stromu, který vygeneruje výraz 2*(2+3*(4+5))



Rozklad věty neboli syntaktická analýza neboli parsing

- Rozklad věty s použitím gramatiky je proces opačný ke generování věty:
  - Vstupem je konkrétní věta jazyka, v předchozím příkladě např. určitý aritmetický výraz, který chceme vyhodnotit
  - Výstupem je rozhodnutí, zda věta je syntakticky správně nebo ne. V případě chyby pak ještě ukazatel na chybný (neočekávaný) terminální symbol
  - Pokud se nám podaří naprogramovat syntaktickou analýzu, pak pro interpretaci věty – v našem příkladě tj. vyhodnocení hodnoty výrazu – musíme ještě doprogramovat sémantiku (význam) jednotlivých částí gramatiky – toto ale teď nechme na později.

Chomského hierarchie gramatik



Chomského hierarchie gramatik

- **Gramatiky neomezené** generují jazyky, rozpoznatelné Turingovým strojem, tj. rekurzivně spočetné jazyky
- **Gramatiky kontextové** generují jazyky rozpoznatelné lineárně ohraničeným Turingovým strojem
  (tj. Turingovým strojem, který smí zapisovat pouze na prvních *n* buněk pásky, kde *n* je funkcí délky vstupního slova).
- **Gramatiky bezkontextové** generují jazyky jazyky rozpoznatelné nedeterministickým zásobníkovým automatem. Tyto gramatiky jsou v praxi nejvyužívanější, patří do nich gramatiky LL(n) a LR.
- **Gramatiky regulární** generují jazyky rozpoznatelné konečným automatem nebo regulárními výrazy.

Příklad

- G=({A, S}, {0 , 1}, {S→0A1, 0A→00A1, A→e}, S)
- Tato gramatika je neomezená
- Příklad derivací (řetězců, které mohou být generovány gramatikou):
- S => 0A1 => 01
  S => 0A1 => 00A11 => 0011
  S => 0A1 => 00A11 => 000A111 => 000111
- Jazyk generovaný touto gramatikou je tedy:
  $L(G) = \{0^n 1^n; n \geq 1\}$

Vliv Chomského na informatiku

- Chomský jako humanitní vědec neměl při své práci na mysli aplikaci v oblasti překladačů, šlo mu spíše o výzkum přirozených jazyků. Jeho práce ale ovlivnila informatiky, kteří hledali cesty pro formalizaci počítačových jazyků
  - **1958:** N. CHOMSKY, G. A. MILLER, "Finite state languages," Information and Control 1:2, 91-112
  - **1959:** N. CHOMSKY, "On certain formal properties of grammars," Information and Control 2:2, 137-167. Zde je mj. zavádí normální formu gramatiky a ukazuje, že neomezené jazyky mohou být rozpoznány Turingovým strojem (viz Church-Turing thesis, 1952)
  - **1962:** N. CHOMSKY, "Context-free grammars and pushdown storage," Quarterly Progress Report 65, 187-194, MIT Research Laboratory in Electronics,Cambridge, Massachusetts
  - **1963:** J. EVEY, "Application of pushdown store machines," Proceedings of the Fall, Joint Computer Conference, 215-227, AFIPS Press, Montvale, New Jersey
  - **1964**: S. Y. KURODA, "Classes of languages and linear bounded automata
  - **1965:** S. A. GREIBACH, "A new normal form theorem for context-free phrase structure grammars," Journal of the Association for Computing Machinery 12:1, 42-52
  - **1966:** S. GINSBURG, The Mathematical Theory of Context-free Languages, McGraw-Hill, New York
  - **1973**: Hopcroft a Ullman: „*Formal grammars and their relation to automata*“ – shrnuje možnosti zpracování jednotlivých Chomského gramatik jednotlivými typy konečných automatů

Teorie pro radostnější programování překladačů

- **1965**: D. E. KNUTH, "On the translation of languages from left to right," Information and Control 8:6, 607-639.
- **1969**:
  - A. J. KORENJAK, "A practical method for constructing LR(k) processors," Communications of the Association for Computing Machinery 12:11 (), 613-623.
  - F. L. DE REMER, "Generating parsers for BNF grammars," Proceedings of the 1969, Spring Joint Computer Conference, 793-799, AFIPS Press, Montvale, New Jersey.

Regulární gramatiky

- Regulární gramatiky obsahují pouze pravidla ve tvaru
  - Pravé lineární gramatiky
    $X \rightarrow wY$
    $X \rightarrow w$
    kde *X* a *Y* zastupuje nonterminální symbol a *w* je řetězec terminálů.
  - Nebo levé lineární gramatiky
    $X \rightarrow Yw$
    $X \rightarrow w$
- Regulární gramatika je ve standardní formě, pokud obsahuje pravidla ve tvaru
  $X \rightarrow aY$
  $X \rightarrow e$
- Kde *a* je právě jeden terminální symbol a *e* je symbol pro prázdný řetězec

Příklad regulární gramatiky

- Gramatika, generující jazyk čísel integer:
  $I \rightarrow +U \mid -U \mid 0U \mid 1U \mid \ldots 9U \mid 0 \mid 1 \mid \ldots 9$
  $U \rightarrow 0U \mid 1U \mid \ldots 9U \mid 0 \mid 1 \mid \ldots 9$
- Na příkladě tak jednoduchého jazyka je dobře vidět nevýhoda regulární gramatiky: příliš mnoho pravidel

Příklad: jednoduchý parser regulární gramatiky

- Parser naprogramujeme prostým přepisem pravidel gramatiky do programovacího jazyka dle následujícího postupu:
  - nonterminály implementujeme jako funkce. Uvnitř každé funkce implementujeme všechny pravé strany pravidel gramatiky, u nichž je daný nonterminál na levé straně, takto:
    - terminály implementujeme pomocí funkce expect(Set symbol), která očekává na vstupu zadanou množinu nonterminálů a pokud nějaký nonterminál z požadované množiny přijde, vrátí nám jej.
    - Non-terminály implementujeme jako zavolání funkce, reprezentující daný nonterminál
    - Pokud se nonterminál vyskytuje na levé straně více pravidel, při implementaci jeho funkce musíme dané pravidlo vybrat podle aktuálního nonterminálu, který jsme získali z funkce expect
- Příklad:

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#define PLUS 1
#define MINUS 2
#define MUL 4
#define DIV 8
#define DIGIT 16

int expect(int symbol) {
    int character;
    if ((character=getchar())==EOF) return EOF;
    if (isspace(character)) return 1;
    switch (character) {
        case '+':
            if ((symbol & PLUS) != 0) return 1;
            break;
        case '-':
            if ((symbol & MINUS) != 0) return 1;
            break;
        case '/':
            if ((symbol & DIV) != 0) return 1;
            break;
        case '*':
            if ((symbol & MUL) != 0) return 1;
            break;
        default:
            if (isdigit(character) && ((symbol & DIGIT) != 0))
            return 1;
            else return 0;
    }
    return 0;

}
```

```c
void unsign() {
    int error;
    if ((error=expect(DIGIT))==EOF) {
        printf("The string was integer!");

    } else if (!error) {
        fprintf(stderr, "Unexpected character in the unsign() nonterminal..");
        exit(EXIT_FAILURE);
    }

    unsign();
}
void u() {
    u();
    d();
}

void integer() {
    int error;
    if ((error=expect(PLUS|MINUS|DIV|MUL|DIGIT))==EOF) {
        printf("The string was integer!");
    } else if (!error) {

        fprintf(stderr, "Unexpected character in the integer() nonterminal..");
        exit(EXIT_FAILURE);
    }
    unsign();
}

int main()
{
    integer();
    return 0;
}
```

Bezkontextové gramatiky

- Přepisovací pravidla mají tvar
  $A \rightarrow \beta$
  kde
  - A je nonterminál
  - β je řetězec terminálů nebo neterminálů
- „Nekontextová“ znamená, že neterminál můžeme přepsat bez ohledu na kontext, ve kterém se vyskytuje – srovnej s kontextovým gramatikami
- **Příklad 1: gramatika čísel integer:**
  číslo → U | +U | −U
  U → D | DU
  D → 0 | 1 | ...9
- **Příklad 2: gramatika aritmetických výrazů**
  S → číslo | S + S | S - S | S * S | S/S | (S)

Priorita vyhodnocování pravidel

- Pokud potřebujeme, aby některé podvěty jazyka měly větší prioritu, než ostatní (např. násobení mělo větší prioritu, než sčítání), vyrobíme v gramatice nová pravidla tak, aby násobení bylo v derivačním stromu níž, než sčítání.
- Příklad:

LL(n) gramatiky

- Podtřída bezkontextových gramatik
- „LL“ znamená Left-left a je to vyjádření postupu při parsování jazyka popsaného LL gramatikou, kde se v derivačním stromu pohybujeme pouze zleva
- Syntaktická analýza pomocí LL gramatiky se implementuje velmi jednoduše pomocí tzv. rekurzivního sestupu
- Gramatika LL(1) zaručuje, že v každém okamžiku analýzy jazyka pomocí takové gramatiky je na základě současného stavu analýzy a posledního načteného terminálu jednoznačně určen další postup analýzy, tj. pravidlo, které bude vybráno v příštím kroku analýzy.
- Příklady jazyků třídy LL(1): Pascal, jazyk aritmetických výrazů generovaný speciálně upravenou gramatikou...
- Jazyky třídy LL(n) potřebují k výběru následujícího pravidla znát *n* následujících terminálních symbolů, tj. syntaktický analyzér musí implementovat nějaký mechanismus „look-ahead“.
  Příklad: gramatika aritmetických výrazů bez speciální úpravy je LL(2), protože ve většině případů potřebuje znát kromě aktuálního terminálu ještě následující terminál

Jak vyrobit LL(1) gramatiku?

- Příklad: gramatika pro aritmetické výrazy z předchozího příkladu není LL(1) ale LL(2) – v každém kroku analýzy nestačí dívat se na aktuální terminál, musíme se dívat ještě o jeden terminál dopředu. To nám parsování velmi zesložiťuje!
- Pokud je možné transformovat gramatiku na LL(1), pak ji transformujme – je to mnohem jednodušší, než programovat parser LL(n)!

Algoritmus transformace gramatiky na LL(1)

- Algoritmus spočívá v opakovaném použití těchto kroků na pravidla gramatiky:
  - Odstranění levé rekurze
  - Odstranění konfliktu first-first (levá faktorizace)
  - Levá rohová substituce (dosazení pravého kontextu)
  - Pohlcení terminálu

Odstranění levé rekurze

- Příklad gramatiky s levou rekurzí:
  S → **S + S**
  S → číslo
  S → (S)
- Postup odstranění:
  - Na levé straně rekurzivního pravidla: nahraďte S novým nonterminálem S'
  - Na pravé straně rekurzivního pravidla: vyjměte nonterminál na první pozici (zde „S“) a na konec pravidla přidejte nový nonterminál S'.
  - Ostatní pravidla začínající neterminálem S opíšeme a nakonec jim přidáme nový neterminál (S')
  - Vytvoříme nové pravidlo S → *e* (prázdný řetězec)
- Výsledek:
  S' → + S S'
  S → číslo S'
  S → (S) S'
  S' → e

Množiny First a Follow

- FIRST(X) je množina terminálů, které se mohou vyskytnout na začátku věty, zderivované z X
- FOLLOW(X) neterminálu je množina terminálů, které se mohou (např. ve větné formě) vyskytovat bezprostředně za tímto neterminálem (X)

Odstranění konfliktu first-first (levá faktorizace)

- Mějme pravidla
  A->ab
  A->ac
- U obou pravidel máme konfliktní množiny FIRST
- Z konfliktních pravidel utvoříme nová pravidla ponecháním konfliktního terminálu (v tomto případě 'a') a přidáním nového nonterminálu (v tomto případě A')
- Pro nový nonterminál (A') vytvoříme nová pravidla z původních konfliktních, tím že z nich odstraníme konfliktní terminál 'a'
- Výsledek:
  A->aA'
  A'->b
  A'->c

Odstranění konfliktu First-follow

- Mějme následující gramatická pravidla
  A->aBc
  B->*e*
  B->cd
- Pokud se B dá přepsat na prázdný symbol, pak při parsování řetězce „abcc“ u písmenka 'c' nevíme, jestli v nonterminálu B použít 2. nebo 3. pravidlo?
- V pravidle, odkud se dostal konfliktní terminál (v tomto případě 'c') do množiny follow, sloučíme 'c' s předcházejícím noterminálem (v tomto případě B) a tuto sloučeninu označíme jako nový nonterminál (v tomto případě '[Bc]')
- Pro nový nonterminál vytvoříme pravidla ze stávajících pravidel pro nonterminál (B) a na jejich konec přidám terminálu ('c')
  A->a[Bc]
  B->epsilon
  B->cd
  [Bc]->c
  [Bc]->cdc

Příklad LL gramatiky

- Gramatika pro jazyk aritmetických výrazů
- G=({S, EX, EX2, M, M2, T}, {id , +, -, *, /, (,)}, P, S)

P={

```
  S   -> EX; S | e

  EX  -> M EX2

  EX2 -> + M EX2 |
         - M EX2 | e

  M   -> T M2

  M2  -> * T M2|
         / T M2| e

   T-> num | + num | -num | ( EX ) | sin ( EX )
```

}

Analýza aritmetického výrazu

- S využitím gramatiky z předchozího příkladu:
- Terminální symboly budou zpracovávány pomocí konečného automatu (voláním funkce lexAnalyzer)
- Každý nonterminální symbol bude implementován jako funkce – viz příklad pro symboly `S a EX2`

```
void s() {
   check (plus|minus|lzavorka|num|ef);
   if (symbol!=ef) {
        ex();
        expect(strednik);
        s();
   }
}
void ex2() {
    check(plus|minus|strednik|pzavorka|ef);
    if (symbol==plus || symbol==minus) {
        symbol=lexAnalyzer();
        m();
        ex2();
    }
}
```