Žilinská univerzita v Žiline
Fakulta riadenia a informatiky

DIPLOMOVÁ PRÁCA

Študijný program: Informačné systémy
Zameranie: Aplikovaná informatika

Bc. Pavol Kovalík

Spätné generovanie UML diagramov

Vedúci: Mgr. Ing. Ľubomír Sadloň, PhD.
Registračné číslo: 217/2006

Žilina, máj 2007

Abstrakt

Výsledkom práce je modul nástroja UML.FRI pre spätné generovanie UML diagramov tried z kódu vyššieho programovacieho jazyka. Úvodná časť práce sa zaoberá popisom a vlastnosťami formálnych jazykov a možnosťami ich rozpoznania. Obsahom druhej časti práce je popis šablóny pre tri vybrané programovacie jazyky a implementácia samotného modulu.

Kľúčové slová: UML CASE, Spätné generovanie kódu, Formálny jazyk, Gramatika, Parser

Resume

The result of this diploma work is module for UML.FRI CASE tool for reverse engineering of higher programming language code. Beginning part of work treat of description and properties of formal languages and possibilities of their recognition. Subject of second part is description of template for three chosen programming language and implementation of module itself.

Key words: UML CASE, Reverse engineering, Formal language, Grammar, Parser

Prehlásenie

Týmto prehlasujem, že som diplomovú prácu vypracoval samostatne s odbornou pomocou školiteľa a s použitím uvedenej literatúry.

Žilina, máj 2007

.....

Poďakovanie

Týmto by som chcel poďakovať všetkým, ktorí mi umožnili vytvoriť túto diplomovú prácu. Osobitne by som sa chcel poďakovať vedúcemu diplomovej práce Mgr. Ing. Ľubomírovi Sadloňovi, PhD., ktorého rady a pripomienky prispeli k zvýšeniu kvality tejto práce.

Úvod

UML (Unified Modeling Language) je moderný štandardizovaný jazyk pre modelovanie procesov v priebehu celého cyklu vyvíjania informačného systému. UML zahŕňa nielen modelovanie softvéru, ale aj business process modeling a modelovanie realizácie a nasadenia. Jazyk UML je široko využívaný v praxi a jeho zvládnutie je častou podmienkou pre získanie zamestnania v obore návrhu a tvorby informačných systémov.

Pre Fakultu riadenia a informatiky ŽU vznikla potreba mať vhodný nástroj pre výučbu UML a princípov objektovo orientovaného programovania. Na trhu existuje niekoľko nástrojov. V akademickom roku 2005 začala skupina študentov Ján Janech, Marek Dovjak, Miroslav Špigura a Pavol Kovalík pod vedením Mgr. Ing. Ľubomíra Sadloňa PhD. vyvíjať nástroj pre tvorbu UML diagramov. V akademickom roku 2006 sa k projektu pripojila Martina Iskerková.

Cieľ práce

Cieľom tejto diplomovej práce je navrhnúť a implementovať modul pre spätné generovanie UML diagramov tried z kódu vyššieho programovacieho jazyka. Modul bude rozšírením pre CASE nástroj UML.FRI vyvíjaný v rámci projektovej výučby. Cieľom je zabezpečiť rozšíriteľnosť pre jednotlivé jazyky a typy diagramov pomocou šablón tak, aby nebolo nutné meniť kód jadra. Šablóna by mala byť jednotná pre generovanie kódu aj generovanie diagramov.

UML.FRI

Technológia

Projekt je implementovaný v programovacom jazyku Python s použitím grafického toolkitu GTK. Python je programovací jazyk založený na technológii virtual machine. Objekty podliehajú automatickej správe pamäti a sú uvoľňované okamžite po zrušení referencie. Tento jazyk bol zvolený z dôvodu jednoduchosti na naučenie a používanie, nakoľko bol navrhnutý na použitie neprogramátormi. Python bol vytvorený v roku 1990 Guido van Rossumom. Bol založený na dobrých vlastnostiach jazyka ABC. Neskôr bol

rozšírený o rôzne unikátne i prevzaté funkcie. Obsahuje množstvo modulov pre riešenie širokého spektra problémov.

Pre implementáciu grafického rozhrania je použitý GTK+ 2.x toolkit. GTK+ je multiplatformový a obsahuje kompletnú knižnicu „widgetov“. Spolu s nástrojom Glade ponúka možnosť rýchleho a efektívneho spôsobu návrhu užívateľského rozhrania. Samotný Glade je nezávislý od používaného jazyka, pretože navrhnuté užívateľské rozhranie netransformuje priamo do kódu, ale ukladá ho vo forme XML.

Na prístup ku GTK z programovacieho jazyka Python slúži knižnica PyGTK. Knižnica neslúži len ako čistý obal, ale implementuje funkcionalitu pre jednoduchšiu spoluprácu s jazykom Python. Knižnicu je možné jednoducho rozširovať o C/C++ moduly cez jednoduché definičné súbory. Spolu s podporou knižnice libglade umožňuje WYSIWYG (What You See Is What You Get) návrh GUI.

Stav projektu

Pôvodným cieľom projektu bolo vytvoriť editor len ako UML CROSS-LIFE CASE nástroj pre UML 2.0. Už v skorých fázach sa však začal vyvíjať k tomu, aby mohol pokryť podstatne širšiu množinu problémov. Nástroj sa tak stal plne konfigurovateľný CASE pre ľubovolný typ diagramov. Tým sa približuje schopnostiam nástrojom DSM (Domain Specific Modeler).

V prvom semestri ročníkového projektu boli navrhnuté a vypracované:

- Diagram tried. Štruktúra programu bola počas implementácie upravovaná a podľa nej bol upravovaný model, aby dokumentácia zodpovedala aktuálnemu stavu.
- Návrh štruktúry definičných súborov pre diagramy, elementy a spojenia vo formáte XML.
- Definičné súbory diagram tried a business diagram, a k ním vytvorené príslušné elementy a spojenia.

V priebehu nasledujúcich dvoch semestrov ročníkového projektu boli implementované moduly:

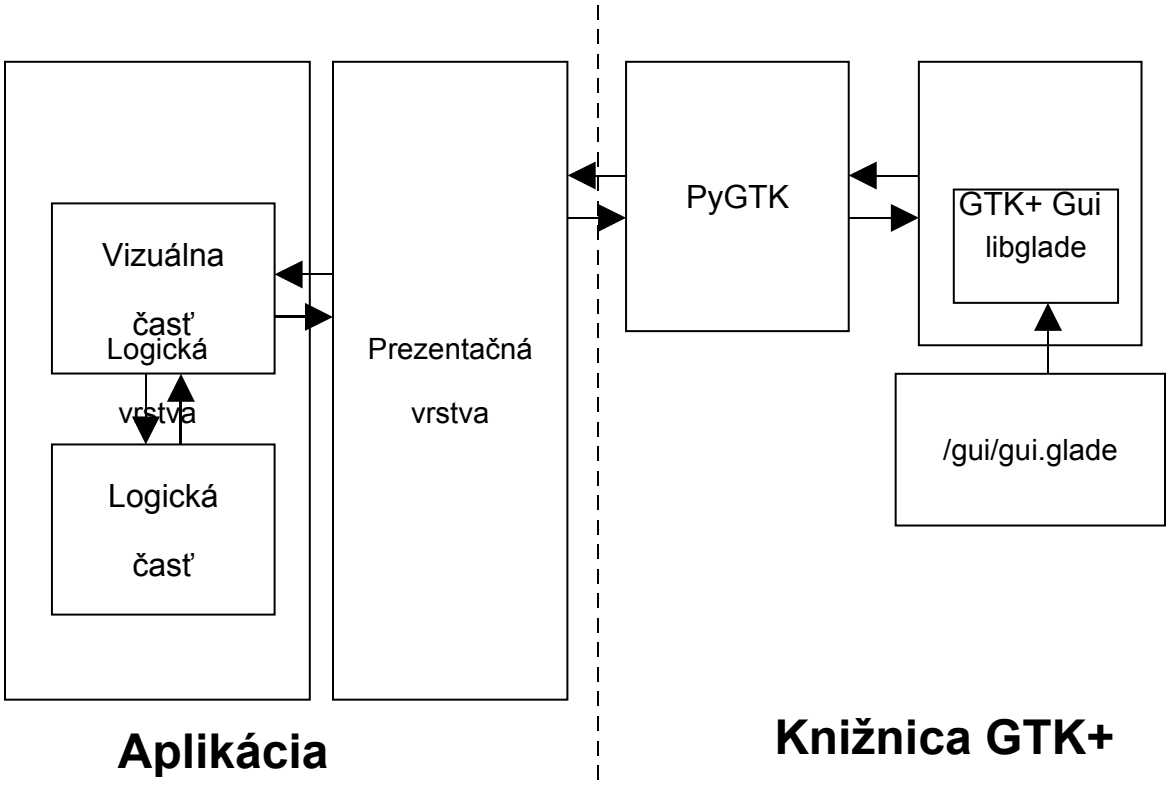
- Connections – načítanie definičného súboru spojenia, uchovávanie údajov.

- Elements – načítanie definičného súboru elementu, uchovávanie údajov.
- Strom projektu – organizácia diagramov a elementov v stromovej štruktúre.
- Drawing.
    - kresliaca plocha pre zobrazovanie v GTK a export obrázkov do SVG
    - vykresľovanie elementov a spojení
    - grafické prvky
- Math2D – operácie s objektmi v rovine.
    - transformačné matice pre posun, rotáciu, škálovanie objektov v rovine
    - definície základných geometrických útvarov v rovine (bod, čiara, zložená čiara, mnohouholník, obdĺžnik), zisťovanie vzdialeností a prienikov
    - prepočet súradníc bodov bezierovej krivky
- Clipboard – prenášanie elementov.
- Gui – obsluha grafického rozhrania aplikácie.

Ako diplomové práce boli pridané moduly pre:

- CodeEngineering – generovanie kódu programovacieho jazyka a dokumentácie (diplomová práca Miroslava Špiguru)
- ReverseEngineering – spätné generovanie diagramov z kódu programovacieho jazyka (predmet tejto diplomovej práce)

Aplikácia je rozdelená na dve základné vrstvy: Prezentačná vrstva a logická vrstva. Prezentačná vrstva (modul Gui) zabezpečuje komunikáciu s užívateľským rozhraním. Logická vrstva (modul Drawing) obsahuje vizuálnu časť a logickú časť. Vizuálna časť určuje zobrazenie elementov a spojení. Logická časť uchováva dáta elementov a spojení a ich usporiadanie v stromovej štruktúre.


Vizuálna
časť
Logická
vrstva
Logická
časť
Prezentačná
vrstva
PyGTK
GTK+ Gui
libglade
/gui/gui.glade
Aplikácia
Knižnica GTK+

Formálny popis jazyka.

Programovací jazyk, podobne ako reč, je definovaný súborom pravidiel. Tieto pravidlá určujú formálnu (syntaktickú) a významovú (sémantickú) stránku jazyka. Preto, ak chceme program v danom jazyku rozpoznať, musíme definovať tieto pravidlá. Z daných pravidiel navrhnúť parser, ktorý prečíta vstupný kód, rozpozná zoskupenia jednotlivých slov, a priradiť im správny význam.

V matematike, formálny jazyk L je zvyčajne definovaný abecedou a formovacími pravidlami. Abeceda formálneho Σ jazyka je množina symbolov, z ktorých je jazyk postavený. Za symboly môžeme považovať písmená, ale aj celé slová, poprípade iné znaky alebo prázdny znak „ε“. Formovacie pravidlá určujú, ktoré reťazce symbolov patria do daného jazyka. Tieto reťazce nazývame slová. Najčastejším spôsobom popisu formálneho jazyka sú gramatiky. Ku každému jazyku existuje gramatika, ktorá ho definuje, a naopak.

Gramatiky

Gramatika G, definujúca formálny jazyk L, je štvorica $G = (N, T, R, \sigma)$, kde

- N je konečná množina neterminálnych symbolov (neterminálov)
- T je konečná množina terminálnych symbolov (terminálov)
- R je konečná množina formovacích pravidiel. R pozostáva z dvojíc (u, v), kde $u, v \in (N \cup T)^*$. Dvojicu (u, v) často zapisujeme $u \rightarrow v$.
- σ je počiatočný neterminál, $\sigma \in N$
- $N \cap T = \varnothing$

Podľa zložitosti pravidiel sú gramatiky usporiadané v tzv. Chomského hierarchii gramatík:

- Frázová gramatika: $u \in (N \cup T)^* N (N \cup T)^*, v \in (N \cup T)^*$
- Kontextová gramatika: $u \in (N \cup T)^* N (N \cup T)^*, v \in (N \cup T)^*$ a platí $|u| \leq |v|$

- Bezkontextová gramatika: $u \in N$, $v \in (N \cup T)^*$
- Regulárna gramatika: $u \in N$, $v \in (T^* \cup T^*N)$

Krok odvodenia v gramatike je binárna relácia $\Rightarrow$ na $(N \cup T)^*$ definovaná takto: $x \Rightarrow y$ práve vtedy, keď existujú slová $a, b \in (N \cup T)^*$ a pravidlo $u \rightarrow v \in R$ také, že $x = aub$ a $y = avb$. Následovne, jazyk generovaný gramatikou G je množina $L(G) = \{ w \in T^* \mid \sigma \Rightarrow^* w \}$, kde $\Rightarrow^*$ je reflexívny a tranzitívny uzáver relácie $\Rightarrow$.

Neformálne povedané, v jednom kroku odvodenia vyberieme jeden alebo viac symbolov (slovo u). Slovo u musí obsahovať aspoň jeden neterminál. Môže obsahovať jeden alebo viac ďalších terminálov a neterminálov, v závislosti od toho, s akou gramatikou pracujeme. Slovo u nahradíme slovom v podľa vhodného pravidla. Jazykom je množina všetkých slov pozostávajúcich len z terminálnych symbolov, ktoré môžeme odvodiť postupným aplikovaním jednotlivých pravidiel gramatiky. Napríklad, v regulárnej gramatike $G = \{ N = \{\sigma, \alpha\}, T = \{a\}, R = \{ \sigma \rightarrow \alpha, \alpha \rightarrow a\alpha, \alpha \rightarrow a \}, \sigma \}$, slovo aa môžeme odvodiť postupným aplikovaním všetkých troch pravidiel následovne: $\sigma \rightarrow \alpha \rightarrow a\alpha \rightarrow aa$.

BNF zápis

BNF je akronym termínu „Backus-Naurova Forma“, ktorú John Backus prvýkrát použil k formálnemu zápisu pre syntax jazyka ALGOL. Peter Naur neskôr zjednodušil zápis. BNF je metasyntax používaný pre zápis bezkontextových gramatík.

Meta-symboly BNF sú:

- ::= znamená „definované ako“
- | znamená „alebo“
- < > ostré zátvorky rozlišujú názvy pravidiel (neterminálne symboly) od terminálnych symbolov, ktoré sú písane presne tak, ako sú reprezentované

Napríklad, BNF zápis veľmi malého jazyka môže byť

```
<program> ::= program
                  <declarations>
              begin
                  <statements>
              end ;
```

V praxi, mnoho autorov predstavilo niekoľko rozšírení BNF pre jednoduchšie použitie

- [ ] hranaté zátvorky pre nepovinné symboly
- { } zložené zátvorky pre opakujúce sa symboly
- ( ) obyčajné zátvorky určujú precedenciu operácií
- terminálne symboly pozostávajúce iba z jedného znaku sú uzavreté v úvodzovkách, aby ich bolo možné rozlíšiť od metasymbolov

Definíciu BNF môžeme vyjadriť v BNF zápise následovne:

```
syntax ::=      { rule }
rule ::=        identifier  "::="  expression
expression ::=  term { "|" term }
term ::=        factor { factor }
factor ::=      identifier |
                quoted_symbol |
                "("  expression  ")" |
                "["  expression  "]" |
                "{"  expression  "}"
identifier ::=  letter { letter | digit }
quoted_symbol ::= """ { any_character } """
```

Automaty

Automat je matematický model pre výpočtový prístroj, ktorý dokáže prijať alebo rozpoznať jazyk. Automat pracuje so symbolmi zo vstupnej abecedy Σ. Obsahuje konečný počet stavov K, kde $q_0 \in K$ je počiatočný stav a $F \subset K$ je množina akceptujúcich stavov. Prechodová funkcia δ určuje správanie sa automatu.

Automat postupne číta vstupné symboly slova a na základe prechodovej funkcie prechádza z jedného stavu do druhého. Ak sa po prečítaní celého vstupu automat nachádza v akceptujúcom stave, automat slovo príjme (akceptuje). Podľa toho, aké slová jazyka dokáže akceptovať, existuje niekoľko druhov automatov.

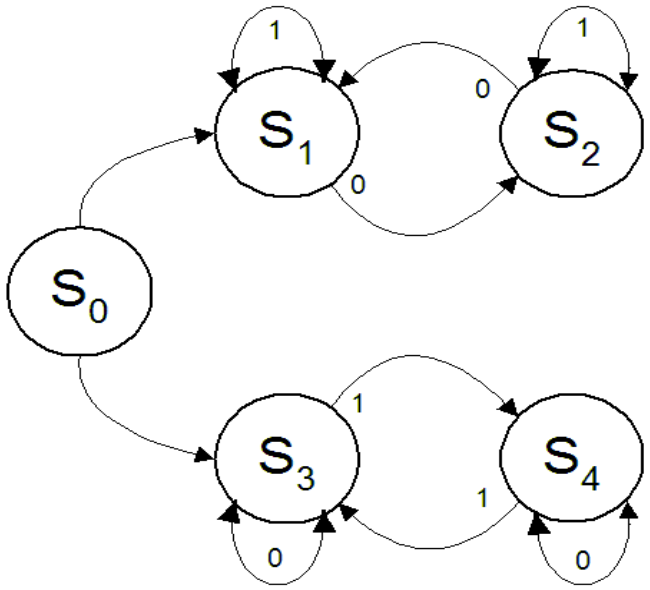
Konečný automat

Konečný stavový automat je najjednoduchšia forma automatu. Existujú dva varianty: deterministický konečný automat a nedeterministický konečný automat.

Formálne, deterministický konečný automat definujeme ako päticu $(K, \Sigma, \delta, q_0, F)$, kde K je množina stavov (abeceda), Σ je množina vstupných symbolov, $\delta : K \times \Sigma \rightarrow K$ je prechodová funkcia a $F \in K$ je množina akceptujúcich stavov.

Konfigurácia deterministického konečného automatu je prvok $(q, w) \in K \times \Sigma^*$, kde q je stav automatu a w je zvyšok vstupného slova.

Krok výpočtu deterministického konečného automatu je relácia $\vdash$ na konfiguráciách $(q, av) \vdash (p, v) \Leftrightarrow p = (q, a)$.



Nedeterministický konečný automat je definovaný podobne, ale prechodová funkcia je definovaná δ: $K \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^K$. To znamená, že automat môže prejsť pri prečítaní

jedného vstupného symbolu do viacerých stavov na raz a pokračovať viacerými rôznymi cestami výpočtu paralelne. Ku každému nedeterministickému konečnému automatu však možno zostrojiť deterministický automat akceptujúci rovnaký jazyk.

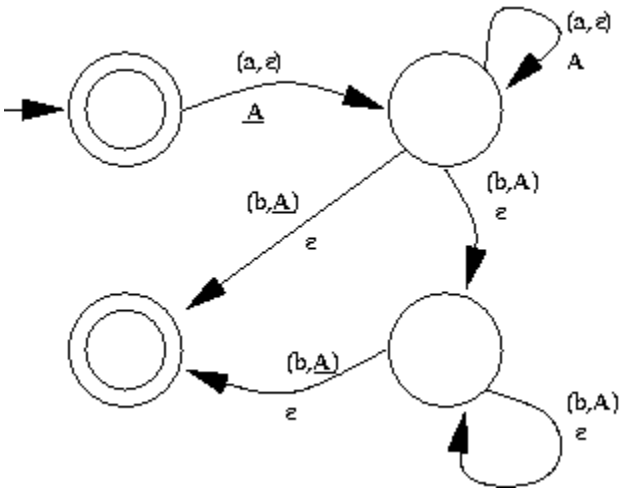
Výpočtová sila obidvoch týchto automatov rovnaká, aj keď veľmi obmedzená. Dokážu rozpoznať iba najjednoduchší jazyk a jemu prislúchajúcu gramatiku z Chomského hierarchie: Regulárny jazyk.

Zásobníkový automat

Nedeterministickým zásobníkovým automatom nazývame 7-icu $(K, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, kde K je abeceda vstupných symbolov, Γ je abeceda zásobníkových symbolov, $Z_0 \in \Gamma$ je symbol, ktorý je v zásobníku na začiatku výpočtu, $F \subseteq K$ je množina akceptačných stavov a $\delta: K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow 2^{K \times \Gamma^*}$ je prechodová funkcia. Prechodová funkcia pre deterministické zásobníkové automaty je $\delta: K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow K \times \Gamma^*$.

Konfiguráciu zásobníkového automatu nazývame trojicu $\eta \in K \times \Sigma^* \times \Gamma^*$.

Krokom výpočtu zásobníkového automatu A nazývame reláciu $\vdash$ na množine konfigurácií definovanú takto: $(q, au, \gamma z) \vdash (p, u, \gamma\beta) \Leftrightarrow (p, \beta) \in \delta(q, a, z)$.



Hovoríme o tzv. deštrukčnom čítaní. To znamená, že automat v jednom kroku môže prečítať vstupný symbol, zo zásobníka vyberie vždy jeden symbol, prejde do iného stavu a na vrch zásobníka vloží ľubovolný počet symbolov.

Vyššie triedy automatov

Pre zložitejšie gramatiky Chomského hierarchie existujú automaty, ktoré ich dokážu rozpoznať. Definovať programovací jazyk ako takúto gramatiku je zbytočné, pretože programovacie jazyky sú kvôli ľahšiemu parsovaniu zostrojené ako podmnožina bezkontextovej gramatiky. Preto tieto automaty spomeniem iba okrajovo.

Kontextovú gramatiku dokáže rozpoznať Lineárne ohraničený automat. Tento automat je podobný zásobníkovému automatu, ale na miesto zásobníka obsahuje pásku, na ktorej sa na začiatku nachádza vstupné slovo. Automat môže čítať a zapisovať na pásku, pohybovať sa po nej, ale jej dĺžka je ohraničená dĺžkou vstupného slova.

Najzložitejší automat, ktorý dokáže rozpoznať Frázové gramatiky, sa nazýva Turingov stroj. Je to v podstate „lineárne ohraničený automat“ s neohraničenou páskou.

Chomského hierarchia	Gramatiky / Jazyky	Automat
0	Rekurzívne vyčísliteľné (frázové) (Recursively enumerable - L_{RE})	Turingov stroj (Turing machine)
1	Kontextové (Context-sensitive – L_{CS})	Lineárne ohraničený (Linear-bounded – LBA)
2	Bezkontextové (Context-free – L_{CF})	Zásobníkový automat (Pushdown automaton – PDA)
-	Deterministické bezkontextové (Deterministic context-free – L_{DCF})	Deterministický zásobníkový a. (Deterministic pushdown a. – DPDA)
3	Regulárne (Regular – R)	Konečný (Finite) – NFA, DFA

Bezkontextové gramatiky

Programovacie jazyky sú najčastejšie definované pomocou bezkontextovej gramatiky. Pravidlá bezkontextovej gramatiky sú pomerne jednoduché, ale zabezpečujú dostatočnú výpočtovú silu.

Trieda deterministických bezkontextových jazykov je ostrou podmnožinou bezkontextových jazykov. Tento poznatok bude pre nás dôležitý pre budúcu implementáciu automatu a návrhu šablóny pre spracovanie programovacieho jazyka. Dôkaz tvrdenia vychádza z tzv. Pumpovacej lemy pre bezkontextové jazyky, ktorá hovorí: K ľubovolnému jazyku $L \in L_{CF}$ nad abecedou Σ existujú p, q také, že

$(\forall w \in L, |w| > p) \exists u, x, v, y, z \in \Sigma^*$ a platí

- $w = uxvyz$
- $xy \neq \varepsilon$
- $|xvy| \leq q$
- $\forall i \geq 0\ ux^i vy^i z \in L$

Nasledovne z Pumpovacej lemy môžeme dokázať že jazyk $L = \{a^n b^n c^n : n \geq 0\}$ nie je bezkontextový.

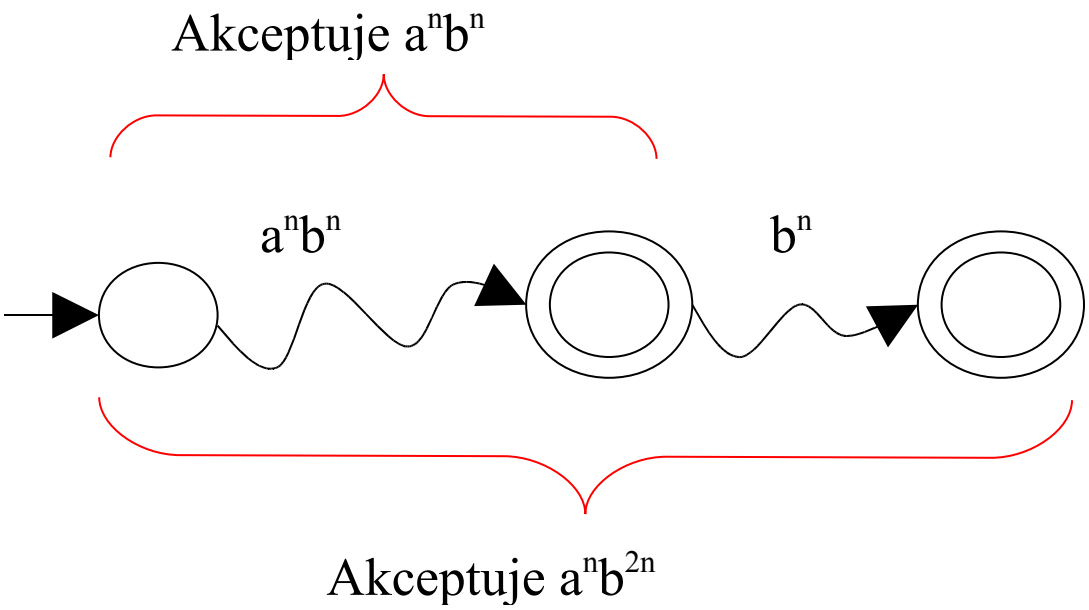
Predpokladajme, že L bezkontextový je. Pumpovacia lema platí pre L a M je číslo i z pumpovacej lemy. Zvoľme si $w = a^M b^M c^M$. Nech $w \in L$ a $|w| > M \geq K$. Z pumpovacej lemy, pre všetky reťazce w, kde $|w| > K$, existuje u, x, v, y, z také, že $w = uxvyz$ a $|vy| > 0$, a $|vxy| \leq M$, a pre všetky $n \geq 0$, $uv^n xy^n z$ patria L. Následne môžu nastať dva prípady:

- Buď x alebo y obsahuje dva alebo viac rôznych typov symbolov z {a, b, c}. Potom ux^2vy^2z nie je vo forme $a^n b^n c^n$.
- Ani jedno x alebo y neobsahuje dva lebo viac typov symbolov. V tomto prípade xy môže obsahovať najviac dva typy symbolov. Reťazec ux^0vy^0z zníži počet dvoch symbolov, ale počet výskytov tretieho symbolu zostane zachovaný.

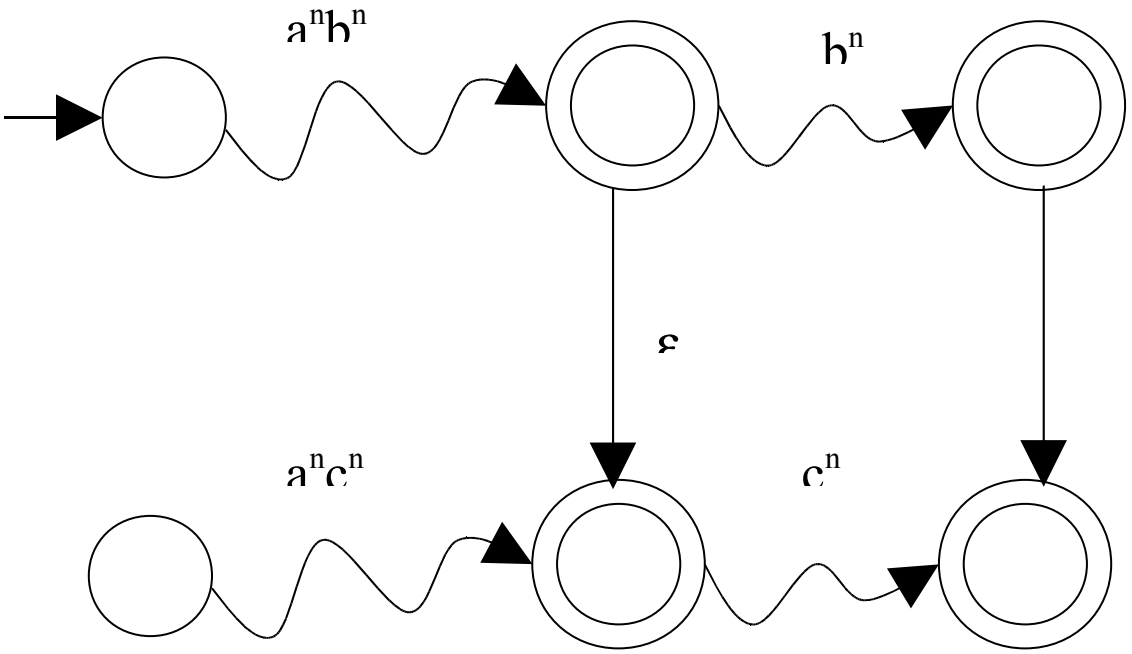
Obidva prípady pokrývajú všetky možnosti. Preto, nezávisle na tom, ako w rozdelíme, môžeme nájsť také uv^nxy^nz, ktoré nepatria do L.

Vráťme sa ale späť k pôvodnému tvrdeniu, že L_{DCF} je ostrou podmnožinou L_{CF}. Najjednoduchší spôsob dôkazu je nájsť taký jazyk L, že $L \in L_{CF}$ a $L \notin L_{DCF}$. Taký jazyk je napríklad $L = \{ a^nb^n \cup a^nb^{2n}; n \in N\}$. Bezkontextová gramatika pre L je $(\{\sigma, A, B\}, \{a, b\}, R, \sigma))$ $R = \{\sigma \rightarrow A \mid B, A \rightarrow \varepsilon \mid aAb, B \rightarrow \varepsilon \mid aBbb \}$

Predpokladajme, že je L deterministická bezkontextová gramatika. Preto existuje DPDA M, ktoré ju akceptuje.



Jednoducho môžeme zostrojiť upravený automat M' pre jazyk $L' = \{ a^nc^n \cup a^nc^n; n \in N\}$. Ak prepojíme akceptujúce stavy M s akceptujúcimi stavmi M' pomocou ε prechodov, dostaneme nasledovný automat:



Keďže $L \cup \{a^n b^n c^n : n \geq 0\}$ je akceptovaný DPDA, potom je bezkontextový. Z predchádzajúceho dôkazu však vyplýva, že sme dospeli k sporu.

LL gramatiky

LL(1) gramatiky sú trieda CF gramatík pre ktorú platí:

- Pravá strana každého pravidla začína terminálom
- Ak dve pravidlá majú rovnakú ľavú stranu, ich pravé strany začínajú rôznymi terminálmi

K formálnej definícií LL gramatiky potrebujeme definovať najprv dve funkcie, Follow(X) a First(X).

$Follow(X) = \{ A \mid S \Rightarrow^* \beta X\gamma, \gamma \Rightarrow^* a\eta, \eta \in (N \cup T) \} \cup \{\varepsilon \mid S \Rightarrow^* \beta X \}$

Hodnotu funkcie Follow(X) je množina obsahujúca všetky terminály, ktoré môžu byť vo vetných formách za symbolom X. V prípade, že X sa vyskytuje na konci niektorej vetnej formy, množina Follow(X) obsahuje prázdny reťazec.

$First(\beta) = \{ a \mid \beta \Rightarrow^* a\gamma, a \in T, \gamma \in (N \cup T)^* \} \cup \{ \varepsilon \mid \beta \Rightarrow^* \varepsilon \}$

Hodnota funkcia je množina terminálov, ktoré sa môžu vyskytnúť na začiatku reťazcov odvodzovaných z A. Ak sa z A dá odvodiť prázdne slovo, tak ho obsahuje aj First(A).

Bezkontextová gramatika G = (N, T, P, σ) sa nazýva LL(1) gramatika, keď pre každé $A \in N$ platí: ak $A \rightarrow \beta$, $A \rightarrow \gamma$ sú pravidlá v P, potom

- $First(\beta) \cap First(\gamma) = \varnothing$
- $Follow(A) \cap First(\gamma) = \varnothing$

Úprava CF gramatiky na LL gramatiku

Pri parsovaní CF gramatiky môže dojsť k situácii, kedy sme začali odvodzovanie podľa nesprávneho pravidla. Preto je potrebné najprv vstupnú gramatiku upraviť do LL formy. Keďže LL je podmnožinou DCF, existujú bezkontextové gramatiky ktoré nemožno upraviť. Na túto vlastnosť treba dbať pri definovaní gramatiky.

1. Ľavá faktorizácia: nahradíme pravidlá N ::= XY | XZ pravidlami typu N ::= X(Y | Z)
2. Odstránime ľavú rekurziu: N ::= X | NY a nahradíme N ::= X (Y)*
3. Nahradenie neterminálnych symbolov: Ak N ::= E nie je rekurzívne a je jediné pravidlo pre N, nahradíme všetky výskyty N v gramatike pomocou E
4. Vytvoríme množiny First a Follow pre jednotlivé pravidlá gramatiky.
   - First[ε] = ∅
   - First[t] = {t}, t je terminál
   - First[N] = First[E], kde N ::= E
   - First[E F] = First[E] ∪ First[F], ak E generuje ε
   - First[E F] = First[E], inak
   - First[E|F] = First[E] ∪ First[F], kde First[F] a First[E] nemajú spoločný prienik
   - First[E*] = First[E], kde First[E] je disjunktné Follow[E*]
   - Follow[N] = {t} v pravidle Nt, t je terminál
   - Follow[N] = First[F] v pravidle NF, F je neterminál

Konverzia CFG na PDA

Každá bezkontextová gramatika môže byť konvertovaná na zásobníkový automat, ktorý akceptuje jazyk ňou generovaný. Vytvorený PDA spracuje najľavejšiu deriváciu slova odvodeného gramatikou.

Predpokladajme, že máme zadanú CFG, T je množina terminálov, a N je množina neterminálov. Nech σ je počiatočný symbol na zásobníku. Potom môžeme zostrojiť PDA nasledovným spôsobom:

- Vstupná abeceda je T
- Existujú iba 2 stavy P a Q, kde P je počiatočný stav a Q je jediný akceptačný stav
- Abeceda zásobníka je $T \cup N \cup \{\$\}$
- Prechodová funkcia je následovná“
    - $(P, \$, \varepsilon) \rightarrow (Q, \sigma)$
    - Pre každé pravidlo gramatiky $A \rightarrow \alpha$ je prechod $(Q, \varepsilon, A) \rightarrow (Q, \alpha)$
    - Pre každý terminálny symbol „a“ je prechod $(Q, a, a) \rightarrow (Q, \varepsilon)$

Úprava pre LL gramatiku

Princíp pre konverziu LL gramatik je veľmi podobný predchádzajúcemu riešeniu. Keďže každé pravidlo začína terminálom, postačuje upraviť iba prechodovú funkciu nasledovne:

- Pre každé pravidlo gramatiky $A \rightarrow a\alpha$ je prechod $(Q, a, A) \rightarrow (Q, \alpha)$

Všeobecný princíp parsovania LL gramatiky

```
push(S)
read(symbol)
while stack not empty do
      if top-of-stack is terminal:
           if top-of-stack == symbol:
                pop()
                read(symbol)
           else:
                error
     elif top-of-stack is non-terminal:
           if LL[top-of-stack, symbol]:
                pop()
                push(LL[top-of-stack,symbol])
           else:
                error
if symbol != $:
     error
```

Implementácia modulu

Návrh šablóny

Syntax popisu jazyka je vo formáte XML, aby bol definičný súbor jednotný s ostatnými definičnými súbormi pre UML.FRI. Stromová štruktúra elementov XML nahradzuje pravidlá BNF zápisu. Niektoré elementy definujú špeciálne pravidlá, ktorých rozpoznanie vytvorí príslušný UML objekt v stromovej štruktúre projektu.

Jednotlivé pravidlá a sú definované v koreňovom elemente Template

- Template –koreňový element šablóny
    - diagram –názov diagramu
    - language –názov jazyka
    - indents –určuje, či je jazyk závislý na odsadení
    - casesensitive –určuje, či jazyk rozlišuje veľké a malé písmena

Následovné elementy nahradzujú zápis BNF.

- Terminal –terminálny symbol
    - id – názov terminálu (token)
    - value – regulárny výraz (lexéma)
- Rule – syntaktické pravidlo
    - id – názov pravidla
- Flow – položky v pravidle nasledujú postupne
- Optional – nepovinná položka
- Loop – opakovanie položiek v pravidle
    - separator – znak oddeľujúci položky medzi jednotlivými cyklami
- Alternate –položky v pravidle sú vo vzťahu „alebo“

- Ref – odkaz na pravidlo alebo terminál
    - value – názov pravidla alebo terminálu

Elementy pre špeciálne preddefinované symboly

- br – Nový riadok (Break)
- WhiteSpace – medzera
- Indent – odsadenie kódu
- Directory –adresár
    - name –názov adresára, ktorý sa používa pri generovaní kódu
    - value –regulárny výraz, podľa ktorého je možne odvodiť názov
    - prefix –predpona
    - sufix –prípona
- File –súbor
    - name –názov súboru, ktorý sa používa pri generovaní kódu
    - value –regulárny výraz, podľa ktorého je možne odvodiť názov
    - prefix –predpona
    - sufix –prípona

Elementy definujúce špeciálne pravidlá, ktoré definujú vytvorenie jednotlivých objektov UML diagramu:

- Element –UML element
    - id –identifikátor elementu
- ConnectionLoop –pridá spojenie.
    - id –atribút spojenia
    - value –hodnota atribútu spojenia (typ spojenia)

- separator –znak oddeľujúci položky medzi jednotlivými cyklami

- PropertyLoop –pridá atribút alebo vlastnosť
  - id –názov vlastnosti
  - separator –znak oddeľujúci položky medzi jednotlivými cyklami
  - parse –metóda parsovania textu vlastnosti
- Property –priradí hodnotu vlastnosti elementu alebo spojeniu
  - id –názov vlastnosti
  - value –pravidlo alebo terminál, podľa ktorého je možne odvodiť hodnotu vlastnosti
  - newLine – symboly, ktoré sa pri generovaní vložia na začiatok nového riadku

Prepis elementov šablóny do BNF

Element Flow

XML

```
<Rule id=”parameter-list”>
     <Flow>
          <Ref value=”parameter1” />
          <Ref value=”parameter2” />
     </Flow>
</Rule>
```

BNF

```
<parameter-list> ::= (<parameter1> <parameter2> )
```

Element Loop

XML

```
<Rule id=”parameter-list”>
```

```
     <Loop>
          <Ref value=”parameter” />
     </Loop>
</Rule>
```

BNF

```
<parameter-list> ::= { <parameter> }
```

Element Optional

XML

```
<Rule id=”parameter”>
     <Ref value=”identifier” />
     <Optional>
          = <Ref value=”expression” />
     </Optional>
</Rule>
```

BNF

```
<parameter> ::= <identifier> [ “=” <expression>]
```

Element Alternate

XML

```
<Rule id=”class-member”>
     <Alternate>
          <Ref value=”method” />
          <Ref value=”attribute” />
     </Alternate>
</Rule>
```

BNF

```
<class-member> ::=   <method> | <attribute>
```

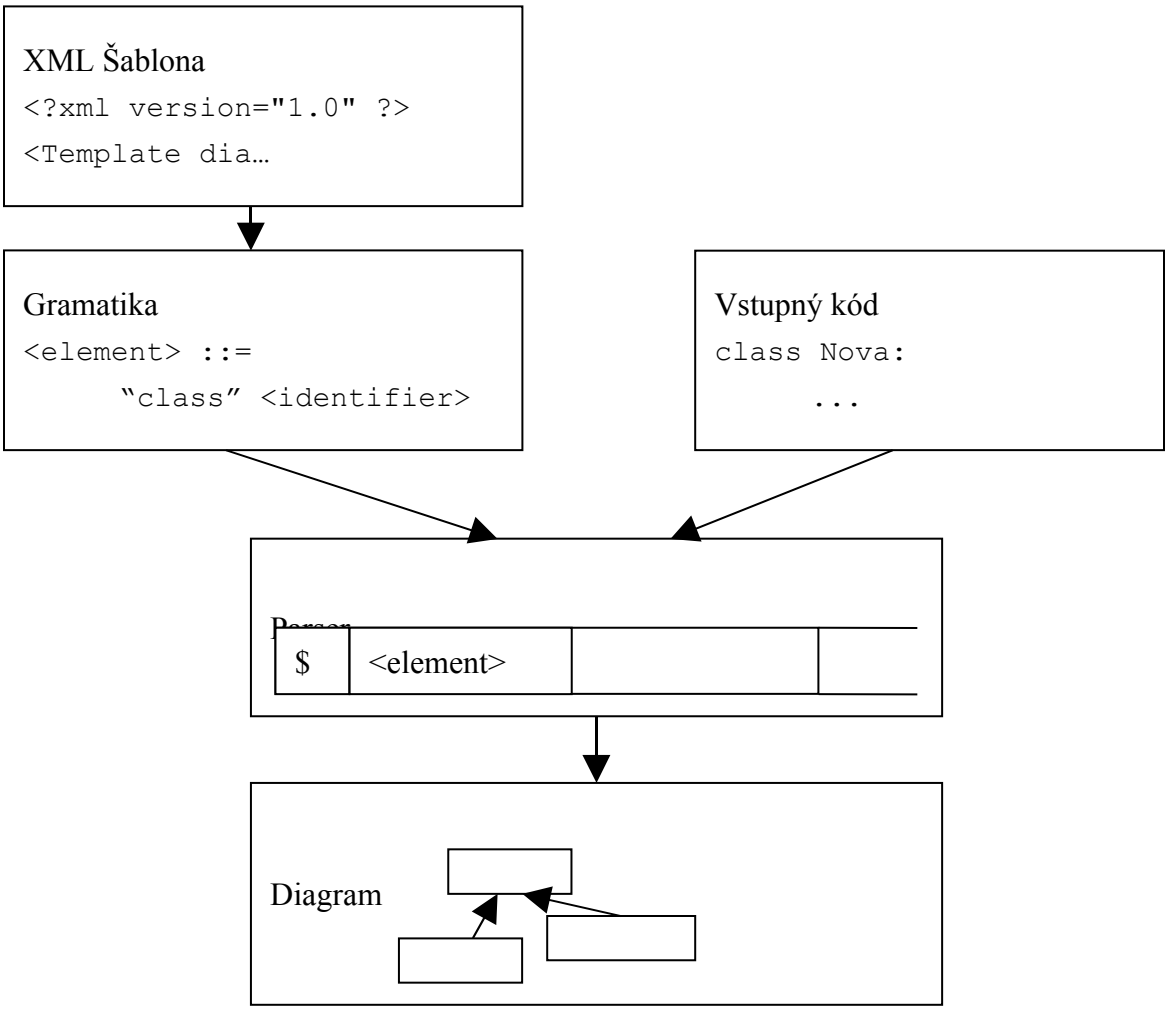
Princíp fungovania modulu

Užívateľ si v Project View vyberie uzol, ktorý slúži ako koreň pre generovanie elementov. Následovne pomocou menu zvolí akciu Project → Source Code Engineering → Generate Diagrams. V dialógu určí adresár obsahujúci kód a vyberie programovací jazyk zdrojového kódu.

Následne, modul vykoná následovné kroky:

1. Načíta šablóny. Toto zabezpečuje modul pre generovanie kódu, ktorý naprogramoval Miroslav Špigura v rámci svojej diplomovej práce
2. Vytvorí bezkontextovú gramatiku a opraví ju na LL gramatiku, ak je to možné.
3. Parser spracuje zdrojový kód podľa gramatiky
4. Vytvorí príslušné elementy

Celý postup je znázornený na nasledovnom obrázku:



Štruktúra modulu

ReverseEngineering

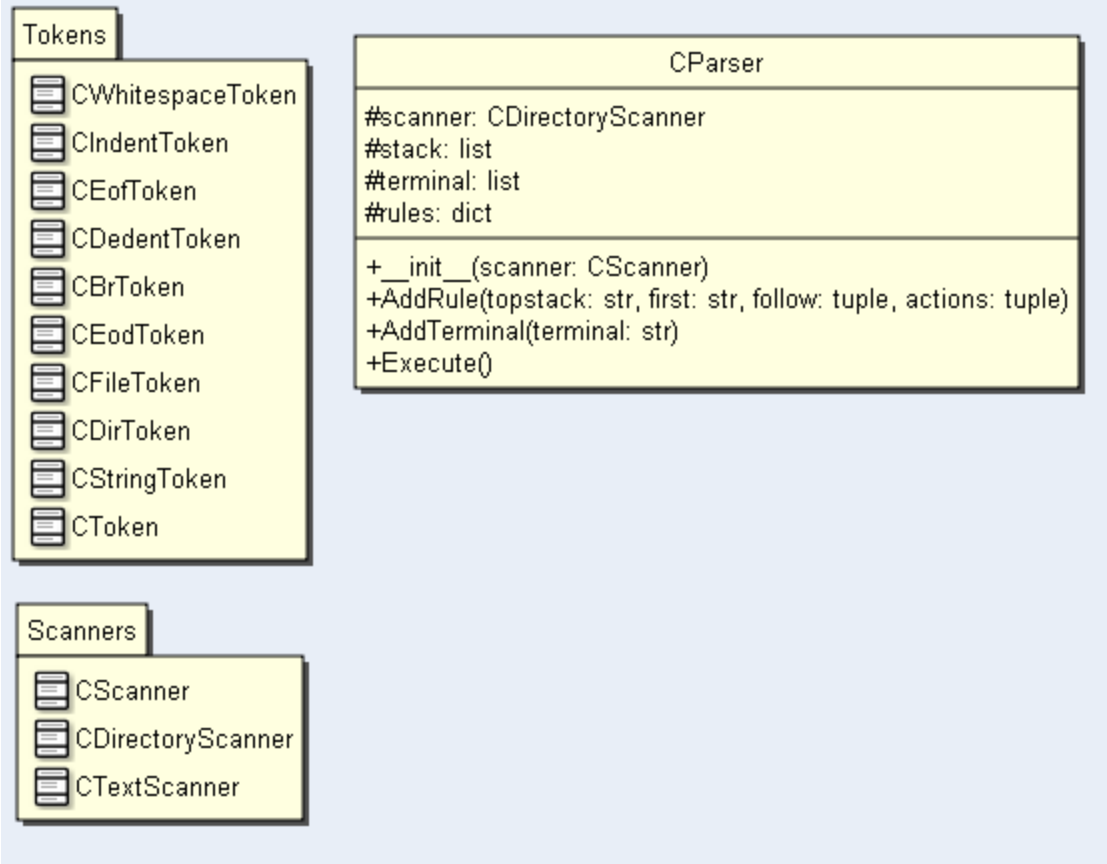
CParser je trieda zabezpečuje spracovanie vstupu. V jednotlivých krokoch postupne spracováva tokeny, ktoré poskytuje scanner. Princíp parsovania je popísaný v kapitole "Všeobecný princíp parsovania LL gramatiky".

Atribúty:

- scanner –objekt skenuje vstup a vracia rozpoznané tokeny (terminály) a lexémy
- stack – zásobník, do ktorého sa ukladajú ešte nespracované terminály a neterminály
- terminals –zoznam terminálnych symbolov
- rules – slovník pravidiel. Kľuč položky je dvojica neterminálny symbol na vrchu zásobníka a terminál načítaný zo vstupu. Hodnotou položky n-tica symbolov, ktoré sa majú pridať na zásobník.

Metódy:

- __init__ –inicializuje inštanciu triedy
    - scanner –viď atribúty
- AddRule –pridá položku medzi pravidlá
    - topstack –neterminál navrchu zásobníka
    - first –terminál načítaný zo vstupu
    - follow –symboly, ktoré sa majú pridať na vrch zásobníka
    - actions –akcie, ktoré sa majú vykonať pri rozpoznaní pravidla
- AddTerminal –pridá terminálny symbol
- Execute –Vykoná parsovanie vstupu



Tokens

Tokeny sú kódy pre neterminálne symboly, ktoré sa podarilo rozpoznať vo vstupnom texte.

CToken

Základná trieda, ktorá zabezpečuje rozpoznanie tokenu a k nemu prislúchajúcej lexémy vo vstupnom texte. Ak metóda *match* nájde

Atribúty:

- terminal –názov terminálneho symbolu
- regexp –objekt zkompilovaného regulárneho výrazu

Metódy:

- __init__ –inicializuje inštanciu triedy
    - terminal –názov terminálneho symbolu

    - regexp –reťazec definujúci regulárny výraz
- GetTerminal –vráti reťazec terminálneho symbolu
- match –pokúsi sa nájsť regulárny výraz vo vstupnom texte.
    - text –vstupný text
    - pos –počiatočná pozícia, na ktorej ma začať hľadať zhodu

CFileToken

Token reprezentuje začiatok súboru, ktorého názov zodpovedá zadanému regulárnemu výrazu.

CDirToken

Token reprezentuje vstup do adresára, ktorého názov zodpovedá zadanému regulárnemu výrazu.

CEodToken

„End of directory“ reprezentuje vyčerpanie všetkých možných adresárov alebo súborov, do ktoré môže otvoriť.

CStringToken

Token rozpoznáva ľubovolný reťazec zodpovedajúcemu danému regulárnemu výrazu.

CWhitespaceToken

Token „whitespace“ zodpovedajúci medzerám. Viacero medzier za sebou predstavuje len jeden token CWhitespaceToken.

CBrToken

Token „br“ zodpovedajúci koncu riadku (znak LF). Kombinácia viacerých znakov LF a medzier je reprezentovaná ako jediný token.

CIndendToken

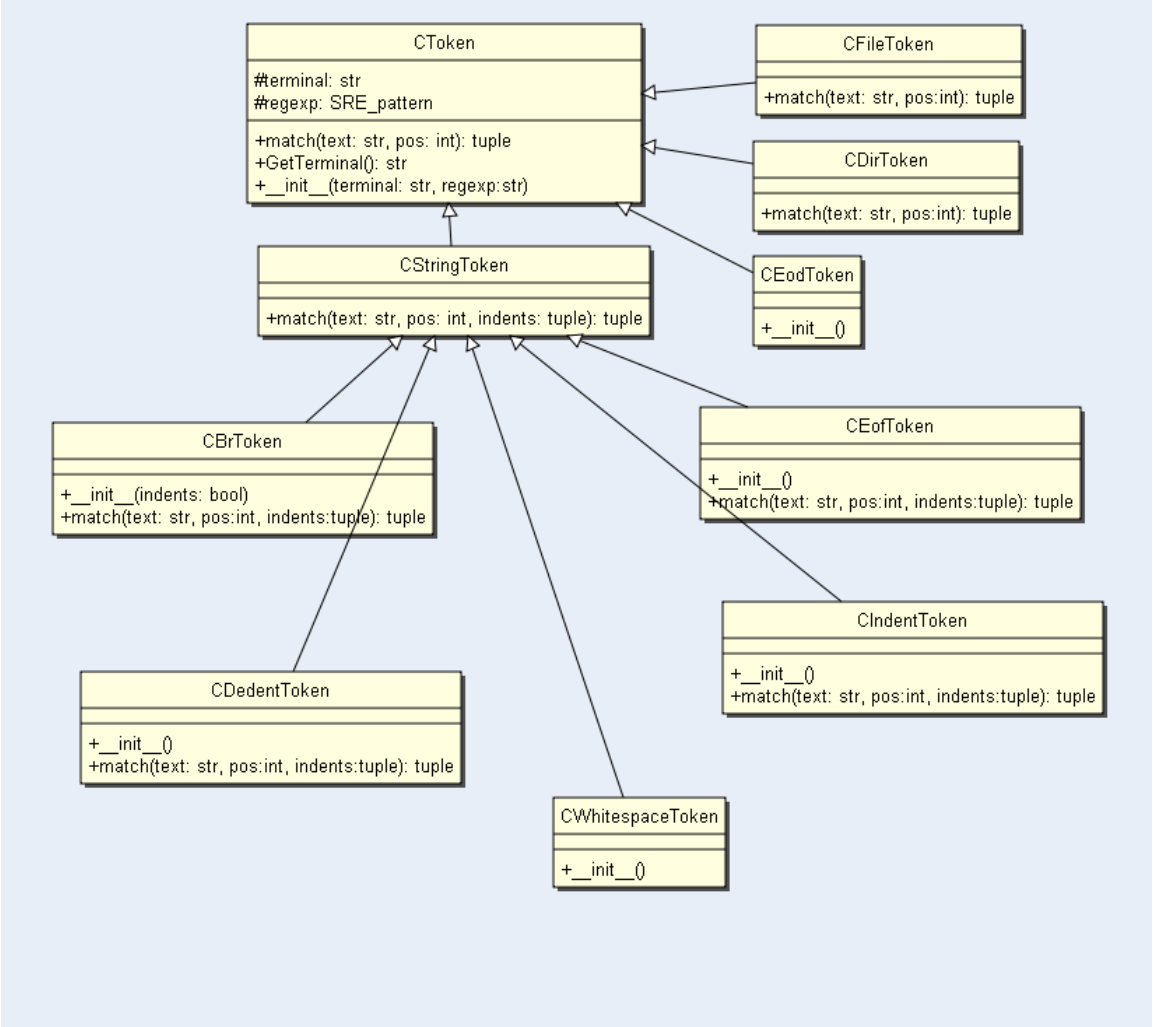
Token „indent“ reprezentuje zvýšenie odsadenia textu.

CDedentToken

Tokent „dedent“ reprezentuje zníženie odsadenia textu.

CEofToken

Token „eof“ reprezentuje koniec súboru.



Scanners

Skener zabezpečuje rozpoznávanie postupnosti tokenov.

CScanner je abstraktná trieda, ktorá implementuje spoločné atribúty a metódy pre nasledujúce dve triedy

Atribúty:

- tokens –zoznam už rozpoznaných tokenov
- restriction –zoznam obmedzení

- ignore – zoznam syntakticky bezvýznamných tokenov pre daný jazyk (napríklad medzery, oddeľovače riadkov a podobne)
- patterns – slovník obsahujúci všetky rozpoznateľné tokeny

Metódy:

- __init__ – inicializuje inštanciu triedy
    - patterns, ignore – viď atribúty
- token – vráti token na i-tej pozícií
    - i – pozícia tokenu (možno zadať hodnotu (-1) pre nasledujúci)
    - restrict – zoznam tokenov, na ktoré sa má pri vyhľadávaní obmedziť
- scan – prototyp metódy na vyhľadanie nasledujúceho tokenu
    - restrict – zoznam tokenov, na ktoré sa má pri vyhľadávaní obmedziť

CTextScanner vyhľadáva tokeny v reťazci.

Atribúty:

- input – vstupný reťazec
- pos – pozícia, na ktorej má nájsť ďalší token
- indents – zoznam jednotlivých odsadení nového riadku textu

CDirectoryScanner prechádza adresárovou štruktúrou. Postupne priradzuje tokeny adresárom a súborom, do ktorých vstúpil. Keď otvorí súbor, vytvorí novú inštanciu triedy CTextScanner, ktorý následne skenuje obsah súboru. Keď CTextScanner prejde celým súborom (posledný nájdený token je “eof”), CDirectoryScanner uzavrie súbor a pokračuje prehľadávaním adresára.

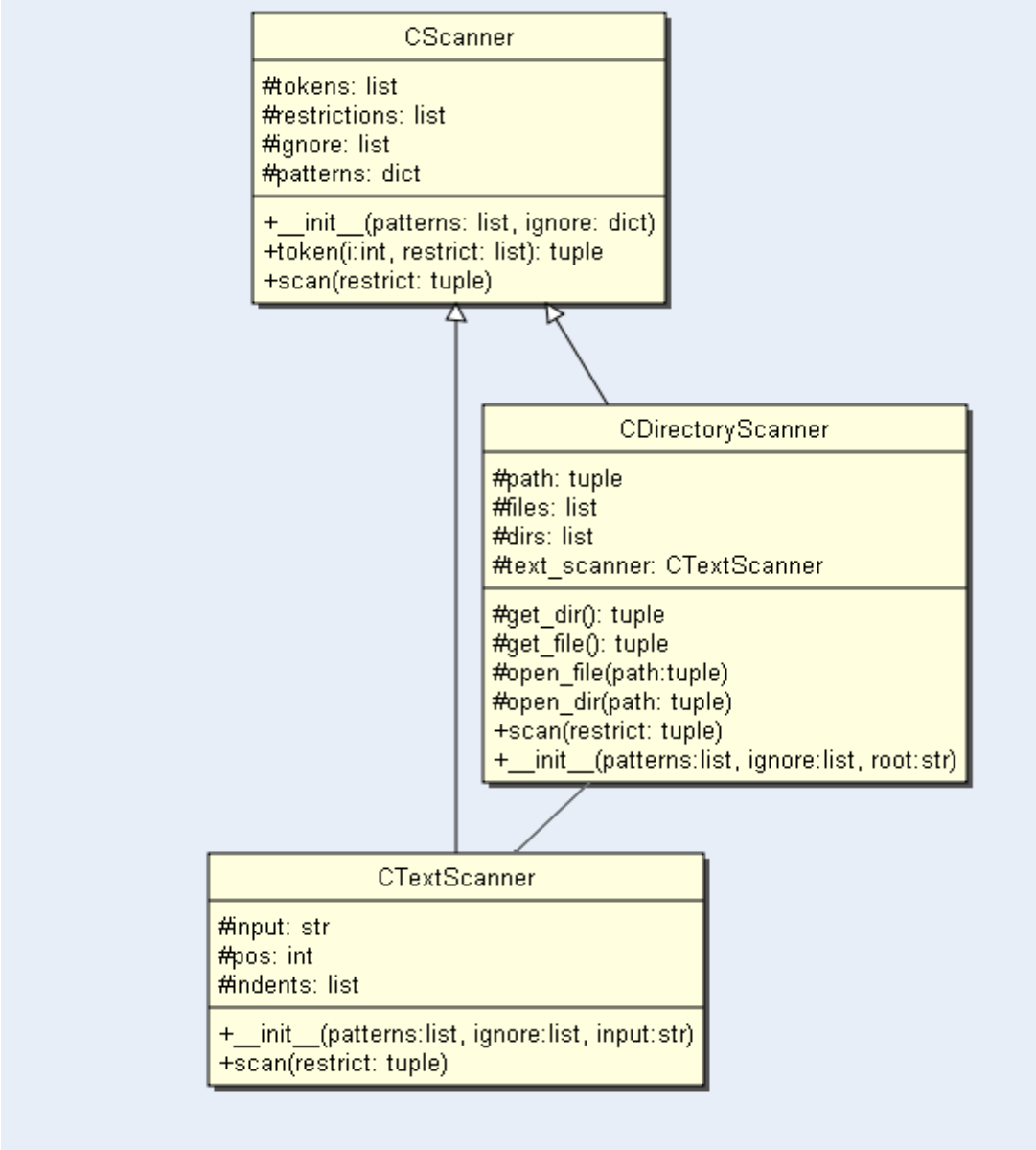
Atribúty:

- path – n-tica obsahujúca cestu k aktuálnemu adresáru
- files – zoznam už otvorených súborov
- dirs – zoznam navštívených adresárov

- text_scanner –CTextScanner, ktorý spracováva posledne otvorený súbor

Metódy:

- get_dir –vyberie ďalší adresár a vráti n-ticu obsahujúcu kompletnú cestu k nemu
- get_file –vyberie ďalší súbor a vráti n-ticu obsahujúcu kompletnú cestu k nemu
- open_file –otvorí súbor a inicializuje text_scanner
- open_dir –vojde do podadresára



Popis vybraných jazykov

C++

C++ je komerčne široko používaný jazyk. Jazyk bol štandardizovaný roku 1998 ako ISO/IEC 14882:1998, v súčasnosti vo verzii ISO/IEC 14882:2003.

<class-name>::=

<identifier>

| <template-id>

<class-specifier>::=

<class-head> "{" [<member-specification>] "}"

<class-head>::=

<class-key> [<identifier>] [<base-clause>]

| <class-key> <nested-name-specifier> <identifier> [<base-clause>]

<class-key>::=

"class"

| "struct"

| "union"

<member-specification>::=

<member-declaration> [<member-specification>]

| <access-specifier> “:“ [<member-specification>]

<member-declaration>::=

[<decl-specifier-seq>] [<member-declarator-list>] “;“

| <function-definition> [";"]

| <qualified-id> ";"

| <using-declaration>

| <template-declaration>

<member-declarator-list>::=

<member-declarator>

| <member-declarator-list> "," <member-declarator>

<member-declarator>::=

<declarator> [<pure-specifier>]

| <declarator [<constant-initializer>]

| [<identifier>] ":" <constant-expression>

<pure-specifier>::=

"=" "0"

<constant-initializer::=

"=" <constant-expression>

Object Pascal

Existuje niekoľko implementácií jazyka Object Pascal. Najznámejšia a najčastejšie využívaná je implementácia Borland Delphi od spoločnosti Borland, najmä vďaka úzkemu prepojeniu s IDE. Keďže sa mi nepodarilo zohnať najnovšiu špecifikáciu gramatiky Delphi, uvediem preto gramatiku pre implementáciu Free Pascal. Free Pascal založený na dialekte Turbo Pascal, Delphi a obsahuje niekoľko konštrukcií z dialektu MacPascal.

<class-type> ::=

[“packed”] “class” [<heritage>] { component-list } end

<heritage> ::=

“(“ <class-identifier> [implemented-interfaces] “)”

<implemented-interfaces> ::=

{ “,” <interface-identifier> }

<component-list> ::=

[<visibility-specifier>] { <field-definition> }

{ <field-definition> | <property-definition> }

<field-definition> ::=

<identifier-list> “:” <type> “;”

<method-definition> ::=

<method-header> “;” [<method-invocation> “;”]

<method-invocation> ::=

“virtual” [“;” “abstract”] | “override”

<method-header> ::=

<function-header> | <procedure-header>

| <constructor-header> | <destructor-header>

<visibility-specifier> ::=

“private” | “protected”

| “public” | “published”

<function-header> ::=

“function” <identifier> <parameter-list> “:” <result-type>

<procedure-header> ::=

“procedure” <identifier> <parameter-list>

<constructor-header> ::=

“constructor” <identifier> <parameter-list>

<destructor-header> ::=

“destructor” <identifier> <parameter-list>

<parameter-list> ::=

“(“ { <parameter-declaration> } “)”

Python

Jazyk Python bol navrhnutý a prvýkrát uvedený jeho autorom Guido van Rossumom. Syntax jazyka je oproti predchádzajúcim dvom jazykom pomerne jednoduchší. Špecialitou jazyka je, že na oddelenie blokov kódu nepoužíva zátvorky alebo kľúčové slová, ale odsadenie.

```
identifier ::=
        (letter|"_") (letter | digit | "_")*
suite ::=
        stmt_list NEWLINE
        | NEWLINE INDENT statement+ DEDENT
expression_list ::=
        expression ( "," expression )* [","]
classdef ::=
        "class" classname [inheritance] ":"
        suite
inheritance ::=
        "(" [expression_list] ")"
classname ::=
        identifier
funcdef ::=
        [decorators] "def" funcname "(" [parameter_list] ")"
        ":" suite
parameter_list ::=
        (defparameter ",")*
        (~"*" identifier [, "**" identifier]
        | "**" identifier
```

```
        | defparameter [","] )
defparameter ::=
        parameter ["=" expression]
```

Záver

V diplomovej práci som implementoval parsovací systém schopný spracovať časť syntaxu programovacieho jazyka. Modul umožňuje vykonanie niekoľkých základných operácií s entitami diagramov. Jedná sa o vytvorenie elementu, vytvorenie spojenia medzi elementmi a priradenie atribútu elementu alebo spojeniu. Systém je navrhnutý tak, aby bolo možné šablóny rozšíriť o ďalšie prvky, prípadne vytvoriť novú šablónu pre iný druh diagramu alebo iný programovací jazyk. Vzhľadom na komplexnosť špecifikácie jazyka UML, samotný parsovací systém je pomerne jednoduchý a jeho možnosti nepokrývajú plne možnosti UML.

Nástroj UML.FRI je uverejnený pod GPL licenciu a dúfam, že budem môcť na jeho ďalšom vývoji pokračovať aj v budúcnosti.

Použité zdroje

[1] Súbor prednášok z predmetu Formálne Jazyky a Automaty, Univerzita Komenského

[2] http://foja.dcs.fmph.uniba.sk/kompilatory/materialy.php –Stránka predmetu Kompilátory

[3] http://cs.wwc.edu/~aabyan/SEscripts/Compiler/ - Language design and implementation

[4] http://www.cs.utsa.edu/~danlo/teaching/cs4713/lecture/lecture.html –CS 4713 Compiler Writing

[5] http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html – What is BNF notation?

[6] http://www.garshol.priv.no/download/text/bnf.html –BNF and EBNF: What are they and how do they work?

[7] http://en.wikipedia.org/wiki/ –Wikipédia

[8] http://python.org/doc/ –Python Documentation

[9] http://www.freepascal.org/docs/ref.pdf –Free Pascal: Reference guide.

[10] http://www.csci.csusb.edu/dick/c++std/cd2/gram.html –C++ Grammar summary

OBSAH

ÚVOD....5

Cieľ práce....5

UML.FRI....5

Technológia....5

Stav projektu....6

FORMÁLNY POPIS JAZYKA....9

Gramatiky....9

BNF zápis....10

Automaty....11

Konečný automat....12

Zásobníkový automat....13

Vyššie triedy automatov....14

Bezkontextové gramatiky....15

LL gramatiky....17

Úprava CF gramatiky na LL gramatiku....18

Konverzia CFG na PDA....19

Úprava pre LL gramatiku....19

Všeobecný princíp parsovania LL gramatiky....20

IMPLEMENTÁCIA MODULU....21

Návrh šablóny....21

Prepis elementov šablóny do BNF....23

Princíp fungovania modulu....25

Štruktúra modulu....27

ReverseEngineering....27

Tokens....28

Scanners....30

POPIS VYBRANÝCH JAZYKOV....33

C++....33

Object Pascal....34

Python....36

ZÁVER....38