Zpracování jazyka Toki Pona

Tomáš Brukner

Ročníková práce 2008/2009

Abstrakt

Tato ročníková práce se zabývá oborem informatiky známým jako zpracování přirozeného jazyka na příkladu zpracování jednoduchého jazyka Toki Pona. Vysvětluje obecný postup překladu a techniky, které jsou v tomto postupu zahrnuty – a to syntaktickou a lexikální analýzu a překladový stroj.
V první části je stručný popis jazyka Toki Pony – jeho filozofie, základní gramatické principy, slova a věty. V druhé části je popis syntaktické a lexikální analýzy spolu s popisem nástrojů, které jsem k tomu použil – lex a yacc. Ve třetí části se zabývám myšlenkou překladového stroje, který jsem sám navrhl – popisuji zde princip fungování a způsob použití.
Výsledkem této práce je kromě následujícího textu i program, který lze použít na překlad vět v jazyce Toki Pona do angličtiny.

Abstract

This annual work is concerned with section of informatics known as natural language processing and it uses example of processing simple language named Toki Pona. It explains common routine of translation as well as techniques, that are involved in this routine – a syntactic and lexical analysis and a translation engine.
The first part of the work shows brief definition of Toki Pona language – it's philosophy, basic grammatical principles, basic words and sentences. In the second part of the work I describe the syntactic and the lexical analysis along with description of tools I used – lex and yacc. Idea of the translation engine is described in the third part of the work. I personally designed the engine and I wrote the principle of operation and how to use it.
The result of this work is beside following text program, which can be used to translate sentences in Toki Pona language to English.

Prohlášení
Prohlašuji, že jsem tuto ročníkovou práci vytvořil samostatně pod vedením Petera Novotného, a že jsem v seznamu použité literatury uvedl veškeré zdroje, které byly při tvorbě práce využity.
Na Kladně dne 15. 1. 2009 Tomáš Brukner

Obsah

Zpracování jazyka Toki Pona

Úvod do problematiky

Zpracování přirozeného jazyka je oblast informatiky, respektive programování, která dle mého názoru není příliš rozšířena mezi lidmi, kteří se o celou sféru informatiky zajímají. Na druhou stranu, výsledky této práce jsou vidět již nějakou dobu, přestože si to někteří lidé neuvědomují. Začal bych dnes již tak samozřejmou věcí, jako je kontrola gramatiky v textových editorech. Právě takováto funkce je dle mého názoru předchůdcem věcí, které nás čekají – a to kontrola gramatiky na té úrovni, že když napíšete špatně čárku či zaměníte měkké i za tvrdé y ve shodě podmětu s přísudkem, že program bude schopný tuto chybu rozpoznat, upozorní vás na ni, případně ji i za vás opraví. A právě vidina možností, které jsou před námi, mě přiměla k tomu se této oblasti více věnovat, a to právě formou ročníkové práce.

Historie tohoto oboru začíná v porovnání s dalšími obory informatiky již celkem dávno – a to v souvislosti s kompilátory kódu. Nejdřív si však položíme otázku – co to je kód a co je kompilátor? Kód je informace v nějaké formě, která je někomu srozumitelná. Vezme-li v úvahu například zdrojový kód, tak to je ukázka informace, které rozumí programátor, ale nikoliv počítač. Naopak strojovému kódu rozumí počítač, ale nikoliv programátor (nebo alespoň ne většina). V tomto okamžiku přichází „magický nástroj“ zvaný kompilátor (někdy též nazýván překladač), který překládá zdrojový kód do strojového kódu.[1]

Překladač (nebo překladatel, jedná-li se o člověka a ne o stroj) je slovo, které je známé i široké veřejnosti. Jedná se o člověka či věc, který vezme informaci v jednom jazyce a převede ji (s větší či menší úspěšností zachování přesného podání informace a jejího obsahu) do jazyka jiného. Bohužel poznámka, kterou jsem uvedl v závorkách, je velmi důležitá. Uvedu příklad – jak byste přeložili do angličtiny hodit s sebou? Throw with myself? Myslím, že by vám angličané nerozuměli (korektní překlad je v tomto případě *get a move on[2]*). Tohle je jedno z úskalí překladu – mnohoznačnost kontextu. Naštěstí u programovacích jazyků tohle není – tam není důležité „co tím chtěl básník říci“. Zde má veškerý obsah právě jeden smysl. To překlad programovacích jazyků značně zjednodušuje.

Jak funguje kompilátor

Existuje několik různých variant, ale obecně se jedná o několik procesů jdoucích za sebou. První je takzvaná lexikální analýza, která převede celý kód na značky. Další je syntaktická analýza, která tyto značky přepíše do mezikódu. Teď se může provádět jeho případná optimalizace. Jako poslední krok se vygeneruje z tohoto mezikódu kód výsledného jazyka. To je již velmi snadné.[1]

Pro další vysvětlení funkce kompilátoru předvedu to samé na překladu věty. Fáze jsou totiž téměř stejné, až na to, že neexistuje vhodný mezijazyk. První fáze, lexikální analýza, by v našem případě znamenalo určení slovního druhu u všech slov a u některých ještě určení další vlastnosti (u podstatných jmen pád, u sloves osobu atp.). Druhá fáze, syntaktická analýza, by byl v tomto případě větný rozbor. Určili bychom co je podmět, přísudek, jestli má věta nějaké další větné členy a podobně. Dalším krokem by bylo přeložení jednotlivých slov a jejich uspořádání do vhodné věty ve výsledném jazyce. Zde ale nastává největší problém – věta má totiž sama o sobě význam, který ovlivňuje překlad. To je asi ten největší rozdíl od překladu programovacích jazyků a také dle mého názoru i největší problém pro absolutní (nejenom gramaticky, ale i významově správný) strojový

překlad. Otázka je, jak lze dosáhnout správného přeložení významu.

Zkusme se zamyslet, jak to dělá mozek. Mozek informaci pochopí. Tím pádem je schopný říct tu samou informaci v jiném jazyce, aniž by výrazně změnil smysl. Pochopením informace jsem se vůbec nezabýval, protože neexistují metody, jak to obejít bez použití inteligence (což je v případě strojového překladu inteligence umělá) a tudíž jsem se snažil udělat překlad takový, aby byl především gramaticky správný a alespoň přibližně vystihoval původní význam.

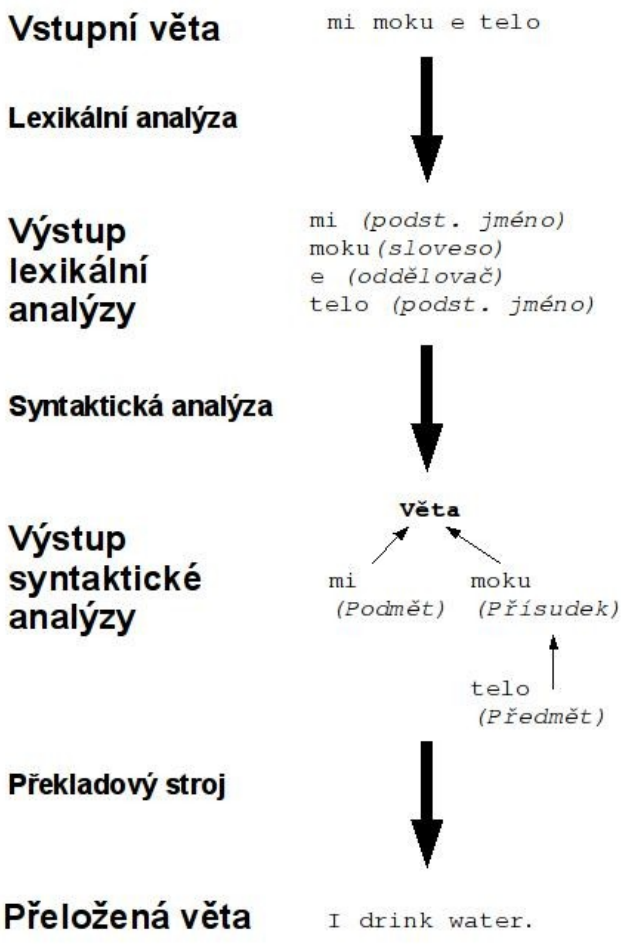
Proč Toki Pona

Je to velmi jednoduchý jazyk (viz první kapitola) a tudíž je lexikální i syntaktická analýza naštěstí relativně snadná. Jediné, co tedy bylo potřeba vymyslet, byl způsob překladu rozebrané věty.

Celou práci jsem se rozhodl rozdělit na tři hlavní kapitoly. První je **Toki Pona jako jazyk**, kde ukazuji základy tohoto jazyka. Další je **Syntaktická a lexikální analýza – lex a yacc**, kde popisuji způsob zpracování vstupu, a konečně třetí kapitola, **Překladový stroj**, ve které prezentuji můj vlastní návrh překladového stroje. Na závěr publikuji některé testy a dosažené výsledky.

Cíl práce

Cílem mojí práce bylo vytvořit jakýsi koncept překladového stroje, který by překládal po větách a ukázat jeho funkčnost na překladu jazyka Toki Pona. Jeho vstupem je rozebraná věta, a tudíž je potřeba nejdřív ji zanalyzovat; díky čemuž se Toki Pona stává vhodným kandidátem na prezentaci tohoto stroje, a to právě díky svým vlastnostem, které umožňují její relativně snadnou syntaktickou analýzu.



Ilustrace 1: Grafické znázornění procesu překladu

Toki Pona jako jazyk

Co to je vlastně Toki Pona za jazyk? Zeptáte-li se člověka na ulici, těžko vám dá uspokojivou odpověď. Dovolím si tedy kratičkou citaci:

*„**Toki pona** (v toki poně znamená **dobrý jazyk** nebo **jednoduchý jazyk**) je umělý jazyk, který vytvořila a v roce 2001 poprvé prezentovala kanadská tlumočnice a lingvistka Sonja Elen Kisa.“[3]*

Historií jazyka a její autorkou se zabývat nebudu, pro mě je důležitý jazyk samotný. Tady jsou některé základní poznatky:

- v Toki Poně se používá pouze 14 písmen (9 souhlásek a 5 samohlásek)
- výslovnost je stejná jako v češtině (to jest pokud bychom vyjmenovávali abecedu v češtině, tak dané písmeno se v Toki Poně vyslovuje stejně)
- všechna oficiální slova jsou psaná malými písmeny
- oficiálních slov je pouze 118 – to jsou slova v oficiálním slovníku (neoficiální slova jsou především jména, místa, náboženství a podobně, která naopak vždy začínají velkým písmenem)

Zde bych seznam takových velmi základních vlastností ukončil a zaměřil se na jednu věc, která velmi stěžuje překlad, a to je mnohoznačnost. Tím, že je jazyk postaven na tak malém počtu slov (pro porovnání – čeština má přes 250 000 slov[4]), vyplývá nutná potřeba zjednodušování. V Toki Poně nelze v podstatě nic vyjádřit jednoznačně (možná tak některé lidské části, třeba oko), všechna ostatní slova jsou mnohoznačná. Většina z nich se navíc dá použít jako více slovních druhů – obvyklé je použití zároveň jako podstatné jméno, jako přídavné jméno a jako sloveso. Je pravda, že každé slovo představuje nějaký „okruh“ věcí či témat, které toto slovo vyjadřuje bez ohledu na slovní druh. Jeden příklad za všechny – slovo *ike* znamená jako přídavné jméno zlý, špatný (a další slova s podobným významem), jako podstatné jméno negativita, zlo a jako sloveso být zlý, ošklivý atd. Existuje zde ještě jeden problém s překladem a to při větné stavbě – může se stát, že daná věta má více významů prostě proto, že je více možností větného rozboru – tyto případy já neřeším (již z toho důvodu, že nástroj yacc není schopen to zpracovat oběma způsoby) a je dobré se jim proto vyhnout; to je doporučováno i v samotné učebnici jazyka.

Myslím, že je celkem logické, že v Toki Poně není žádné skloňování ani časování. Všechna slova jsou tedy psána vždy stejně a jsou i vždy stejně čtena. Bez problémů lze tedy domyslet, že vztahy ve větě se tvoří převážně různými pomocnými slovy – předložkami a dalšími. Možná bude trochu více překvapující, že v Toki Poně nejsou ani časové tvary slovesa (existují ale způsoby, jak vyjádřit, že se něco stalo v budoucnosti, přítomnosti či v minulosti). Dalším významným faktem je absence slovesa být – ve větě, ve které bychom normálně použili sloveso být v češtině, se v Toki Poně použije jako sloveso předmět (já jsem zlý → já zlý) a sloveso být si člověk musí domyslet.

Když se člověk zamyslí nad tou vlastností, že každé slovo může být spousta slovních a větných druhů, tak si nejspíš domyslí, že pro správné pochopení věty v jazyce je potřeba kromě pomocných slov ještě jedna věc – a to relativně striktní pořadí větných členů a slovních druhů. Tato vlastnost, která je pro přirozené jazyky dle mého názoru neobvyklá (samozřejmě to nikde není úplně volné, ale třeba v češtině je spousta možností, kdy lze prohodit slova či větné celky), mi umožnila lépe jazyk zpracovat – a to z toho důvodu, že lze použít nástroje jako yacc. Tyto nástroje pracují s bezkontextovými jazyky – tedy s jazyky, u kterých nepotřebuji znát slova okolo, abych provedl správně syntaktickou analýzu.[5]

Myslím, že je čas se začít zabývat nějakými konkrétními slovy a větami v Toki Poně, na nichž

vysvětlím základní principy v tomto jazyce. Nejdříve ale ilustrující obrázek všech písmen v Toki Poně s obrázky slov začínajících na dané písmeno:

Kresba 1: Graficky znázorněná některá slova v Toki Poně (zdroj: http://bknight0.myweb.uga.edu/toki/lesson/aei.jpg)

Slovník základních slov

Nejdříve slovníček slov, které využiji v následujícím textu:

- *mi* – já, my
- *sina* – ty, vy
- *ona* – on, ona, ono (třetí osoba)
- *jan* – někdo, osoba
- *pona* – dobrý, jednoduchý; opravovat, spravit
- *moku* – jíst, pít; jídlo, pití
- *suno* – Slunce, světlo
- *telo* – voda, tekutina
- *suli* – velký, dlouhý, důležitý
- *ilo* – nástroj, zařízení, stroj
- *pipi* – brouk, hmyz, pavouk
- *wile* – chtít
- *lukin* – koukat, dívat se; vize, pohled
- *nasa* – hloupý, blbý, divný, opilý
- *utala* – válka, bitva, bojovat

Nyní ukázka vět (již bylo zmíněno, že zde není sloveso být):
mi pona. – Já jsem dobrý/důležitý.
sina moku. – Ty jíš/piješ.

Podmět je první ve větě a následuje rovnou přísudek. V praxi toto ovšem platí jenom pro věty, ve kterých je podmět já nebo ty (*mi* nebo *sina*). V ostatních větách se používá slovo *li* na oddělení podmětu a přísudku. Příklad:
jan li pona. – Člověk/lidé jsou dobří.
suno li suli. – Slunce/světlo je velké/důležité.

Nyní není těžké si povšimnout vlivu mnohoznačnosti na překlad – u všech těchto vět není jednoznačný. Je ještě celkem snadné rozlišit množné a jednotné číslo (použitím slova *mute*, viz dále), ale u sloves to je jiné. Jak tedy člověk přijde na správný význam? Těžko – musí si to domyslet z kontextu. Jeden příklad za všechny – *moku* znamení jídlo i jíst. V druhé větě je tedy možný překlad jak Ty jíš/piješ, ale též Ty jsi jídlo/pití. Co s tím? Odpověď přímo z učebnice:

„Koneckonců, jak často slyšíte někoho říkat „Já jsem jídlo.“? Doufám, že ne příliš často!“[6]

Když už jsme u jídla a pití – jak tyhle dvě akce rozlišíme? Obvykle předmětem – ten se dává za přísudek a před něj se dává slovo *e*, což je další oddělovač.
mi moku e telo. - Já piji vodu.
jan wile e pipi. - Člověk chce brouka.

Použití infinitivů je celkem snadné – prostě se dají za sloveso, co již ve větě je, čímž rozšíří přísudek.
mi wile lukin e suno. - Já chci koukat na slunce.

Základní slovní spojení

Poslední věcí, kterou se budu zabývat v tomto krátkém průletu jazykem Toki Pona, jsou přídavná jména – ty se dávají vždy hned za podstatná jména (třeba jako ve francouzštině[1]) a rozšiřují jejich význam. Důležité je, že záleží na jejich pořadí – změna pořadí přídavných jmen mění význam. Příklady:
jan pona – přítel (dobrý člověk)
ilo moku – nástroj pro jedení či pití (vidlička, lžíce, nůž; hrníček, sklenička)
mi mute – my

Zde je vidět znovu mnohoznačnost a ještě jedna věc, která je důležitá pro překlad – interpretace významu. Jedná se vlastně o kombinaci více slov pro vyjádření konkrétní věci. Problém pro strojový překlad je následující: stroj bez inteligence nemůže vědět, že dobrý člověk je přítel. Proto můj program na to používá slovník ustálených slovních spojení (viz třetí kapitola).

Nyní ukážu, proč tak záleží na pořadí slov.
jan utala nasa – hloupý voják
jan nasa utala – bojující hlupák

Myslím, že rozdíl je jasný. Stejně se používají i přivlastňovací zájmena:
ilo mi – můj nástroj
telo sina – tvoje voda

Poslední věcí k vysvětlení budou příslovečná určení – ta následují sloveso stejně jako infinitivy (které se ovšem používají jen u některých sloves, třeba u *wile*):
mi utala ike – Bojuji špatně.
ona li wile mute e pipi – Chce hodně brouka.

Tím bych završil krátký průlet jazykem Toki Pona. Pro komplexnější pochopení tohoto jazyka doporučuji učebnici.[6]

1 Ve francouzštině se některá přídavná jména dávají před podstatná jména – v Toki Poně ne, tam je vždy toto pořadí: podstatné jméno, přídavné jméno, další přídavná jména (bez čárek).

Syntaktická a lexikální analýza – lex a yacc

Pojmy syntaktická a lexikální analýza jsem již trochu nakousl v úvodu, ale myslím, že je čas, si je znovu zopakovat. Když to vezmeme z celkového pohledu, tak obě tyto analýzy jsou součástí překladu, a to jeho první části – pochopení významu původní věty. Mějme to jako dva procesy, které mají vstup a výstup. První věc, kterou máme, je věta v Toki Poně. Tato věta je vstupem lexikální analýzy. Tato analýza rozdělí větu na slova a každému slovu přiřadí nějaký příznak – že to je sloveso, podstatné jméno, přídavné jméno, jestli se má dané slovo vůbec překládat, jestli se nemá překládat nebo jestli to je neoficiální název něčeho, co má pevný překlad a podobně. Výstupem lexikálního analyzátoru je tedy rozdělení slov. Tento výstup je logicky rovnou předáván na vstup dalšího „stroje" – syntaktického analyzátoru. Ten, jak již bylo zmíněno v úvodu, dělá jakýsi větný rozbor – určuje, co je podmět, přísudek, předmět a všechny tyto údaje uloží do struktury údajů, se kterou pak pracuje v překladovém stroji.[7]

Lexikální analýza

Ilustrace 2: Grafické znázornění lexikální analýzy

Celá analýza je v principu jednoduchá – prochází se znak po znaku vstupní soubor a jakmile se narazí na určitý znak nebo na sekvenci znaků, která je definována v souboru definic, tak se předá na výstup jako definovaný symbol. Ve zkratce se tedy z původního souboru znaků vytvoří jiný soubor, který se již neskládá ze znaků, ale ze značek – tokenů[2]. Tyto tokeny jsou pak zpracovávány syntaktickým analyzátorem.

Na začátku jsem přirovnal lexikální analýzu ke slovnímu rozboru – určení slovních druhů. Je to ovšem tak snadné? Je tak snadné určit slovní druhy? Bohužel ne. Představte si to v češtině – lexikální analyzátor by musel znát veškerá česká slova, všechny jejich tvary a podobně. Nehledě na to, že některé tvary slov můžou být shodné pro více tvarů, jeden tvar může být třeba podstatné jméno a sloveso zároveň a podobně. Takže takhle by to pro češtinu adaptovat nešlo a bylo by potřeba zvolit sofistikovanější metodu rozboru.

Proč to tedy používám v Toki Poně, když to je také přirozený jazyk jako čeština? Jelikož Toki Pona má oproti češtině řadu vlastností, které mi usnadňují práci, jelikož na ně lze díky ním tuto relativně primitivní analýzu provést:

- v Toki Poně jsou známá všechna oficiální slova
- nemění se slovní tvary
- u každého slova jsou známy jeho slovní druhy – ovšem vzhledem k tomu, že každé slovo jich má víc, tak nastávají obtíže – viz dále

Díky tomu lze všechna slova uvést v souboru definic lexikálního analyzátoru. Jak jsem ale již zmínil, tak je celkem problém s tím, že v podstatě každé slovo může být použito jako víc slovních druhů než jeden. Svým způsobem to je ryze technický problém a řešení existuje, přestože je pracné – a to používáním skupin tokenů nad slovními druhy.

Abych vysvětlil poslední větu předchozího odstavce, tak stručně vysvětlím, jak funguje samotný lex. V seznamu definic jsem si nadefinoval skupiny slov, které zastávají slovní druhy. Definice je

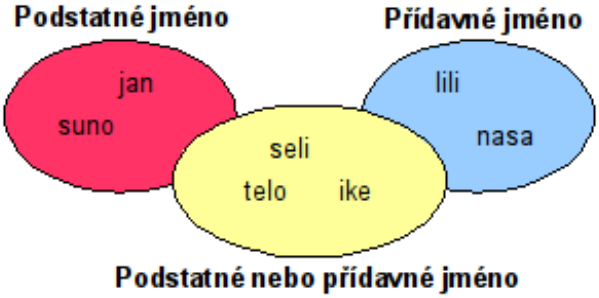
2 Jedná se tedy v principu o sekvenci symbolů.

poměrně snadná – vytvořím regulérní výraz, který se ztotožní se všemi slovy tohoto slovního druhu. Používám k tomu primitivní regulérní výraz: `(první slovo|druhé slovo|třetí slovo)`. Pro snadnější pochopení uvedu ukázku z vývojové verze:

```
podstatne_jmeno (jan|suno|telo)
pridavne_jmeno (ike|lili|nasa|seli)
```

Jenže zde nastává problém – slovo *telo* (které je v kategorii podstatných jmen) může být i přídavné jméno a *seli* může být i podstatné jméno, to samé *ike*. Jak to vyřešit? Spojenou kategorií:

```
podstatne_jmeno (jan|suno)
pridavne_jmeno (lili|nasa)
podstatne_pridavne_jmeno (telo|seli|ike)
```



Ilustrace 3: Grafické znázornění spojených kategorií

Toto řešení může někomu připadat jako nesmyslné; mohli by namítnout, že dané slovo by mohlo být ve více kategoriích bez použití spojené kategorie. To je pravda, bohužel by to bylo k ničemu. Slovo byl bylo označeno jako pouze jeden druh (jeden token) – a to (v případě lexu) ten dříve definovaný. Tudíž mě nenapadá lepší řešení než toto.

Problém s tímto řešením je následná složitost syntaktické analýzy – jelikož pak není třeba pro podstatné jméno jedna kategorie, ale je jich několik. (Pokud někde má být podstatné jméno, tak tam nelze uvést jen slovní druh `podstatne_jmeno`, ale je potřeba uvést ještě `podstatne_pridavne_jmeno` a případné další.)

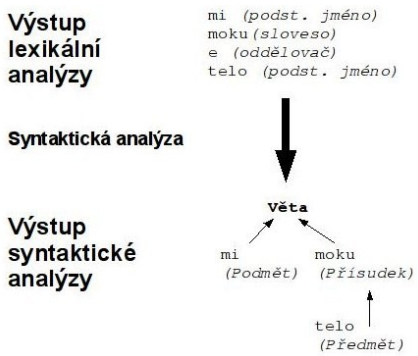
Lex

Program lex je velmi starý – byl publikován již v roce 1975. Jde o program, který definici lexikálních pravidel převede na zdrojový kód jazyka C, který se následně zkompiluje a chová jako lexikální analyzátor. Vzhledem k tomu, že přímo převádí soubor definic na zdrojový kód, tak se do souboru definic vkládají přímo kusy kódu v jazyce C, které se následně vykonají.

Lex používá na rozpoznávání sekvencí znaků regulérních výrazů. Těmto výrazům se nebudu příliš věnovat, jako jedinou důležitou věc zmíním to, že se jedná v podstatě o univerzální nástroj na vyhledávání textových řetězců. Proto není divu, že je použit i v tomto případě.

Princip fungování lexu je jednoduchý – prochází soubor znak po znaku a při shodě znaku nebo sekvence znaků z definice se vykoná akce daná v souboru definic – která nejčastěji zkopíruje tuto sekvenci pro další použití (je-li to nutné) a přiřadí mu značku (token). Tento token je vlastně to jediné, co následně zajímá syntaktický analyzátor – ta sekvence znaků (ono slovo) následně zajímá mne, jelikož ji používám na překlad a ukládám do větné struktury.

Syntaktická analýza



Ilustrace 4: Grafické znázornění syntaktické analýzy

Syntaktická analýza je jedna z nejdůležitějších částí z celého překladače, přestože já dělám pouze soubor definic – gramatiku. Jedná se o proces, který ve své podstatě určí, jaké slovo je který větný člen. Oproti lexikální analýze mě tato analýza přijde mnohem složitější – přestože jsou v praxi použitelné pouze dohromady.

V předchozím odstavci jsem zmínil jedno opravdu důležité slovo, a tím je gramatika. Co to vlastně je? Existuje třeba gramatika českého jazyka – to ví každý. Když někdo píše gramaticky správně, tak to znamená, že vlastně píše česky. Co to ale znamená pro syntaktickou analýzu?

Gramatika je soubor pravidel, kterým se definuje jazyk. Gramatiku jsi lze představit jako krabici. Když budeme s krabicí třepat, tak z ní začnou padat věty – a postupně by z ní vypadli všechny věty, které patří do jazyka.[8] Jinými slovy gramatikou lze určit všechny věty, které existují. To zní moc hezky – jenže pomůže nám to vůbec? Kupodivu ano – existuje totiž podle mne naprosto geniální metoda, která gramatiku (návod na tvoření vět) otočí tak, že lze určit, které slovo je jaký větný druh; jedná se vlastně o reverzní proces, při kterém se otočí proces vytvoření té samotné věty.

Definice gramatiky

Nejdříve uvedu příklad zápisu, který by se hodil na matematické příklady, kde se pouze sčítá a násobí (yacc používá pro popis variantu formátu zvaného BNF[3]):

```
(p1) V → V + V
(p2) V → V * V
(p3) V → id
```

Nyní jsem si nadefinoval tři pravidla (p1, p2 a p3). Názvy, co se vyskytují na levé straně (jako V – Výraz) jsou takzvané nonterminály. Ty, co se naopak vyskytují pouze na pravé straně (jako id – identifikátor), jsou takzvané terminály a jsou to vlastně tokeny, které vrací lex. Pro vysvětlení pojmů terminál a nonterminál – je-li výsledek tvořen pouze terminály, pak jsme již skončili s jeho tvořením; existuje-li v něm však ještě nějaký nonterminál, pak je potřeba nahradit tento nonterminál výrazem, ve kterém jsou pouze terminály, protože jinak není tvoření výsledku (v tomto případě matematického příkladu) dokončeno.

Vraťme se nyní k příkladu. Celé to tedy znamená, že výraz (V) může být tvořen součtem dvou výrazů, součinem dvou výrazů nebo nějakým identifikátorem (id), což může být třeba číslo nebo proměnná. Teď ukážu, jak lze pomocí těchto pravidel vytvořit výraz x + y * z:

```
V → V * V (p2)
  → V * z (p3)
  → V + V * z (p1)
  → V + y * z (p3)
  → x + y * z (p3)
```

(v závorkách jsou vypsáná pravidla, podle kterých jsem postupoval)

3 Formát BNF (Backus Naur Form) byl představen pány Johnem Backusem a Peterem Naurem pro popis jazyka ALGOL60.[9]

Použití gramatiky v syntaktické analýze

Dostali jsme se k požadovanému výrazu. Postupovali jsme od původního startovacího nonterminálu (V) k výslednému tvaru, kde již žádný nonterminál není. Nyní k výše zmíněné geniální metodě, podle které jsem schopen zpětně přijít na to, kterými pravidly vznikl daný výraz. Já potřebuji původní proces obrátit – místo postupu od jednoho nonterminálu k terminálům se potřebuji dostat od terminálů k jednomu nonterminálu. Tato metoda je známá jako *odzdola-nahoru* či *přeskoč-redukuj*[4] a používá zásobník pro skladování dat. Nyní ukázka výše zmíněného procesu tvoření, ale v opačném pořadí:

```
 1   . x + y * z    přeskoč
 2   x . + y * z    redukuj (p3)
 3   V . + y * z    přeskoč
 4   V + . y * z    přeskoč
 5   V + y . * z    redukuj (p3)
 6   V + V . * z    přeskoč
 7   V + V * . z    přeskoč
 8   V + V * z .    redukuj (p3)
 9   V + V * V .    redukuj (p2)  (násobení)
10   V + V .        redukuj (p1)  (sčítání)
11   V .            přijmout
```

Názvy vlevo od tečky jsou na zásobníku, názvy vpravo od ní jsou na vstupu. Začneme přidávání názvů ze vstupu do zásobníku (přeskakováním). Pokud se na zásobníku objeví něco, co je na pravé straně v gramatice, tak se to nahradí odpovídajícím nonterminálem na straně levé. Nakonec nastane stav, kdy bude na zásobníku pouze původní nonterminál (v případě, že vstup byl vytvořen podle této gramatiky). Pokud by někdo zkoumal předchozí proces blíže, tak by jistě našel nesrovnalosti a více možností, co dělat (třeba v kroku 6) – to je tím, že gramatika není jednoznačná (je více způsobů, jak je možné postupovat k vytvoření téhož).

Doufám, že toto na pochopení principu syntaktické analýzy stačí. Pro mě je vlastně důležitá z toho důvodu, že díky ní jsem schopen efektivně uložit větu v Toki Poně do své vnitřní struktury a věnovat se pak již přímo překladu.

```
veta:
        podmet li prisudek { printf ("podmet: %s (li) \n", $1); }
        | podmet_bez_li prisudek { printf ("podmet: %s \n", $1); }
        ;

predmet_single:
        podmet { printf("predmet: %s\n", $1); }
        | podmet_bez_li  { printf("predmet: %s\n", $1); }
        ;

predmet:
        predmet_single
        | predmet e predmet_single
        ;

pridavne:
        pridavne_jmeno { printf ("pridavne jmeno: %s \n", $1);}
        | mute { printf ("pridavne jmeno: mute \n"); }
        | ni { printf ("pridavne jmeno: ni \n"); }
        | pridavne_jmeno pridavne { printf ("pridavne jmeno: %s \n", $1);}
        ;

podmet:
        podstatne_jmeno
        | podstatne_jmeno pridavne
        ;
```

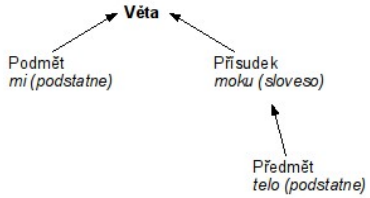
Ilustrace 5: Ukázka zdrojového kódu syntaktického analyzátoru

4 *Bottom-up* a *shift-reduce*

Překladový stroj

Překladový stroj je poslední a zároveň dle mého názoru nejdůležitější částí celého překládacího mechanismu. Já jsem sám navrhl jeden takový stroj, který jsem použil právě pro překlad Toki Pony.

Před popisem vlastního fungování shrnu poznatky, které si myslím, že jsou důležité pro vlastní pochopení funkčnosti. První z nich je vstup – vstupem je v tomto případě věta v interní struktuře, kterou dostanu od syntaktického analyzátoru. Tato struktura je nositelem všech informací o větě samotné; jsou zde mimo větné struktury (větný rozbor; vztahy mezi slovy) i informace o jednotlivých slovech (slovní druh, osoba podmětu a podobně). Překladový stroj se tedy nestará o to, jak se vytvoří tato struktura věty, což z něj teoreticky dělá nástroj použitelný na překlad i ostatních jazyků, nejenom Toki Pony (při použití jiného syntaktického analyzátoru). Pro jednoduchost uvádím příklad věty ve vnitřní struktuře:



Ilustrace 6: Struktura věty pro překladový stroj

Ještě pro další upřesnění – **věta** je takzvaný kořen stromu (nejsvrchnější element), Podmět a Přísudek jsou větve a Předmět je list (nejspodnější element). Pro potřeby samotného překladu si každé slovo pamatuje, jestli již bylo přeloženo, nebo ne.

Dalším poznatkem je výstup – výstupem je (celkem logicky) gramaticky správná a významově podobná věta ve finálním jazyce. Významovou podobností jsem se zabýval již v úvodu, ale ještě ji doplním přímo v souvislosti s Toki Ponou. Vzhledem k tomu, jak mnohovýznamová Toki Pona již ze své podstaty je, tak nelze v podstatě přímo určit přesný překlad, jelikož žádný takový neexistuje. Z toho důvodu chápu pro sebe samotný překlad jako čistě technickou věc, nikoliv věc nějak výrazně praktickou; jde mi o použitelný překlad.

Při navrhování tohoto překladového stroje jsem se inspiroval provedením programů lex a yacc. Zalíbila se mi myšlenka exportování vstupního souboru do podoby spustitelného programu. Má to obrovskou výhodu v tom, že pak v samotné definici vstupního souboru není potřeba implementovat vlastní logiku operací – jen stačí dát možnost vložit vlastní kód. K této možnosti přispěl i fakt, že celý program je psán v jazyce C. Proto jsem se rozhodl napsat tento stroj v duchu podobné myšlenky, i když zdaleka ne tak sofistikovaný.

Posledním faktem, který bych chtěl zmínit, je samotné propojení struktury překladače s Toki Ponou. To se projevuje extra částí na překlad slov s přídavnými jmény. Je to dáno tím, že v Toki Poně je tento způsob vyjadřování velmi důležitý, jelikož lze tímto způsobem vyjádřit největší množství popisných informací. Z toho důvodu jsem přidal tuto část, protože se mi tím naskytla relativně jednoduchá možnost, jak zvětšit slovní zásobu tohoto stroje. Tato část má i tu výhodu, že lze využít i pro překlad jiných typů vazeb, než mezi podstatnými a přídavnými jmény – a to třeba pro překlad, kde je závislost mezi přísudkem a předmětem (*mi moku e telo* – já piji vodu).

Princip fungování

Vlastní překlad by se dal rozdělit na tři části. V první fázi se překládají „větve“. Toto jsem zmínil v odstavci o propojení překladače s Toki Ponou – překládají se zde celky složené z více než jednoho slova. Další fáze je již celkem primitivní – v této fázi se projde postupně celý strom a přeloží se každé slovo samo o sobě, bez jakékoliv závislosti. V poslední, třetí fázi, se nechá tento strom spojit dohromady ve výslednou větu.

Před popisem konkrétních metod překladu bych rád upozornil na jeden fakt – a to na druh slov. Každé slovo má uloženo jak svůj slovní druh, tak svou roli ve větě (větný člen). Tím pádem lze překládat podle obou těchto kategorií a „druh slova“ v následujícím textu může tedy znamenat jak větný člen, tak slovní druh.

Začneme první částí – překladem „větví“. Předpokládejme, že v souboru definic je uložená nějaká část věty, která se má jako celek přeložit – třeba přísudek s předmětem, nebo podstatné jméno s přídavným. Formát vypadá asi takto:

```
jan utala = soldier;
moku telo [prisudek>predmet] = drink; water;
```

V prvním případě se jedná o překlad podstatného jména s navazujícím přídavným jménem. Není zde uveden druh slov (v hranatých závorkách), proto se předpokládá, že první je podstatné jméno a další slova že jsou přídavná jména (viz odstavec o propojení Toki Pony s překladačem). V druhém případě se jedná o vztah mezi přísudkem (slovesem) a předmětem (podstatným jménem) a je zde proto uveden druh slov v hranatých závorkách. Mezi těmito druhy je znak menší než, a to z toho důvodu, že tam je podřadný vztah[5]. Pokud by to byl vztah souřadný (třeba mezi dvěma či více přídavnými jmény), tak by tam byla mezera. Následuje znak rovnítka a za ním překlad ukončený středníkem. Proč je tam různý počet středníků? Důvod je jednoduchý – jedná se o to, která slova budou překládána a která ne. V první případě se přeloží pouze podstatné jméno (*jan*) za slovo soldier a slovo *utala* se smaže. Naopak v druhém případě se nahradí slovo *moku* za drink a slovo *telo* za water a nic se tedy nemaže. Pokud bych chtěl udělat nějaké další akce při překladu, tak mohu před středník vložit mezi sekvence znaků %{ a %} kód v jazyce C, který se přímo při překládání vykoná.

Teď bych se rád dostal k druhé fázi, která překládá samotná slova. Tato fáze je vlastně zjednodušení fáze první – nepřekládá se již více slov, ale vždy jen jedno naráz. Syntaxe je velmi podobná:

```
telo [podstatne] = water; [pridavne] = liquid;
```

První rozdíl je v tom, že na začátku se určí pouze jedno slovo, které se překládá, místo několika. Druhým rozdílem je fakt, že zde může následovat více slovních druhů bez opakování původního slova; což šetří místo a taky vzhledem k faktu, že u některých slov by měl být překlad třeba tří nebo čtyř druhů, tak to činí soubor definic výrazně přehlednější.

Nyní je potřeba zdůraznit, v jakém pořadí se překládá. V první případě se postupně prohledávají všechny možné fráze, jestli se vyskytují, a to v tom pořadí, v jakém jsou uvedeny v souboru definic. Je to celkem logické – obráceně to dělat nejde (hledat všechny možné části stromu v souboru definic), a to z toho důvodu, že by bylo potřeba vyzkoušet všechny možnosti, což by bylo dost pracné a dle mého názoru nelogické. Naopak v druhé fázi stačí již projít postupně celý strom

5 Podřadný vztah je v tomto případě vztah mezi rodičem a dítětem, tedy třeba mezi kořenem a větví. Souřadný vztah by byl naopak vztah mezi dvěma slovy se stejným rodičem, třeba mezi podmětem a přísudkem (mají jednoho rodiče – větu).

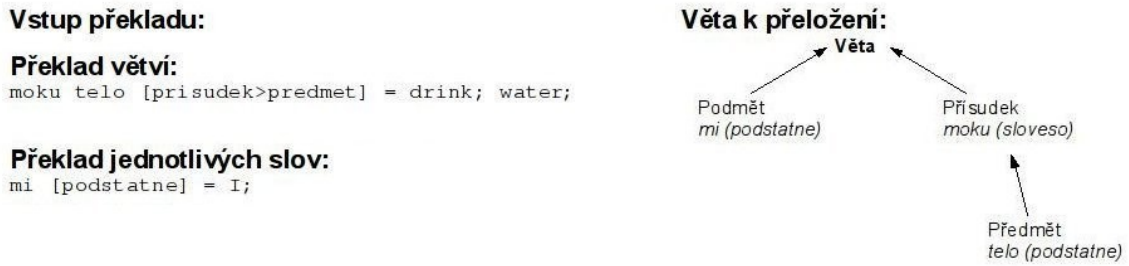
(protože jakýkoliv závislý překlad by se měl provést již v první fázi) a postupně nechat přeložit všechna slova; zde se naopak prochází strom a vyhledává se v seznamu definic. Pro jistotu však nechávám strom překládat zespodu (od listů ke kořenu), jelikož to není technický problém (je stejně složité nechat to překládat od kořenu k listům jako od listů ke kořenům) a protože pokud by se zde náhodou vyskytoval nějaký závislý překlad, tak bude spíše záviset na podřadném elementu než na elementu nadřazeném.

V poslední fázi se vytvoří samotná přeložená věta. K tomu slouží funkce v jazyce C, která se uvede v souboru definic a která spojí všechny přeložená slova dohromady. Vzhledem k tomu, že se tento kód i s funkcí na překlad věty vkládá na začátek výsledného zdrojového kódu, tak je možné zde uvést také další případné pomocné funkce, které jsou potřeba pro dílčí překlady.

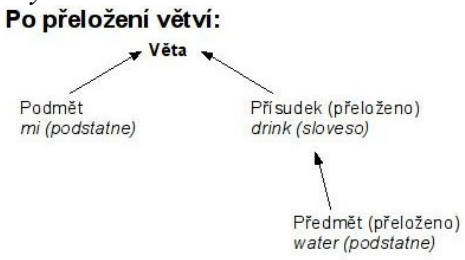
Nyní je jasné, že soubor definic je tvořen třemi různými sekcemi; v první se definují překlady větví, v druhé překlady slov a ve třetí je již jen zdrojový kód s funkcí na vytvoření přeložené věty. Tyto tři části jsou od sebe odděleny sekvencí znaků %%.

Překladový stroj je sám o sobě tou nejdůležitější částí překladu. Mě se zde snad povedlo popsat základní fungování tohoto stroje a také to, jakým způsobem funguje uvnitř. Neříkám, že se jedná o naprosto univerzální nástroj, se kterým by se dalo hned jít na překlad takového jazyka, jakým je třeba čeština či angličtina. Jedná se dle mého názoru o jednoduché, ale účinné řešení; plně vyhovuje mé situaci a je rozšiřitelné právě vkládáním kusů kódu.

Ukázka fungování



Ilustrace 7: Vstup překladového stroje – nalevo vstup ze souboru definic, napravo ze syntaktického analyzátoru



Ilustrace 8: Po první fázi překladu – po přeložení větví

Ilustrace 9: Po druhé fázi překladu – po překladu slov

Ilustrace 10: Po třetí fázi – po spojení ve výslednou větu

Testování, dosažené výsledky, závěr

Jak jsem testoval celý program? Relativně jednoduše. Vytvořil jsem si nějaké předdefinované fráze, jejich překlad, a vždy s novou verzí jsem kontroloval, jestli se překládají tak, jak se mají přeložit. Jiné testování jsem neprováděl – časová a paměťová náročnost není u takto malého programu důležitá, jelikož překlad se provede v podstatě okamžitě a slovník je relativně malý.

Přesněji jsem testoval takto – v jedné složce mám soubory, kde u každého souboru je jeho název věta k přeložení a obsah tohoto souboru je výstup programu na překlad (výsledek překladu), který jsem vytvořil. Skript pro testování pak projde všechny soubory v tomto adresáři, nechá překladový stroj přeložit název každého souboru a zkontroluje výsledek s obsahem souboru. Zde je ukázka výstupu testovacího skriptu:

```
tomas@tomas-laptop:~/rocnikovka$ ./tests
Test 'jan li moku' passed
Test 'jan li suli' passed
Test 'mi moku' passed
Test 'mi moku e kili' passed
Test 'mi moku e kili e telo' passed
Test 'mi moku e kili li pakala e sina' passed
Test 'mi moku li pakala' passed
Test 'mi pona e ijo' passed
Test 'mi suli' passed
Test 'mi wile e sina' passed
Test 'mi wile lukin e ma' passed
Test 'mi wile lukin e ma e suno' passed
Test 'mi wile pakala e sina' passed
Test 'mi wile sina' passed
Test 'ona li lukin e pipi' passed
Test 'ona li pona e ilo' passed
Test 'ona li wile jo e ilo' passed
Test 'pipi li lukin li unpa' passed
Test 'suno li suli' passed
Passed: 19; Errors: 0; Total 19
```

Ilustrace 11: Výstup testovacího skriptu

Čeho se mi povedlo dosáhnout? Povedlo se mi sepsat, úspěšně spustit a otestovat na vzorku některých vět z učebnice jazyka správnost překladu. Správnost překladu jsem kontroloval osobně; vždy jsem dával pozor na to, aby výsledná věta dávala smysl (ne aby nutně pokryla dokonale význam). Nelze předpokládat, že překlad bude dokonalý; a to z podstaty samotné Toki Pony.

Dále by se slušelo uvést, čím by se dalo pokračovat. Podle mého názoru by se bez celkové výměny některé z částí překladače dalo pokračovat na následujících věcech: na ještě detailnějším a podrobnějším slovníku (aby se při překladu slova *telo* neukázal jako překlad jenom voda, ale třeba i jezero, moře, oceán a podobně), na překladu více vět naráz a nestandardních vět (a obecně na větším využití interpunkce, která ale není pro Toki Ponu příliš vlastní, přestože se v ní používá) a případně na překladu různých sporných částí Toki Pony (třeba u vět s dvěma možnými výklady). Samotný překladač by se nejspíš dal vylepšit interně – a to asi nejvíce po stránce optimalizace kódu.

Na úplný závěr bych chtěl říct, jaký byl přínos celé práce pro mne. Já jsem především rád, že jsem se seznámil s programy lex a yacc – vzhledem k tomu, že se mi líbí práce s různými formáty souborů, tak jsem rád, že jsem nalezl nástroje, které jsou schopné je efektivně zpracovat. Leccos jsem se od nich přiučil – nejenom styl zpracování dat, ale i myšlenku přímého překladu definice do zdrojového kódu, který se sám vykoná. Dále jsem rád, že jsem trochu nahlédl do světa zpracování přirozeného jazyka – přestože ve vlastní práci nepoužívám žádnou ze sofistikovaných metod, tak bylo dobré si rozšířit obzory i tímto směrem. V neposlední řadě jsem si obnovil znalosti jazyka C.

Seznam použité literatury

1: Wikipedie, Překladač, http://cs.wikipedia.org/wiki/Překladač, Perex; verze 17. 1. 2009, 18:07
2: PhDr. Miroslav Řešetka a kol., Frazeologický & Idiomatický slovník, 1999, Fin Publishing, ISBN 80-86002-57-8
3: Wikipedie, Toki Pona, http://cs.wikipedia.org/wiki/Toki_pona, Perex; verze 15. 1. 2009 v 18:01
4: Wikipedie, Čeština, http://cs.wikipedia.org/wiki/Čeština, Slovní zásoba; verze 6. 2. 2009 v 08:54
5: Wikipedie, Bezkontextová gramatika, http://cs.wikipedia.org/wiki/Bezkontextová_gramatika, Perex; verze 15. 1. 2009 v 21:13
6: Sonja Elen Kisa – Richard, B. J. Knight, Toki Pona: The Language of Good – The Simple Way of Life, PDF verze http://rowa.giso.de/languages/toki-pona/english/index.php, nebo jako online lekce http://toki.co.nr/lesson/lesson0.html
7: Tom Niemann, A Compact Guide to LEX & YACC, e-book, http://epaperpress.com/lexandyacc/index.html
8: Ondřej Holeček, Jak se dělá překladač, http://www.root.cz/clanky/jak-se-dela-prekladac/
9: Wikipedie, Backus–Naur Form, http://en.wikipedia.org/wiki/Backus-Naur_form, Perex a Historie; verze 13. 1. 2009 v 17:15