UNIVERZITA KOMENSKÉHO, BRATISLAVA
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

SYNTAKTICKÁ ANALÝZA VNORENÝCH PROGRAMOVACÍCH JAZYKOV

DIPLOMOVÁ PRÁCA

2015

Bc. Tomáš Belan

UNIVERZITA KOMENSKÉHO, BRATISLAVA
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

SYNTAKTICKÁ ANALÝZA VNORENÝCH PROGRAMOVACÍCH JAZYKOV

DIPLOMOVÁ PRÁCA

Študijný program:	Informatika
Študijný odbor:	2508 informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	RNDr. Richard Ostertág, PhD.

Bratislava, 2015

Bc. Tomáš Belan

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Tomáš Belan
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Syntaktická analýza vnorených programovacích jazykov
Syntactic analysis of nested programming languages

Cieľ: Cieľom práce je:
1. Preskúmať existujúce metódy syntaktickej analýzy s ohľadom na použitie viacerých programovacích jazykov v jednom zdrojovom texte.
2. Navrhnúť nový parser, ktorý rozširuje klasické parsery pre jednotlivé vnorené jazyky o kooperáciu pri parsovaní zloženého zdrojového textu.
3. Implementovať prototyp navrhnutého parsera.
4. Ukázať príklady rozšírenia existujúcich jazykov o podporu ich kombinácie s inými jazykmi.
5. Demonštrovať použitie implementovaného parsera pre parsovanie ukážkových jazykov.

Vedúci: RNDr. Richard Ostertág, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 26.11.2013

Dátum schválenia: 26.11.2013 prof. RNDr. Branislav Rovan, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Ďakujem svojmu školiteľovi, RNDr. Richardovi Ostertágovi, PhD., za jeho pomoc a podporu.

Abstrakt

V tejto práci sa pozrieme na problém parsovania viacerých programovacích jazykov v jednom súbore. Navrhneme a implementujeme systém, ktorý umožňuje prepojiť viaceré parsery, a to aj ak sú postavené na rôznych teoretických základoch alebo používajú iné pravidlá lexikálnej analýzy. Jednotlivé parsery sa navzájom volajú na základe oddeľovačov, ktoré vo vstupnom texte určujú oblasti v rôznych jazykoch.

KĽÚČOVÉ SLOVÁ: syntaktická analýza, parsovanie, programovacie jazyky

Abstract

In this thesis we look at the problem of parsing multiple programming languages in a single file. We design and implement a system that allows connecting multiple parsers, even if they are based on different theoretical models or require varied lexical analysis rules. The parsers call each other based on separators that mark input areas written in different languages.

KEYWORDS: syntactical analysis, parsing, programming languages

Obsah

Úvod

Počet programovacích jazykov stále rastie. Moderný programátor obyčajne musí ovládať veľa rôznych jazykov – nielen vtedy, keď pracuje na viacerých rôznych projektoch, ale často aj v rámci jedného projektu. Rôzne programovacie jazyky majú rôzne výhody a nevýhody, a môžu byť určené na rôzne účely. Niektoré projekty majú jeden hlavný jazyk a viaceré pomocné jazyky špecializované na konkrétny účel (napríklad databázové dotazy v SQL, shell skripty na kompiláciu), iné používajú viaceré plnohodnotné jazyky (napríklad webová aplikácia, ktorá na strane servera používa PHP a na strane klienta JavaScript). Z týchto dôvodov dnes patrí použitie viacerých jazykov v jednom projekte k bežnej praxi.

Zdrojové texty napísané v rôznych jazykoch sú obvykle uložené oddelene v rôznych súboroch, ale neplatí to v každej situácii. Zvlášť v prípade jazykov špecializovaných na jeden účel (*domain-specific languages*), ako je napríklad SQL, sa malé fragmenty kódu často píšu dovnútra väčšieho celku napísaného v inom, všeobecnejšom jazyku. To platí aj o regulárnych výrazoch – špecializovanom jazyku umožňujúcom kompaktne zapísať rozpoznávanie zložitých vzorov. Ďalšie špecializované jazyky sa používajú na definície dátových štruktúr, textové šablóny, symbolické výpočty a tak ďalej. Niekedy je vhodné zdrojový kód v špecializovanom jazyku oddeliť do samostatného súboru, ale často by sme chceli, aby bol súčasťou väčšieho celku.

Historicky sa vyskytli viaceré metódy, ako tieto *vnorené programovacie jazyky* implementovať. Rôzne programovacie jazyky používajú za normálnych okolností rôzne parsery – ak chceme použiť viaceré v jednom súbore, musíme si vybrať, ako ich spojiť. Tu sú niektoré možné spôsoby:

Prvá možnosť je nijako parsery nespájať. Vnorený jazyk môžeme z pohľadu vonkajšieho jazyka zapísať ako reťazcovú konštantu. Keď parser načíta daný súbor, reťazcové konštanty nebude ďalej spracovávať. Napríklad v jazyku Python sa regulárne výrazy píšu ako reťazcové konštanty, ktoré prekladá funkcia `re.compile()`. Vnáranie SQL dotazov do iného jazyka obvykle tiež používa tento princíp. Nie je náročný na implementáciu, ale má viaceré nevýhody: Ak je vo vnorenom zdrojovom texte chyba, zistí sa to až za behu (pri zavolaní `re.compile()`, spustení SQL dotazu, a podobne). Neopatrná manipulácia s reťazcami môže viesť k *SQL injection* a ďalším bezpečnostným dieram. Navyše, druhoradé postavenie vnorených jazykov

komplikuje alebo znemožňuje akúkoľvek spoločnú statickú analýzu.

Druhá možnosť je využiť funkcie vonkajšieho jazyka a skonštruovať systém, ktorý síce vyzerá ako iný jazyk, ale dá sa prečítať tým istým parserom. Tento prístup je obľúbený napríklad v komunite jazyka Ruby, ktorého gramatika je veľmi flexibilná a používa len málo interpunkcie. Mnohé Ruby nástroje používajú minijazyky, ktoré majú vlastnú sémantiku, ale pritom technicky sú podmnožinou jazyka Ruby a číta ich ten istý parser. Ešte silnejším príkladom sú makrá v dialektoch jazyka Lisp. Výhodou tejto metódy je, že špecializovaný jazyk môže poskytovať bežná knižnica a do parsera netreba robiť žiaden zásah. Ale nedá sa spraviť nič, na čo pôvodná gramatika nebola pripravená. Nie je možné spojiť dva ľubovoľné jazyky.

Tretia možnosť je manuálne vytvoriť gramatiku, ktorá spája oba jazyky, a pre túto gramatiku vygenerovať parser. Napríklad v jazykoch JavaScript a Perl sú regulárne výrazy plnohodnotnou súčasťou jazyka – ak je v JavaScriptovom zdrojovom kóde nesprávny regulárny výraz, zistí sa to ešte pred jeho spustením. V jazyku C# sa namiesto SQL reťazcov môže použiť LINQ [12], ktorého syntax je zabudovaná priamo do gramatiky C#.

Táto metóda rieši problémy s reťazcovými konštantami, ale často si vyžaduje veľký zásah do pôvodnej gramatiky. Pridať do existujúceho jazyka, ako napríklad Python, nové rozšírenie, ako napríklad LINQ, môže byť netriviálny problém. Vnorená gramatika nemusí byť kompatibilná s typom parseru, čo používa vonkajší jazyk. Aj ak oba jazyky používajú ten istý typ parserov, pri ich spojení môžu vzniknúť nové konflikty. Okrem toho medzi jazykmi môže nastať ešte zásadnejší konflikt, napríklad v zápise komentárov, význame bielych znakov a iných pravidlách lexikálnej analýzy.

Štvrtá možnosť je použiť parser, v ktorom je možné odvolať sa na ďalšie parsery. Z formálneho hľadiska budú jednotlivé vnorené jazyky samostatné – každý bude mať vlastnú gramatiku a vlastný lexikálny analyzátor, ale keď pri parsovaní na vstupe príde fragment kódu napísaný v inom jazyku, kontrolu prevezme jeho parser. Tento spôsob je zvlášť praktický, ak všetky jednotlivé parsery používajú rekurzívny zostup (*recursive descent parsers*). Ak sa každý parser skladá z niekoľkých vzájomne rekurzívnych funkcií, ľahko sa dajú upraviť, aby funkcie pre jeden jazyk volali druhý. Keby sme mali rekurzívny parser pre Python a rekurzívny parser pre SQL, do funkcie *parsuj Pythonový výraz* by sme mohli pridať konštrukciu: „Ak vidíš SQL kľúčové slovo ako `SELECT`, `INSERT` a tak ďalej, zavolaj funkciu *parsuj SQL program*“.

Spojiť takto viaceré ručne písané rekurzívne parsery nie je náročné a má to mnohé výhody. Na rozdiel od použitia reťazcových konštánt je výsledkom celý syntaktický strom, takže syntaktické chyby dokážeme nájsť bez spustenia programu. Na rozdiel od druhej a tretej metódy z nášho zoznamu môžeme spojiť dva takmer ľubovoľné jazyky – a to (s trochou opatrnosti) aj ak majú rôzne pravidlá lexikálnej analýzy. Aby v zloženom jazyku nedochádzalo

k nejednoznačnostiam, môžeme pridať oddeľovače, podľa ktorých parser aj programátor zistí, kde začína a končí ktorý jazyk.

Ale nie každý jazyk používa ručne napísaný parser. Existujú mnohé generátory parserov, ako napríklad Bison [4] a ANTLR [13], ktoré dokážu vygenerovať celý parsovací program iba na základe vstupnej gramatiky. Autorovi jazyka v tom prípade stačí, aby napísal gramatiku, a generátor pre ňu vyrobí vhodný parser. Pre autora jazyka je to podstatne pohodlnejšie, ale problém je v tom, že dnešné implementácie generátorov nepočítajú s možnosťou vnorených programovacích jazykov. Vyrábajú parsery, ktoré chcú za každých okolností spracovať celý zdrojový text.

Cieľom tejto práce je preto navrhnúť a implementovať nový generátor parserov, v ktorom bude podpora pre vnorené jazyky od základov vstavaná. Tento generátor musí podporovať generovanie všetkých štandardných typov parserov, aby bolo možné bez veľkých zásahov použiť existujúce gramatiky. Tiež musí byť konfigurovateľný, aby bolo triviálne pridávať ďalšie vnorené jazyky a nastavovať povolené kombinácie.

Implementovali sme systém *Multiparser*, ktorý spĺňa tieto požiadavky. Dokáže prepojiť nielen parsery založené na rekurzívnom zostupe, ale aj štandardné parsery založené na deterministických zásobníkových automatoch. Vygenerovaný zložený parser dokáže prečítať zdrojový text a vytvoriť kompletný syntaktický strom, v ktorom budú reprezentované všetky jazyky, z ktorých sa zdrojový text skladal.

Ale podotknime, že Multiparser rieši iba syntaktickú analýzu a nepredpisuje, ako výsledný syntaktický strom ďalej spracovať. Aby sa dal prečítaný program spustiť, môže byť nutné jednotlivé jazyky ďalej prekladať – napríklad preložiť všetky časti vstupného súboru do jednotného bytekódu. Ako príklad by sme si mohli predstaviť kompilátor, ktorý pomocou Multiparsera prečíta zdrojový text kombinujúci jazyky Java a Scala. Oba jazyky sa potom preložia do bytekódu pre Java Virtual Machine. Úlohou Multiparsera je iba prečítať vstupný zdrojový text a zostrojiť syntaktický strom, ktorý ho reprezentuje.

Zvyšok práce je usporiadaný nasledovne:

V kapitole 1 popíšeme základnú architektúru Multiparsera, predstavíme podporované typy parserov a ukážeme, ako ich Multiparser rozširuje o podporu vnorených jazykov. Kapitola predpokladá, že čitateľ je zoznámený s bezkontextovými gramatikami a konečnými a zásobníkovými automatmi, ale nevyžaduje predchádzajúce znalosti LL a LR parserov.

Kapitola 2 rozoberá konkrétnu implementáciu systému Multiparser – generátory parserov a kód, ktorý generujú, aplikačné rozhrania, a formáty zápisu gramatík a konfigurácií.

Kapitola 3 popisuje ukážkové jazyky priložené ku generátoru parserov. Praktické programovacie jazyky málokedy zapadajú do formálnych teoretických modelov – v tejto kapitole na niekoľkých jazykoch demonštrujeme, ako Multiparser rieši niektoré problémy z praxe.

Kapitola 1

Návrh systému

Štandardné parsery založené na automatoch a parsovacích tabuľkách, ako napríklad LL a LR parsery, nezvyknú mať zabudovanú podporu pre vnáranie iného jazyka. Problém nevzniká iba kvôli nejednoznačným gramatikám: Syntax programovacích jazykov v praxi nie je tvorená len gramatikou, ale aj pravidlami lexikálnej analýzy a ďalšími aspektmi, ktoré sa medzi jazykmi rôznia. (Rôzne jazyky napríklad môžu mať rôznu prioritu operátorov, ktorú niektoré parsery nepokladajú za súčasť gramatiky, ale za samostatné nastavenie.)

Náš systém, Multiparser, preto používa iný prístup. Miesto toho, že by sa snažil viaceré jazyky parsovať jediným spoločným LL alebo LR automatom, akceptuje, že rôzne jazyky majú rôzne parsery a tokenizéry.

To pochopiteľne vyžaduje určitý kompromis. V prípade Multiparsera je nasledovný: keď vnárame jeden jazyk do druhého, musia mať tieto jazyky vždy oddeľovače, ktoré presne určia, kde začína a končí text napísaný vo vnorenom jazyku. Tieto oddeľovače musia byť „jednoznačné“ – rozoznateľné bez toho, že by parser čítal ďalej. (Neskôr tento koncept definujeme presnejšie.) Vďaka oddeľovačom môže Multiparser obísť riziko nejednoznačných gramatík. A v praxi je to našťastie prirodzená požiadavka – aj cieľový používateľ (programátor) chce rýchlo vedieť zistiť, ktorá časť súboru je napísaná v ktorom jazyku.

Pomocou týchto oddeľovačov môže Multiparser kombinovať viaceré parsery a tokenizéry. Keď uvidí nový otvárací oddeľovač, spustí podprogram pre parsovanie daného vnoreného jazyka. Tento podprogram neskončí na konci vstupu, ako obvykle, ale už keď uvidí zodpovedajúci zatvárací oddeľovač. Celkový výsledok parsovania vnoreného jazyka sa z pohľadu vonkajšieho jazyka javí ako jeden celok, napr. jeden token. Takže upraviť existujúcu gramatiku, aby sme umožnili vnáranie iných jazykov, si nevyžaduje veľký zásah. Navyše, keďže rôzne jazyky sú spracované rôznymi podprogramami, nemusia ani používať tie isté typy automatov. Pre Multiparser nie je žiaden problém, ak vonkajší jazyk používa LL parser a vnútorný jazyk používa LR parser, alebo naopak.

1.1 Všeobecná architektúra

Keďže predpokladáme existenciu vhodných oddeľovačov, mohli by sme sa pokúšať začať parsovanie celého súboru tým, že najprv nájdeme na vstupe všetky reťazce zodpovedajúce oddeľovačom, rozdelíme podľa nich vstupný text, a vo vhodnom poradí použijeme parsery na jednotlivé časti. Ale tento prístup má veľké slabiny. Väčšinou napríklad nechceme, aby jazykové oddeľovače fungovali vnútri reťazcov alebo komentárov. Ale to, ako sa píše reťazec a komentár, závisí od konkrétneho jazyka.

Nestačí teda spraviť nejaké globálne predspracovanie a potom použiť pôvodné parsery pre jednotlivé jazyky. Spracovanie otváracích a zatváracích oddeľovačov sa musí diať za behu. Každý parser je nutné rozšíriť, aby do neho bolo možné vnoriť iné jazyky a aby jeho samotného bolo možné vnoriť do iných jazykov.

Analýza celého vstupného súboru preto bude mať hierarchický charakter. Celkový parser sa bude skladať z viacerých vzájomne rekurzívnych funkcii, ktoré vedia parsovať jednotlivé jazyky. Všetky budú zdieľať jeden spoločný vstupný buffer. Keď práve bežiaca funkcia uvidí na aktuálnej pozícii otvárací oddeľovač, rekurzívne zavolá inú funkciu, ktorá zanalyzuje text súboru až po zodpovedajúci zatvárací oddeľovač, posunie aktuálnu pozíciu za neho a vráti prečítané dáta.

Multiparser podporuje tri základné typy parserov: LL parsery, LR parsery a PEG parsery. V prípade LL a LR parserov sa parsovanie ďalej delí na lexikálnu analýzu (tokenizáciu) a syntaktickú analýzu. Lexikálna analýza má za úlohu preložiť vstupný prúd znakov na zmysluplné symboly – „tokeny“, zatiaľčo syntaktická analýza z nich má zostrojiť syntaktický strom. PEG parsery nepoužívajú takéto rozdelenie.

V ďalších sekciách sa podrobnejšie pozrieme na každý z týchto typov parserov a ukážeme si, ako ich treba upraviť, aby mohli obsahovať vnorené jazyky a byť vnárané do iných jazykov.

1.2 LL parsery

LL parsery [14] sú jedným zo štandardných typov parserov pre bezkontextové gramatiky. Čítajú vstup zľava doprava a hľadajú ľavé krajné odvodenie. Parsujú zhora nadol, čiže budujú strom odvodenia od koreňa, v ktorom je počiatočný neterminál. Strom počas čítania vstupu priebežne narastá: najľavejší neterminál, ktorý ešte nebol prepísaný (je v liste), sa podľa nejakého pravidla gramatiky prepíše a vzniknú pod ním nové listy.

Strom odvodenia preto počas parsovania tiež vzniká „zľava doprava“. Ľavá časť stromu už je „prečítaná“ a vo všetkých listoch sú terminály (alebo epsilón), ktoré presne zodpovedajú prečítanej časti vstupu. Za ňou ide „neprečítaná“ časť, kde si ešte nie sme istí odvodením, a v listoch môžu byť neterminály.

Konceptuálne sa LL parser vždy pozerá na prvý „neprečítaný“ list stromu. Ak je v ňom terminál, strom netreba nijako meniť – parser iba musí skontrolovať, že na vstupe naozaj nasleduje tento terminál, a zaradiť ho do „prečítanej“ časti. Ak je v ňom neterminál, tak je to najľavejší neterminál z aktuálnej vetnej formy, a preto ho treba preložiť. Parser si musí vybrať, ktorým pravidlom ho preloží. Pritom má dovolené pozrieť sa na k terminálov dopredu. Ak sa podľa nich vždy dá jednoznačne rozhodnúť, aké pravidlo treba v ľavom krajnom odvodení použiť, gramatiku voláme „LL(k) gramatika“ a parser je „LL(k) parser“.

Implementácie ale nepracujú priamo so stromom odvodenia, ktorý by mal „prečítanú“ a „neprečítanú“ časť. Parser si miesto toho pamätá pozíciu na vstupe, kde sa nachádza, a zásobník udávajúci, aké neterminály a terminály by mali byť na zvyšku vstupu. Obsah zásobníka vlastne zodpovedá listom v „neprečítanej“ časti stromu. Na začiatku je na zásobníku iba počiatočný neterminál σ a pod ním špeciálny terminál označujúci koniec vstupu (budeme ho označovať „$“).

Automat postupne číta vstup a podľa symbolu na vrchu zásobníka sa rozhoduje, čo ďalej. Ak je na vrchu zásobníka nejaký terminál t, znamená to, že podľa gramatiky musí práve on byť ďalším terminálom na vstupe. Keď tam naozaj je, všetko je v poriadku, automat ho vyhodí zo zásobníka (ako keby prejde do „prečítanej“ časti stromu) a pohne sa o jeden terminál ďalej. Keď tam je niečo iné, vstup je chybný a parsovanie zlyhalo. Automat úspešne končí vtedy, keď má prázdny zásobník.

Keď je na vrchu zásobníka neterminál ξ, automat sa musí rozhodnúť, na čo sa táto ξ vo vetnej forme prepíše. Na to slúži statická *parsovacia tabuľka* T, ktorá bola vopred skonštruovaná podľa vstupnej gramatiky. Pre každý neterminál ξ, čo sa snažíme prepisovať, a každú k-ticu terminálov, čo môže nasledovať na vstupe, sa v tabuľke dozvieme, ktorým pravidlom sa má ξ prepísať (alebo, že táto k-tica nezodpovedá žiadnemu korektnému odvodeniu). Ak sa má použiť pravidlo $\xi \to w$, automat z vrchu zásobníka odstráni ξ a pridá tam w^R. To zodpovedá tomu, že vo vytváranom strome odvodenia už ξ nie je listom, ale symboly z w sa stali listami.

V praxi sa najčastejšie používajú LL(1) parsery. Konštrukcia parsovacej tabuľky pre LL(1) parser prebieha nasledovne: Nech $FIRST(w)$ je množina všetkých terminálov, ktorými môže začínať vetná forma odvodená z w. Nech $FOLLOW(\xi)$ je množina všetkých terminálov, ktoré môžu v korektnom odvodení nasledovať po ξ. To jest:

$$FIRST(w) = \{\, x \in T \mid \exists u \in (N \cup T)^* : w \Rightarrow^* xu \,\}$$

$$FOLLOW(\xi) = \{\, x \in T \mid \exists u, v \in (N \cup T)^* : \sigma\$ \Rightarrow^* u\xi xv \,\}$$

Pre každé pravidlo $\xi \to w$ rekurzívne vypočítame hodnoty $FIRST(w)$, $FOLLOW(\xi)$ a zistíme, či je w vymazávajúce (t.j. $w \Rightarrow^* \varepsilon$). Potom tabuľku vyplníme nasledovne: hodnotou $T[\xi, x]$ bude pravidlo $\xi \to w$ práve vtedy, keď $x \in FIRST(w)$, alebo je w vymazávajúce a

$x \in FOLLOW(\xi)$. Ak by sme touto metódou dostali viacero rôznych hodnôt pre nejakú bunku tabuľky, našli sme konflikt a daná gramatika nie je LL(1).

1.2.1 Lexikálna analýza

Úlohou lexikálnej analýzy [11] je predspracovať vstupný text, aby mal jednoduchšiu formu a dal sa ľahšie analyzovať pomocou bezkontextových gramatík. V gramatike potom nemusia ako terminály vystupovať priamo jednotlivé znaky, ale tokeny, ktoré pôvodne mohli byť zapísané viacerými znakmi.

Lexikálna analýza sa v programovacích jazykoch často používa na riešenie úloh ako čítanie číselných a reťazcových literálov, odstraňovanie medzier a komentárov, a rozlišovanie mien premenných a funkcií od kľúčových slov ako `if`, `while` a podobne. LL(1) parser sa môže pozerať iba na jeden terminál dopredu, a podľa neho sa musí rozhodnúť, aké gramatické pravidlo použije. Keby terminály gramatiky neboli kľúčové slová, ale priamo jednotlivé znaky, parser by sa nevedel vždy rozhodnúť. Lexikálna analýza sa používa aj pri LR parseroch, ktoré si ukážeme neskôr.

Príčinou je to, že LL a LR parsery vo všeobecnosti nedokážu parsovať každú bezkontextovú gramatiku. Keby sme mali parser, čo to dokáže, lexikálnu analýzu by sme nepotrebovali. Bezkontextové gramatiky majú dostatočnú silu, aby vyjadrili číselné aj reťazcové literály programovacích jazykov, a dokážu rozlíšiť aj mená premenných od kľúčových slov: napríklad „jazyk všetkých slov z malých a veľkých písmen anglickej abecedy, okrem slov `if` a `while`“ nepochybne patrí do triedy bezkontextových jazykov (dokonca je regulárny). Ale aj keby sme ho mohli zapisovať v podobe bezkontextovej gramatiky, bolo by to určite veľmi nepohodlné.

Vo väčšine programovacích jazykov má lexikálna analýza iba jednoduché pravidlá. Lexikálny analyzér (tokenizér) obvykle rozpoznáva viaceré regulárne výrazy, ktoré korešpondujú s jednotlivými typmi tokenov, a podľa toho, ktorý z nich na aktuálnej pozícii vstupu nájde najdlhšiu zhodu, vyprodukuje ďalší token. Miesto skúšania viacerých regulárnych výrazov sa dá spraviť transformácia nedeterministického konečného automatu na deterministický, čím môžeme dosiahnuť, že časová zložitosť lexikálnej analýzy bude lineárna od dĺžky vstupu (bez ohľadu na počet pôvodných regulárnych výrazov).

Niektoré jazyky používajú aj zložitejšie pravidlá. Napríklad môžu mať významné biele znaky a označovať bloky kódu nie kľúčovými slovami `begin` a `end`, alebo zátvorkami `{` a `}`, ale odsadením riadkov. Keď tokenizér uvidí riadok, ktorý začína viac medzerami, ako predošlý, vloží na toto miesto jeden token „INDENT“, a keď menej medzerami, vloží jeden alebo viacero tokenov „DEDENT“ (môže sa končiť viacero blokov naraz). Taký tokenizér si musí pamätať predošlé úrovne odsadenia, takže už nie je regulárny. Ale aj v prípadoch, keď nejde použiť deterministický konečný automat, bývajú tokenizéry zväčša jednoduché a rýchle, a ťažkú prácu delegujú na samotný parser.

Keďže výsledkom lexikálnej analýzy je prúd tokenov, mohli by sme celý vstup premeniť na tokeny dopredu, a až potom začat parsovať. Ale väčšinou sa to tak nerobí. Miesto toho si parser pýta tokeny po jednom a tokenizér ich vyrába až vtedy, keď treba. Implementácia v praxi nie je o nič komplikovanejšia a netreba mať v pamäti pole tokenov. Tiež to rieši problémy v jazykoch ako napr. C, ktorého gramatika potrebuje rozlišovať medzi menami typov a menami premenných. To sa často rieši cez „the lexer hack“ – tokenizér dostáva od parsera za behu informácie o definíciach typov, aby mohol pre mená premenných a typov vyrábať rôzne tokeny. Keby sa parser spustil až po tom, čo skončí tokenizér, niečo také by nebolo možné.

Keď tokenizér dočíta celý vstup a preloží ho celý na prúd tokenov, na záver vyprodukuje ešte špeciálny token „$“ reprezentujúci koniec vstupu. Ten sa priamo nenachádza v žiadnom pravidle gramatiky, ale LL aj LR parserom oznamuje, kedy skončil vstup. V prípade LL parserov sa tento špeciálny terminál celý čas nachádza na spodku zásobníka, takže musí nasledovať hneď po počiatočnom neterminál σ. Čiže ak za textom odvodeným zo σ nasleduje ešte niečo ďalšie, nie koniec vstupu, je to chyba. A tiež je chyba, ak naopak koniec vstupu príde priskoro a na vrchu zásobníka je iný terminál alebo neterminál. LL parser korektne skončí iba vtedy, keď je jeho zásobník prázdny.

1.2.2 Vnorené jazyky pre LL parser

V predchádzajúcom texte sme si ukázali štandardné LL parsery. Teraz sa pozrieme na to, ako ich Multiparser upraví, aby podporovali vkladanie iných jazykov. Ako sme uviedli v sekcii 1.1, celková analýza bude mať hierarchický charakter. Keď náš LL parser uvidí na vstupe korektný otvárací oddeľovač, musí zavolať parser pre daný jazyk. A keď je naopak vnorený v inom jazyku a uvidí zatvárací oddeľovač, musí predčasne skončiť a vrátiť výsledok.

V celom Multiparseri chceme, aby sa vnorený text napísaný v inom jazyku zvonka javil ako jeden celok. To v reči LL parserov znamená, že by to mal byť jeden token. Vďaka tomu bude ľahké upraviť gramatiku a pridať na správne miesto schopnosť vnárať kód v inom jazyku. Jeho štruktúra bude z pohľadu gramatiky nezaujímavá, rovnako ako štruktúra číselných alebo reťazcových literálov.

To znamená, že o detekciu vnorených jazykov sa musí starať tokenizér. Medzi regulárne výrazy, ktorými hľadá jednotlivé tokeny, pribudne ďalší – taký, ktorý hľadá otvárací oddeľovač. (Alebo viaceré, ak existuje viacero povolených otváracích oddeľovačov.) Keď ho tokenizér nájde, zavolá parsovaciu funkciu pre vnorený jazyk, a z jej výsledku vyrobí token. Každá parsovacia funkcia má za úlohu aj posunúť pozíciu vo vstupnom bufferi až za zatvárací oddeľovač. Ten totiž môže závisieť od vnútorného jazyka.

V sekcii 1.2.1 sme upozornili, že tokenizácia neprebieha celá pred parsovaním, ale za behu. Tokenizér zvyčajne poskytuje parseru funkciu, ktorá vygeneruje jeden nasledujúci

token (a presne tak je to aj v implementácii Multiparsera). To znamená, že keď je do jazyka A vnorený jazyk B (oba používajúce LL parser), na zásobníku volaní sú v istý moment štyri funkcie: na spodku je „parser pre A“ – LL automat konštruujúci strom odvodenia podľa gramatiky A. Ten zavolal funkciu „prečítaj ďalší token podľa jazyka A“, ktorá ale našla otvárací oddeľovač a preto zavolala „parser pre B“, a ten podobne volá funkciu „prečítaj ďalší token podľa jazyka B“.

Otváracie oddeľovače sa teda dajú priamočiaro doplniť do pravidiel tokenizéra. Spracovanie zatváracích oddeľovačov bude o čosi zložitejšie. Naivný postup by bol hľadať zatvárací oddeľovač v tokenizéri ďalším regulárnym výrazom, a generovať z neho token „$“ (koniec vstupu). To je príjemné v tom, že samotný LL parser netreba nijako meniť – tento terminál už pozná a vie, že ním musí končiť vstup.

Ale Multiparser podporuje aj také zatváracie oddeľovače, ktoré sa zhodujú s už existujúcimi tokenmi. Napríklad v niektorých ukážkových nastaveniach môže jazyk nazvaný „Foo“ byť oddelený so značkami „`<?foo`“ a „`?>`“ (inšpirované z PHP a XML), ale aj „`%foo{`“ a „`}`“. Keby tokenizér prekladal znaky „`?>`“ na ukončovací token „$“, nebol by to taký problém, ale prekladať každú „`}`“ na koniec vstupu si nemôžeme dovoliť.

Zodpovednosť za detekciu zatváracích oddeľovačov musí prevziať samotný parser. Od volajúcej funkcie sa dozvie, či je vnorený v inom jazyku, a aký zatvárací oddeľovač má očakávať (v prípade, že môžu byť viaceré). Ak parsujeme jazyk najvyššej úrovne (koreň hierarchie vnorených jazykov), nemusíme nič meniť, počiatočný obsah zásobníka bude stále σ a \$. Keď sme vnorení a očakávame napríklad zatvárací oddeľovač „`}`“, na spodok zásobníka miesto \$ dáme „`}`“. Parser nebude čítať vstup až do konca súboru, ale len tú časť, ktorá sa dá odvodiť zo σ`}`. Za ňou pokračuje vonkajší jazyk.

Túto zmenu nesmieme zabudnúť zakomponovať do počítania množín $FOLLOW$. Kde v pôvodnej definícii bolo „$\sigma\$ \Rightarrow^* u\xi xv$“, musíme pridať „$\vee\ \sigma\texttt{\}} \Rightarrow^* u\xi xv$“, a podobne pre každý ďalší zatvárací oddeľovač. Takže „`}`“ sa dostane do množiny $FOLLOW(\sigma)$ a ďalších neterminálov, ktoré sa môžu vyskytovať na konci vetnej formy.

Keďže výber oddeľovačov ovplyvňuje množiny $FOLLOW$, nevhodný oddeľovač jednoducho spôsobí LL konflikt. Ak vieme zostrojiť celú parsovaciu tabuľku, máme záruku, že rozšírenie parsera o zatváracie oddeľovače nepokazilo LL vlastnosti a nespôsobilo žiadnu nejednoznačnosť. Takéto konflikty sú v praxi málokedy problém, či už používame „neobvyklé“ zatváracie oddeľovače, ako je „`?>`“, alebo zatváracie zátvorky ako „`}`“ a „`)`“. Počiatočný neterminál programovacieho jazyka je väčšinou nejaký „zoznam príkazov“ alebo „zoznam deklarácii“, a príkazy a deklarácie väčšinou nezačínajú na zatváraciu zátvorku.

V LL parseri zostáva spraviť už len jednu úpravu. V sekcii 1.2 sme napísali, že keď je na vrchu zásobníka terminál, a ten istý terminál je na vstupe, LL automat „ho vyhodí zo zásobníka a pohne sa o jeden terminál ďalej“ – prečíta ďalší token zo vstupu. Tu musíme byť

opatrní: ak vyhadzujeme zo zásobníka posledný terminál (zatvárací oddeľovač), už nesmieme načítať ďalší. V tom momente už tokenizér vnútorného jazyka prečítal zatvárací oddeľovač. Za ním ide zase vonkajší jazyk. Úloha vnútorného parsera sa v tom momente skončila a vráti kontrolu vonkajšiemu parseru.

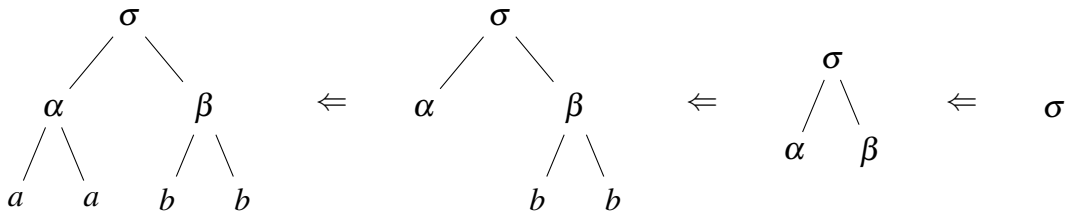
1.3 LR parsery

LR parsery [9] predstavujú ďalšiu štandardnú triedu parserov. Na rozdiel od LL parserov parsujú zdola nahor – začínajú od listov stromu odvodenia (čiže vstupného slova), a postupne ich zamieňajú za neterminály. LR parsery hľadajú pravé krajné odvodenie.

Podobne ako LL parsery, aj LR parsery majú povolené pozrieť sa na k terminálov dopredu, a podľa tohto čísla sa volajú „LR(k) parsery“. Navyše existujú viaceré varianty kanonických LR parserov – SLR parsery [3], LALR parsery [2] a GLR parsery [10]. V tejto práci sa zameriame na kanonické LR(1) parsery.

Poďme si najprv intuitívne vysvetliť, ako LR parsery pracujú. Písmeno „L“ z názvu znamená, že vstup čítajú zľava doprava, a písmeno „R“, že hľadajú pravé krajné odvodenie (ale nájdu ho v opačnom poradí, keďže čítajú zľava doprava a parsujú zdola nahor). Ukážme si jednoduchý príklad. Majme gramatiku G s pravidlami $P = \{\sigma \to \alpha\beta, \alpha \to aa, \beta \to bb \mid \varepsilon\}$ a vstupné slovo *aabb*.

Pravé krajné odvodenie pre slovo *aabb* je $\sigma \Rightarrow \alpha\beta \Rightarrow \alpha bb \Rightarrow aabb$. LR parser ho nájde v opačnom poradí: $aabb \Leftarrow \alpha bb \Leftarrow \alpha\beta \Leftarrow \sigma$. Pozrime sa na jednotlivé stromy odvodenia pre tieto vetné formy:



LR parser tieto stromy nepozná (hotový strom odvodenia pre *aabb* zistí, až keď skončí), ale môžeme pomocou nich sformulovať, čo od neho očakávame. Parser má v každom kroku nájsť najľavejší taký vrchol, že jeho synmi sú samé listy, a tieto listy odstrániť. V našom príklade najprv odstráni listy *aa*, potom *bb* a nakoniec $\alpha\beta$ (vtedy už sa stali listami). Túto skupinu listov budeme volať *handle stromu*. Úlohou parseru bude rozoznať, kde je handle, aj keď pozná iba listy a nie celý strom.

Náš parser musí byť rýchly, takže chceme, aby sa handle dal nejako rozoznať bez toho, že by sme čítali celý vstup. Definujeme gramatiky, kde sa to dá: hovoríme, že gramatika je LR(k) práve vtedy, keď je jednoznačná (každej odvoditeľnej vetnej forme prislúcha práve jeden strom odvodenia) a navyše, ak má vetná forma w handle končiaci na pozícii n a používajúci pravidlo

p, tak je jednoznačne určený prvými $n+k$ písmenami vetnej formy. To jest, každá vetná forma u, ktorá začína na rovnakých $n+k$ písmen, ako w, bude mať rovnaký handle (rovnaké n a p). (Na koniec foriem u a w ešte pridáme k špeciálnych znakov „$“ označujúcich koniec vstupu, aby sme vždy mohli hovoriť o prvých $n+k$ písmenách.)

Táto podmienka nám ešte nehovorí spôsob, ako handle hľadať. Je veľmi všeobecná a hovorí iba to, že pre danú gramatiku by sa to *mohlo* dať. Ak naša gramatika nie je LR(k), žiaden deterministický parser v nej nenájde handle bez toho, že by prečítal väčšiu časť vstupu. Ale ukazuje sa, že všetky gramatiky, čo podmienku spĺňajú, sa naozaj dajú rýchlo parsovať.

Predstavme si, že by sme už poznali celý strom odvodenia, a mali sme v ňom nájsť handle. To je jednoduché: Začneme v koreni a budeme strom postupne prehľadávať. Ak sme vo vnútornom vrchole, handle bude určite niekde pod ním, takže prejdeme na jeho prvého syna. Ak sme v liste, pokračuheme ďalej na pravého súrodenca. Keď už žiadneho pravého súrodenca nemá, našli sme handle a algoritmus končí.

Tento prehľadávací algoritmus je taký jednoduchý, že ani nepotrebuje zásobník – v strome sa nikdy nevraciame do rodiča. Táto vlastnosť je kľúčová: vďaka tomu môžeme zostrojiť nedeterministický konečný automat (NKA), ktorý si bude tipovať tvar stromu a overovať, že jeho listy sedia s vetnou formou, ktorú parser teraz má.

Ako bude vyzerať NKA pre LR(0)? Všetky stavy budú tvaru $(\xi \to u \bullet v)$, čo znamená, že sa nachádzame v synoch rodiča ξ, videli sme listy u a za nimi budú nasledovať vrcholy v. Z každého stavu pôjdu nasledovné prechody: Ak $v = xv'$ pre nejaký terminál x, v NKA bude prechod na x do stavu $(\xi \to ux \bullet v')$ (prehľadávací algoritmus prečítal ďalšie písmeno). Ak $v = \gamma v'$ pre nejaký neterminál γ, v NKA bude prechod na γ do stavu $(\xi \to u\gamma \bullet v')$ (tipli sme si, že γ je list), ale aj ε-prechod do stavu $(\gamma \to \bullet p)$ pre každé pravidlo $\gamma \to p$ (prehľadávací algoritmus zostupuje do synov – tipli sme si, že γ je vnútorný vrchol so synmi p). Ak je $v = \varepsilon$, našli sme handle a môžeme prejsť do konečného stavu. Celý výpočet začne v umelom stave $(\bullet\sigma)$, ktorý značí, že očakávame koreň stromu σ (ktorý nemá rodiča).

Čo to znamená, keď automat spustíme na vetnej forme w, a po prečítaní n znakov prejde do konečného stavu? Ešte to nemusí znamenať, že našiel handle vo w. Automat nedeterministicky prehľadáva všetky možné stromy odvodenia, ktoré sedia s w vzhľadom na prečítaných n písmen, ale môžu pokračovať inak. Takže vieme len to, že v gramatike existuje taká vetná forma u, čo začína rovnako, a má handle na tomto mieste. Ak gramatika nie je LR(0), táto informácia je zbytočná. Ale ak je LR(0), sme hotoví: u a w sa zhodujú na prvých n písmenách, z definície LR(0) preto musia mať rovnaký handle, takže automat naozaj našiel handle pre w.

Náš NKA môžeme štandardnou konštrukciou previesť na DKA (odstránime ε-prechody a vyrobíme DKA, ktorého stavy korešpondujú s množinami stavov NKA). Vďaka tomu vieme v LR(0) gramatike postupne čítať vetnú formu a v momente, keď sme prečítali celý handle, o tom budeme vedieť.

Zostrojme NKA aj pre LR(k) gramatiky s $k > 0$. Predtým sme si nedeterministicky tipovali štruktúru stromu – čo je list a čo je vnútorný vrchol, a aké pravidlá používame. Teraz navyše budeme tipovať, akých k znakov (terminálov alebo \$) bude na vstupe nasledovať za handlom. Aby si automat mohol svoj tip overiť, dostane schopnosť pozerať sa na k znakov dopredu. Každý stav bude tvaru $(\xi \to u \bullet v, w)$, kde w je predpovedaná k-tica znakov, ktorú očakávame na vstupe za ξ. Stavové prechody budú nasledovné: ak $v = xv'$, automat môže opäť pri prečítaní x prejsť z $(\xi \to u \bullet xv', w)$ do $(\xi \to ux \bullet v', w)$, a podobne ak $v = \gamma v'$, môžeme pri prečítaní γ prejsť z $(\xi \to u \bullet \gamma v', w)$ do $(\xi \to u\gamma \bullet v', w)$. Zostupovanie do neterminálov sa oproti LR(0) automatu mierne zmení: ε-prechod z $(\xi \to u \bullet \gamma v', w)$ do $(\gamma \to \bullet p, w')$ povolíme pre všetky gramatické pravidlá $\gamma \to p$ a všetky tie k-znakové w', kde w' môže nasledovať po γ, čiže platí $v'w \Rightarrow^* w'w''$ pre nejaké w''. A nakoniec, ak $v = \varepsilon$, automat prejde do konečného stavu, ale iba vtedy, keď na vstupe naozaj nasleduje w. Celý výpočet začne v stave $(\bullet\sigma, \$^k)$, lebo po σ umelo pridáme k ukončovacích znakov \$.

Preklad tohto NKA na DKA trochu technicky komplikuje fakt, že má schopnosť pozerať sa na k znakov dopredu. NKA to využíva iba pri záverečnej kontrole, a pri čítaní kontroluje iba ten najbližší znak. Ale v DKA musíme pre každý prechod vymenovať, aké sú povolené k-tice, čo môžu nasledovať na vstupe. Pri $k > 1$ to znamená, že sa musíme pozrieť, čo všetko vieme dosiahnuť na k NKA-krokov dopredu. Napríklad keby sme robili LR(2) parser pre gramatiku s $P = \{\sigma \to a\beta, \beta \to b\}$, v DKA bude prechod zo stavu $\{(\sigma \to \bullet a\beta, \$\$)\}$ do stavu $\{(\sigma \to a \bullet \beta, \$\$)\}$ povolený iba vtedy, keď na vstupe nasleduje ab. To závisí nielen od znaku, čo nasleduje hneď za „•“ (v tomto prípade a), ale aj od definície pre β, respektíve od ďalších stavov automatu. DKA musí „vopred“ zistiť, že môžeme vojsť do β a potom prečítať b. Situáciu navyše komplikujú vymazávajúce neterminály.

Toto je ale iba technická komplikácia, a vhodný DKA sa napriek tomu dá skonštruovať pomocou upravených *FIRST* množín. (Viď definíciu $H'_k(\sigma)$ v [9].) Detaily tejto konštrukcie tu pre jednoduchosť nebudú uvedené. Keď sa obmedzíme na $k = 1$, stačí nám pozerať sa na znak, čo nasleduje hneď za „•“, a je možné použiť štandardnú konštrukciu DKA.

Teraz už pre všetky LR(k) gramatiky poznáme deterministický automat, ktorému môžeme dať ľubovoľnú vetnú formu, a dozvieme sa, kde je v nej handle. Automat pritom nepotrebuje čítať ďalej, ako k znakov za koncom handlu. Vyzbrojení týmto automatom ľahko zostrojíme prvý funkčný parser. Ako sme si povedali vyššie, chceme v opačnom poradí zostrojiť pravé krajné odvodenie. Budeme teda postupne konštruovať vetné formy v tomto odvodení. Začneme s celým vstupným slovom. Spustíme na ňom automat a dozvieme sa, kde je handle. Celý handle teda vyškrtneme a nahradíme daným neterminálom, čím dostaneme druhú vetnú formu v pravom krajnom odvodení. To opakujeme, až kým nezostane iba σ. Keď sa pozrieme, aké vetné formy sme videli, ľahko z nich určíme strom odvodenia pôvodného slova.

Takýto parser by bol funkčný, ale pomalý. Konštruovať zakaždým novú vetnú formu

a spúšťať na nej automat vždy odznovu je zbytočne neefektívne. Miesto toho použijeme zásobník, kde si budeme pamätať nielen prečítané znaky, ale aj stavy, v ktorých bol automat po prečítaní každého znaku. Keď automat nájde handle, vyhodíme ho zo zásobníka a pridáme tam nový neterminál. Ale automat už nemusí ísť znova od začiatku – pred handlom sa nič nezmenilo, takže sa môže vrátiť do zapamätaného stavu. Celková časová zložitosť bude vďaka tomu lineárna od dĺžky vstupu.

1.3.1 Vnorené jazyky pre LR parser

Podobne ako v sekcii 1.2.2 si teraz ukážeme, ako formalizmus LR parserov rozšíriť, aby sme mohli do LR jazyka vkladať iné jazyky a aby sme ho mohli vkladať do iných jazykov.

LR parsery sú podobne ako LL parsery spojené s tokenizérom, ktorý predspracováva vstup. Hľadanie otváracích oddeľovačov a vkladanie iných jazykov do LR jazyka bude preto riešené priamo v tokenizéri, rovnako ako pri LL parseroch.

Parser ale musí podporovať zatváracie oddeľovače. A to aj také, ktoré sa môžu vyskytovať ako terminály aj vnútri gramatiky. Ako príklad nám podobne ako pri LL parseroch poslúži pravá kučeravá zátvorka „}“. Programovacie jazyky ju často používajú na ukončovanie blokov príkazov – mohli by sme chcieť, aby ukončovala aj celý „blok príkazov v inom jazyku“.

Za zatváracím oddeľovačom už nesmieme čítať ďalšie tokeny. Tie už sú zase napísané vo vonkajšom jazyku, ktorý môže mať iné pravidlá tokenizácie. Toto je ďalším dôvodom, prečo používame iba LR(1) parsery, a nie $k > 1$. (Hlavným dôvodom je, že rozpoznávajú rovnakú triedu jazykov, ako LR(1). [9])

Ako upraviť LR(1) parser, aby nečítal až do konca súboru, ale iba po zatvárací oddeľovač? Netreba k tomu veľa. Jediné, čo musíme zmeniť, je začiatočný stav. Náš NKA pôvodne začínal v stave $(\bullet\sigma, \$)$. Keď chceme čítať iba po „}“, použijeme začiatočný stav $(\bullet\sigma, \})$. Analogicky sa zmení aj DKA.

Samozrejme, ak podporujeme viacero zatváracích oddeľovačov, nie je nutné konštruovať samostatný automat pre každý z nich. Stačí si pamätať, v ktorom stave začíname pre ktorý oddeľovač. Situácia je úplne rovnaká, ako keď v gramatike chceme mať viacero počiatočných neterminálov, a vyberať si z nich podľa situácie – príkaz verzus výraz, a podobne.

Teraz si ukážeme, prečo je táto zmena korektná, a nevybočuje z formalizmu LR parserov. Nevhodne zvolený zatvárací oddeľovač môže spôsobiť konflikty, takže by sa zdalo, že musíme upraviť definíciu LR(k) gramatiky, a okrem vetných foriem ukončených znakom „$“ zahrnúť aj tie, čo sú ukončené znakom „}“. Podobným spôsobom sme pri LL parseroch rozšírili definíciu množín $FOLLOW$.

Tentoraz použijeme iný postup. Na základe pôvodnej gramatiky G zostrojíme upravenú gramatiku G'. Pridáme nový štartovný neterminál σ' s jediným pravidlom $\sigma' \to \sigma\}\eta$, a nový neterminál η, z ktorého sa dá odvodiť čokoľvek ($\eta \to \varepsilon; \forall x \in T : \eta \to x\eta$). Pre gramatiku G'

skonštruujme LR(1) parser. Ak sa to nedá a G' nie je LR(1) gramatikou, zvolili sme nevhodný oddeľovač. Ak sa to dá, pozrime sa na NKA pre tento parser.

NKA bude začínať v stave $(\bullet\sigma', \$)$, z ktorého môže prejsť iba do stavu $(\sigma' \to \bullet\sigma\}\eta, \$)$ a z neho do stavov $(\sigma \to \bullet p, \})$ pre všetky pravidlá $\sigma \to p$. NKA si teda tipne, že v strome odvodenia má σ synov p, a za nimi na vstupe nasleduje „}“. Keď sa to naozaj stane, parser vymaže zo zásobníka handle a pridá tam σ. NKA sa tým dostane do stavu $(\sigma' \to \sigma \bullet \}\eta, \$)$. Potom už len prečíta „}“ (o ktorej už vieme, že na vstupe naozaj nasleduje), a odignoruje zvyšok vstupu.

Vidíme, že ak pre G' naozaj vieme zostrojiť LR(1) parser, správa sa takmer presne tak, ako by sme chceli. Vďaka nedeterminizmu si dokáže tipnúť, ktoré „}“ sú normálne terminály vnútri odvodenia σ, a ktorá „}“ predstavuje koniec celého σ. Porovnajme tento parser pre G' s naším parserom pre G, ktorý začína v stave $(\bullet\sigma, \})$. Z tohto stavu opäť idú ε-prechody na $(\sigma \to \bullet p, \})$ pre všetky pravidlá $\sigma \to p$. Oba parsery robia to isté, až do momentu, keď zo zásobníka vymažú handle a dajú tam σ. Náš parser pre G v tom momente úspešne končí, zatiaľčo parser pre G' číta až do konca, pričom všetko ignoruje.

Takže začínať zo stavu $(\bullet\sigma, \})$ nie je nič „výnimočné“. Vzniknutý parser je naozaj korektným LR parserom. Len parsuje trochu inú gramatiku – vyžaduje, aby definíciu LR(1) gramatík spĺňala G', nie G. Inak je zvláštny už len tým, že si šetrí prácu a číta vstup len po zatvárací oddeľovač, nie až do konca.

1.4 PEG a packrat parsery

Tretí typ parserov, ktorý si ukážeme, sa od LL a LR parserov bude značne líšiť. Nebude ani založený na bezkontextových gramatikách. Namiesto toho používa PEG [6] – „parsing expression grammars“, čiže „gramatiky parsovacích výrazov“.

Pravidlá v bezkontextovej gramatike v prvom rade popisujú, ako generovať slová jazyka. Každý neterminál v istom zmysle predstavuje množinu slov, ktoré z neho môžeme odvodiť. Vytvoriť pre bezkontextovú gramatiku parser nemusí byť triviálne, ako sme sa mohli presvedčiť v predchádzajúcich sekciách. PEG používajú opačný prístup, a v prvom rade popisujú, ako slová rozoznávať. V PEGu predstavuje každý neterminál funkciu, ktorá číta vstupné slovo a vracia výsledok.

PEG vyzerá na prvý pohľad podobne ako bezkontextová gramatika – tiež má neterminály, terminály a pravidlá – ale v detailoch sa líši. Pravidlá sú tvaru $A \leftarrow e$, kde A je neterminál a e je takzvaný *parsovací výraz*. Každý neterminál musí byť na ľavej strane práve jedného pravidla. Parsovacie výrazy sú rekurzívne definované nasledovne:

- Každý terminál $t \in T$ je parsovací výraz.
- Každý neterminál $X \in N$ je parsovací výraz.

- Bodka . aj epsilón ε sú parsovacie výrazy.
- Ak sú e_1 a e_2 parsovacie výrazy, aj $e_1 e_2$ a e_1 / e_2 sú parsovacie výrazy.
- Ak je e parsovací výraz, aj $e*$, $e+$, $e?$, $\&e$ a $!e$ sú parsovacie výrazy.
- Nič iné nie je parsovací výraz.

Neterminály si môžeme predstaviť ako vzájomne rekurzívne funkcie, a parsovacie výrazy ako ich kód. Každá funkcia bude vidieť zdieľanú premennú n, v ktorej bude uložená „aktuálna poloha“ na vstupe. Túto premennú môže počas svojho behu ľubovoľne meniť, nakuknúť dopredu a neskôr sa vrátiť späť. Vstup nemusí čítať postupne.

Keď kód pre nejaký parsovací výraz spustíme s nejakou konkrétnou hodnotou n, môže buď vrátiť „úspech“, alebo „neúspech“, alebo sa môže večne zacykliť. Ak skončí úspešne, nová hodnota n môže byť rovnaká (na vstupe sme sa nepohli) alebo väčšia („prečítali“ sme niekoľko znakov). Ak skončí neúspešne, premenná n musí mať rovnakú hodnotu, ako na začiatku.

Jednotlivé parsovacie výrazy majú nasledovný význam. Tu si ukážeme iba „pseudokód“ každého výrazu – formálnejšie by sme mohli PEG definovať pomocou rekurzívne definovanej relácie krokov odvodenia, alebo kompletnou operačnou sémantikou.

t: *Konkrétny terminál.* Ak je n-tým znakom vstupu t, inkrementuj n (posuň sa za ten znak) a vráť úspech. Inak vráť neúspech.

.: *Ľubovoľný terminál.* Ak si na konci vstupu, vráť neúspech. Inak inkrementuj n (posuň sa za ten znak) a vráť úspech. (Výraz . by sme mohli brať ako syntaktický cukor pre $x_1 / x_2 / \ldots / x_n$, kde x_i sú zaradom všetky terminály.)

A: *Neterminál.* Rekurzívne zavolaj parsovaciu funkciu pre A a vráť jej výsledok. (Každý neterminál má práve jedno pravidlo, ktoré mu priraďuje nejaký parsovací výraz.)

ε: *Prázdny reťazec.* Vždy vráť úspech a pozíciu na vstupe n nechaj nezmenenú.

$e_1 e_2$: *Zreťazenie.* Zapamätaj si aktuálnu pozíciu n ako n_0. Vykonaj pseudokód pre e_1. Ak vrátil neúspech (a neposunul sa), tiež vráť neúspech. Ak vrátil úspech (a možno sa posunul na novú pozíciu), vykonaj e_2. Ak vrátil neúspech, nastav n naspäť na n_0 a tiež vráť neúspech. Ak vrátil úspech, vráť úspech.

e_1 / e_2: *Usporiadaná voľba.* Vykonaj pseudokód pre e_1. Ak vrátil úspech, tiež hneď vráť úspech (e_2 ani nevyskúšame). Ak vrátil neúspech (a teda nezmenil n), vykonaj pseudokód pre e_2 a vráť jeho výsledok.

$e*$: *Nula alebo viac opakovaní.* Opakuj pseudokód pre e, až kým nevráti neúspech. Potom vráť úspech.

$e+$: *Jedno alebo viac opakovaní.* Syntaktický cukor pre $ee*$.

$e?$: *Nula alebo jedno opakovanie.* Syntaktický cukor pre e / ε.

&*e*: *Predikát „a zároveň“*, resp. *pozitívny lookahead*. Zapamätaj si aktuálnu pozíciu n ako n_0. Vykonaj pseudokód pre e. Ak vrátil úspech, nastav n naspäť na n_0 a vráť úspech (neposúvame sa na vstupe). Ak vrátil neúspech, vráť neúspech. (Výraz &*e* by sme mohli brať ako syntaktický cukor pre !!*e*.)

!*e*: *Predikát „zápor“*, resp. *negatívny lookahead*. Zapamätaj si aktuálnu pozíciu n ako n_0. Vykonaj pseudokód pre e. Ak vrátil neúspech, nechaj n nezmenené a vráť úspech. Ak vrátil úspech, nastav n naspäť na n_0 a vráť neúspech.

Bezkontextové gramatiky sa skladajú z množiny neterminálov, množiny terminálov, množiny pravidiel a počiatočného neterminálu. PEGy sa skladajú z množiny neterminálov, množiny terminálov, množiny pravidiel a počiatočného *parsovacieho výrazu*. To znamená, že jednoduché PEGy nemusia mať vôbec žiadne neterminály a pravidlá. Jazyk rozoznávaný konkrétnym PEGom je jazyk všetkých slov, pre ktoré počiatočný parsovací výraz vráti úspech, keď n začne na nule. Na finálnej hodnote n nezáleží – počiatočný parsovací výraz nemusí skončiť na konci vstupného slova.

Zápis PEGov miestami pripomína regulárne výrazy a bezkontextové gramatiky, ale vidíme, že v sémantike sa líšia. Hlavný rozdiel je v tom, že v PEGu je parsovanie *greedy* (pažravé). Napríklad $e*$ predstavuje v regulárnych aj parsovacích výrazoch nula alebo viac opakovaní výrazu e. Regulárny výraz $a*a$ uspeje pre všetky vstupy, čo sa skladajú z jedného alebo viacerých písmen a – nedeterministicky sa určí, že podvýraz $a*$ skončí tesne pred tým posledným a-čkom. Ale parsovací výraz $a*a$ je greedy a nikdy neuspeje, lebo podvýraz $a*$ vždy prečíta všetky a-čka, čo môže. Podobne, v bezkontextovej gramatike je k dispozícii neusporiadaná voľba $\alpha \to u \mid v$, v ktorej sú u a v rovnocenné – parser si môže vybrať, ktorú vetvu použije. PEG poskytuje iba usporiadanú voľbu $A \leftarrow u/v$, ktorá je greedy. Najprv skúsi u, a ak uspeje, v sa ignoruje.

Keď už máme predstavu o sémantike PEGov, pozrime sa, akými ďalšími spôsobmi sa dajú parsovať. Všimnime si, že pseudokód uvedený vyššie síce používa zdieľanú premennú n, ale ľahko ho môžeme upraviť, aby sme sa premenlivým premenným vyhli. Namiesto toho, že by parsovacie funkcie jednotlivých neterminálov čítali a písali do jednej premennej, môže každá funkcia dostať aktuálnu pozíciu na vstupe ako argument. Ich „obor hodnôt“ už nebude iba booleovský $\{\text{úspech}, \text{neúspech}\}$, ale $\mathbb{N} \cup \{\text{neúspech}\}$ – pri úspechu vrátime z funkcie novú pozíciu na vstupe (čo sme doteraz vracali v podobe novej hodnoty premennej n).

Táto interpretácia sa zvlášť hodí pre funkcionálne programovanie. Z každej parsovacej funkcie sa totiž stáva *čistá funkcia*, ktorá nemá žiadne vedľajšie účinky a pre rovnaký vstup (vrátane rovnakej pozície na vstupe) vráti vždy rovnaký výstup (ak sa nezasekne vo večnej rekurzii).

To nám prináša aj ďalšiu výhodu, či už používame funkcionálny programovací jazyk alebo nie. Štandardný parser, implementovaný priamočiaro podľa vyššie uvedeného pseudokódu,

môže mať až exponenciálnu časovú zložitosť. Ale ak každý neterminál zapíšeme ako funkciu bez vedľajších účinkov, môžeme použiť memoizáciu. Keď sa zavolá funkcia A s nejakým argumentom n, zapamätáme si jej výsledok, aby sme ho mohli hneď vrátiť, keď sa zavolá znovu s rovnakým n. Takýto memoizujúci parser sa volá *packrat parser* [5]. Vďaka memoizácii dosahuje časovú a pamäťovú zložitosť $O(lg)$, kde l je dĺžka vstupu a g je „veľkosť gramatiky“ (súčet dĺžok parsovacích výrazov).

Greedy vlastnosti PEGov nie sú dobré len na to, aby sa dali rýchlo parsovať. Týmto modelom napríklad dokážeme rozoznať aj niektoré jazyky, ktoré nie sú bezkontextové. Jednoduchým príkladom je jazyk $L = \{a^n b^n c^n \mid n \geq 1\}$. Dokáže ho rozoznať PEG s pravidlami $\{A \leftarrow a\ A?\ b, B \leftarrow b\ B?\ c\}$ a počiatočným výrazom „&A $a+$ B !.“. Neterminál A reprezentuje výrazy $a^n b^n$ a B reprezentuje výrazy $b^n c^n$. V počiatočnom výraze najprv pomocou &A skontrolujeme, že a-čiek a b-čiek je rovnako veľa, ale potom sa vrátime na začiatok vstupu – & spôsobí, že ak parsovanie A uspeje, n sa vráti na predošlú hodnotu. Potom pomocou $a+$ preskočíme všetky a-čka a pomocou B skontrolujeme, že aj b-čiek a c-čiek je rovnako veľa. Výraz !. overí, že bol dočítaný celý vstup.

Programovacie jazyky zapísané ako PEG navyše nepotrebujú samostatný tokenizér, a ako terminály môžu použiť priamo znaky súboru. Greedy opakovanie a usporiadaná voľba ľahko prevezmú štandardné úlohy tokenizácie: preskakovanie bieleho miesta, rušenie komentárov, čítanie stringových reťazcov a escape sekvencií. Negatívny lookahead zase vyrieši, aby kľúčové slová neboli považované za validné mená premenných.

1.4.1 Vnorené jazyky pre PEG

Rovnako ako pre LL a LR parsery, aj PEGy, resp. packrat parsery chceme rozšíriť o podporu otváracích a zatváracích oddeľovačov. Chceme, aby otvárací oddeľovač spôsobil spustenie iného vnoreného parsera, aby sa do jazyka zapísaného ako PEG dali vnárať ďalšie. A naopak, keď je náš jazyk vnorený v nejakom inom jazyku, chceme zatvárací oddeľovač brať ako predčasný „koniec súboru“ a vrátiť kontrolu vonkajšiemu parseru.

Odlišnosť PEG od LL a LR parserov znamená, že aj podpora oddeľovačov sa bude musieť líšiť. Zatiaľčo pri LL a LR parseroch sme otváracie oddeľovače mohli spracovať počas lexikálnej analýzy, PEG parsery žiadnu lexikálnu analýzu nepotrebujú.

To, čo by v bezkontextovej gramatike bol jeden terminál, je v PEGu väčšinou vyjadrené ako vhodný neterminál. Uvažujme napríklad jazyk, v ktorom sa operátor rovnosti píše ako „==“. Keby sme ho parsovali s LL alebo LR parserom, tokenizéru by sme povedali, aby znaky == preložil na jediný token „*rovná_sa*“, ktorý by vystupoval ako terminál v danej bezkontextovej gramatike. Tokenizér by navyše mal všelijaké ďalšie pravidlá o tom, že má ignorovať biele znaky a komentáre. V PEGu by sme miesto toho mohli mať neterminál „*ROVNÁ_SA*“ a pravidlo „*ROVNÁ_SA* ← *BIELY_ZNAK** ‘=’ ‘=’ *BIELY_ZNAK**“.

Jediné primitívne parsovacie výrazy, ktoré PEG pozná, sú jeden konkrétny terminál (znak), jeden konkrétny neterminál a epsilón. (Bodka, ktorá číta ľubovoľný znak, sa dá pokladať za syntaktický cukor.) Takže PEG v konečnom dôsledku číta celý vstup znak po znaku. Pri LL a LR jazykoch sa tokenizér staral nielen o hľadanie otváracieho oddeľovača, ale aj o zavolanie parsera pre daný vnorený jazyk, aby sa s celým vnoreným programom dalo pracovať ako s jedným tokenom. Ak sa obmedzíme na štandardnú definíciu PEGov, ktorá povoľuje iba čítanie znak po znaku, nič také by sme nemohli spraviť.

Ale praktické implementácie PEG parserov formálnu definíciu často rozširujú. Napríklad umožňujú ako atomický *parsovací* výraz použiť aj štandardný *regulárny* výraz, s klasickou nepažravou sémantikou. Samozrejme ho treba vhodne označiť. Toto rozšírenie je v poriadku: neruší záruky parsera, ani nezhoršuje jeho zložitosť. Hľadať na vstupe od nejakej pozície daný regulárny výraz je tiež memoizovateľná operácia bez vedľajších účinkov. Buď vráti úspech a prečíta zo vstupu nula alebo viac znakov, alebo skončí neúspešne. A nič viac nevyžadujeme. (Ak by sme striktne vyžadovali lineárnu časovú zložitosť, mohli by sme predrátať výsledok pre každú pozíciu a každý stav regulárneho automatu. Či sa to prakticky oplatí závisí od situácie.)

Podobne teda pridáme naše vlastné rozšírenie. Volanie parsera pre vnorený jazyk „Foo“ označme napríklad novým výrazom P_{foo}, ktorý pridáme do definície parsovacích výrazov. Sémantika tohto výrazu bude zavolanie daného parsera od aktuálnej polohy na vstupe.

Keby sme chceli použiť otvárací oddeľovač „`<?foo`“, ako v príklade pri LL parseroch, tento primitívny výraz by sme zabalili do väčšieho pravidla „*VNORENÝ_FOO_PROGRAM* ← *BIELY_ZNAK** ‘`<`’ ‘`?`’ ‘`f`’ ‘`o`’ ‘`o`’ P_{foo} *BIELY_ZNAK**“. Toto pravidlo by preskočilo biele znaky, prečítalo otvárací oddeľovač, a ak by ho naozaj našlo, zavolalo by parser jazyka Foo. (Ten by sa časom postaral o prečítanie zatváracieho oddeľovača.)

Toto rozšírenie je bohužiaľ podstatne silnejšie, než regulárne výrazy. Parsovanie vnoreného jazyka sa nedá len tak ľahko memoizovať. Keď rozšírime PEG model o plnú silu výrazu P_{foo}, dala by sa skonštruovať aj taká PEG gramatika, ktorá sa pokúsi hľadať P_{foo} (a teda spustí vnorený parser pre Foo) od každej možnej polohy vstupu. A navyše, parsery prakticky používaných programovacích jazykov môžu mať aj vedľajšie účinky – v jazyku C si napríklad musíme udržiavať tabuľku definovaných typov, aby sme vedeli rozlišovať mená typov od premenných.

Tieto problémy sú našťastie iba teoretické. V praxi neskúšame výraz P_{foo} po každom znaku na vstupe, ale iba vtedy, keď nájdeme otvárací oddeľovač. Také je aj pravidlo pre *VNORENÝ_FOO_PROGRAM* vyššie. Mohli by sme zmeniť sémantiku parsovacieho výrazu P_{foo}, aby sám overil prítomnosť otváracieho oddeľovača, ale to by obmedzilo flexibilitu pri výbere neobvyklých otváracích oddeľovačov. Multiparser sa prikláňa na stranu flexibility a dáva autorovi gramatiky k dispozícii celý P_{foo}. Autor túto moc musí používať zodpovedne.

Otázka je, čo sa má stať, ak parser vnoreného jazyka Foo zlyhá – ak za otváracím oddeľovačom nenasleduje korektný Foo program. Parsovacie výrazy môžu vracať úspech aj neúspech, takže aj výraz P_{foo} by sme mohli definovať tak, že v tomto prípade jednoducho vráti neúspech. Ale to by malo iba pochybné využitie. Ak vonkajší jazyk používa P_{foo} tak, ako to bolo myslené, čiže za jednoznačným otváracím oddeľovačom, so syntaktickou chybou vo vnútornom parseri veľa nespraví. Multiparser to preto berie ako fatálnu chybu a syntaktická chyba vo vnorenom jazyku okamžite ukončí parsovanie celého vstupu, či už je vonkajší jazyk založený na LL gramatike, LR gramatike alebo PEG gramatike.

A čo so zatváracími oddeľovačmi? LL a LR gramatiky používali token „$“ iba nepriamo v parseri, ale nebol priamo súčasťou gramatiky. Podpora zatváracích oddeľovačov si preto vyžadovala úpravu parsera. V PEGu je to zase naopak. Definícia hovorí, že počiatočný parsovací výraz musí vrátiť úspech, ale nemusí prečítať celý vstup. Parsovací výraz $a*$ vráti úspech aj na vstupnom slove *bbb*. Autor PEG gramatiky preto musí na koniec výrazu napísať „!.“ – tento podvýraz vráti úspech iba vtedy, keď už sme na konci. (To sme použili aj v príklade pre jazyk $a^n b^n c^n$ v predošlej sekcii.)

Koniec vstupu teda nie je riešený na úrovni parsera, ale musí sa o to ad-hoc starať samotná gramatika. Podporu záverečného oddeľovača preto uskutočníme úpravou použitej gramatiky. Ak pôvodná gramatika hľadala koniec vstupu pomocou podvýrazu „!.“, a chceme ju napríklad adaptovať pre zatvárací oddeľovač „?>“, podvýraz „!.“ prepíšeme na „?>“.

Nevzniknú žiadne „konflikty“, lebo PEGy sú z definície vždy jednoznačné. Všetky „nejednoznačnosti“ sú rozhodnuté podľa poradia, v ktorom sú uvedené možnosti usporiadanej voľby e_1/e_2. Otázka je, či budú rozhodnuté tak, ako by sme očakávali. Ale toto nie je problém špecifický pre zatváracie oddeľovače. Je to základná vlastnosť PEGov ako takých, a závisí len od nášho uhla pohľadu, či je to chyba, alebo dobrá vlastnosť.

Poznamenajme, že PEGy dávajú vzniknúť aj iným zdegenerovaným prípadom. Vďaka pozitívnemu a negatívnemu lookaheadu môžu parsovacie výrazy PEGov „nakuknúť“ aj dovnútra textov napísaných v inom jazyku. Napríklad keby počiatočný parsovací výraz začínal podvýrazom „$\&(.*x)$“, korektné by boli len tie vstupy, ktoré niekde obsahujú písmeno x. To môže viesť k nečakanému správaniu. Ale opäť sa prejavuje iba to, že PEGy sa prikláňajú na stranu flexibility. PEGy majú tento „problém“ z princípu, a nič by sa nezmenilo, ani keby sme nemali vnorené jazyky. Výrazy $\&e$ a $!e$ sa vždy dajú použiť mätúco, ale dávajú PEGom veľkú silu – umožňujú nielen rozoznávať jazyky ako $a^n b^n c^n$, ale aj ľahko popisovať jazyky ako „všetky slová z písmen abecedy, okrem `if` a `while`“, vďaka čomu nepotrebujú samostatný tokenizačný krok a celý programovací jazyk sa dá opísať jedinou gramatikou.

Kapitola 2

Implementácia

Systém Multiparser je praktickou implementáciou myšlienok predstavených v predchádzajúcej kapitole. Používateľ Multiparsera si vyberie gramatiky, ktoré chce parsovať, a nastaví, ako sa do seba môžu vnárať a aké otváracie a zatváracie oddeľovače sa dajú použiť. Systém Multiparser vyrobí pre každú vstupnú gramatiku vlastný parser, a prepojí ich, aby sa vygenerované parsovacie funkcie mohli navzájom volať.

Jadro Multiparsera sa delí na dve časti: *generátory parserov*, ktoré vedia previesť zápis gramatiky na spustiteľný parser, a na *runtime podporu*, ktorú vygenerované parsery používajú ako spoločný základ a platformu pre vzájomnú komunikáciu. Celý Multiparser je implementovaný v jazyku Python 3 [15], a výstupom generátorov je priamo Pythonový kód.

Ako príklady obsahuje Multiparser viacero ukážkových „hračkárskych“ jazykov, ktoré demonštrujú jeho schopnosti. Väčšina z nich vznikla ako zjednodušená forma ozajstných programovacích jazykov, čo sa používajú v praxi. Tieto „hračkárske“ jazyky síce nie sú stopercentne kompatibilné s ich originálmi, ale kopírujú ich najdôležitejšie aspekty. A to vrátane špeciálnych parsovacích pravidiel a výnimiek z formálneho gramatického modelu, aké sa v praktických programovacích jazykoch zvyknú objaviť. Na tieto jazyky sa bližšie pozrieme v kapitole 3.

2.1 Rozhranie parserov

Celkový výsledný program – viacjazyčný parser – sa skladá z viacerých oddelených parserov pre jednotlivé jazyky. Nech majú akúkoľvek implementáciu a vnútornú štruktúru, poskytujú jednotné rozhranie. Každý je Pythonová funkcia, ktorá ako prvý argument dostane „parsovací kontext“ a ako výsledok vráti spracované dáta – obvykle v podobe abstraktného syntaktického stromu.

„Parsovací kontext“ je objekt, v ktorom je uložené všetko, čo je spoločné pre celé parsovacie sedenie. V prvom rade udržiava samotný vstupný buffer a aktuálnu pozíciu v ňom.

(Multiparser nečíta vstup priebežne z disku, ale drží ho celý v pamäti, aby mohol podporovať PEG.) Parsovanie súboru začne tým, že program vyrobí inštanciu triedy `ParserContext` (zo súboru `context.py`) a dá ju parsovacej funkcii pre jazyk najvyššej úrovne. Keď jedna parsovacia funkcia volá druhú, dá jej tú istú inštanciu `ParserContext`, takže vnútorný parser môže čítať ten istý vstup a vonkajší parser sa potom dozvie, ako sa zmenila pozícia.

Aktuálna pozícia na vstupe je v kontexte reprezentovaná dvoma spôsobmi – priamo ako počet prečítaných znakov, a nepriamo ako číslo aktuálneho riadku a stĺpca. Ten prvý spôsob sa používa na prístup ku vstupnému bufferu, lebo vstupný buffer je jednoducho jeden Pythonový reťazec a nie je rozdelený po riadkoch. Druhý spôsob sa používa v chybových hláškach, a prípadne v ladiacich informáciach uložených vo výslednom abstraktnom syntaktickom strome. Hľadať chybu podľa čísla riadku je pre programátorov omnoho jednoduchšie, než podľa počtu znakov od začiatku súboru.

Kontext tiež slúži ako univerzálny objekt, kde môžu jednotlivé parsovacie funkcie ukladať, čo chcú. (Python nie je staticky typovaný jazyk a umožňuje rozširovať triedy a objekty o nové atribúty.) To sa hodí, keď chceme nejakú informáciu zdieľať naprieč viacerými volaniami vnoreného parsera. Nové atribúty v Pythone môžu mať ľubovoľné mená, ale jednotlivé parsery by mali používať rôzne predpony, aby nedochádzalo ku vzájomným konfliktom.

Napríklad ak by sme pomocou Multiparsera parsovali jazyk C, parser by potreboval tabuľku symbolov, aby vedel rozlíšiť mená typov od mená premenných a predišiel tým nejednoznačnosti v gramatike. Predstavme si, že náš program je primárne napísaný v inom jazyku, a miestami obsahuje kusy vnoreného C kódu. Každý bude spracovaný iným volaním parsovacej funkcie. Keby bola tabuľka symbolov uložená v lokálnej premennej, mená typov by sa nezachovali medzi jednotlivými volaniami. Parser miesto toho môže uložiť tabuľku symbolov priamo v parsovacom kontexte ako atribút `c_symbols`. Vďaka tomu ju bude zdieľať všetok C kód z celého vstupného súboru.

Prvým argumentom parsovacích funkcií je vždy `ctx` – parsovací kontext. Za ním nasledujú dva nepovinné argumenty, `start_nt` a `close_with`. `start_nt` umožňuje zmeniť počiatočný neterminál gramatiky – napríklad rozlišovať medzi situáciami, keď očakávame zoznamom príkazov, a keď očakávame jeden výraz. Ak je vynechaný alebo má predvolenú hodnotu `None`, použije sa predvolený počiatočný neterminál podľa nastavení gramatiky.

Hodnota argumentu `close_with` určuje zatvárací oddeľovač, ktorým sa má tento jazyk končiť. (Pripomíname zo sekcie 1.2.2, že každý jazyk môže mať viaceré páry otváracích a zatváracích oddeľovačov.) Keď jedna parsovacia funkcia volá inú, nastaví `close_with` na ten zatvárací oddeľovač, ktorý zodpovedá prečítanému otváraciemu oddeľovaču. Ak je `close_with` vynechaný alebo má predvolenú hodnotu `None`, znamená to, že tento parser je na vrchnej úrovni a nie je vnorený v inom jazyku. Preto má čítať až do konca súboru.

Argumenty `start_nt` a `close_with` sa nedajú nastaviť na ľubovoľné hodnoty, aké sa

vonkajšemu jazyku páčia. Majú väčšinou iba niekoľko povolených hodnôt, ktoré treba určiť už pri generovaní parsera. Parsovacie tabuľky LL aj LR parserov závisia od toho, aké počiatočné neterminály a zatváracie oddeľovače má podporovať.

2.2 Rozhranie tokenizérov

Tokenizéry sú väčšinou špecifické pre konkrétny jazyk, a sú volané iba zvnútra konkrétneho parsera. V tom zmysle sú takmer len implementačný detail. Keby sme v Multiparseri niektorý jazyk parsovali ručným ad-hoc parserom, jeho tokenizér by mohol vyzerať úplne inak, ako ostatné. Ale štandardné LL a LR parsery generované Multiparserom očakávajú od tokenizéra konkrétne rozhranie.

Úlohou tokenizérov je čítať vstupné znaky a produkovať prúd tokenov. Preto sú implementované ako Pythonové iterátory. Vo všeobecnosti je iterátor objekt, ktorý opakovane produkuje hodnoty. Zvyčajne sa používa na prechod nejakou dátovou štruktúrou (poľom, slovníkom alebo podobne), ale Python má aj vstavané iterátory na menovanie celých čísel od *a* po *b*, všetkých permutácii nejakej postupnosti, a podobne. Čokoľvek, čo postupne produkuje sériu hodnôt, môže byť iterátor.

Iterátory používajú nasledovný dohodnutý protokol. Aby bol objekt iterátorom, musí mať metódu `__iter__()`, ktorá vracia jeho samého, a metódu `__next__()`, ktorá vracia „ďalší prvok“, alebo vyhodí výnimku `StopIteration`, ak už žiaden ďalší prvok neexistuje. Každý objekt, čo spĺňa tieto podmienky, je z pohľadu Pythonu iterátor.

Náš LL alebo LR parser dostane zvonka funkciu `tokenizer`. Tú zavolá s argumentami `ctx` a `close_with` a očakáva, že vráti samotný iterátor (objekt s metódami `__iter__` a `__next__`). Ten sa v parseri uloží do lokálnej premennej `tok_iter`. Keď potom parser potrebuje načítať ďalší token, použije štandardnú Pythonovú funkciu `next(tok_iter)`, ktorá zavolá metódu `__next__` a vráti jej výsledok.

Hodnoty, ktoré produkuje iterátor, musia byť tokeny. Každý token je reprezentovaný ako objekt s troma atribútmi: `type`, čo určuje jeho typ a vystupuje ako terminál v bezkontextovej gramatike, `value`, čo môže obsahovať konkrétnu hodnotu asociovanú s týmto tokenom, a `location`, čo je lokácia tokenu použiteľná v chybových hláškach a ladiacich informáciach. Napríklad vstupné znaky „`0xFF`“ sa môžu preložiť na token s `type` rovným „NUMBER“ a `value` rovnou číslu 255. Z pohľadu gramatiky nás nezaujíma konkrétna hodnota a všetky číselné konštanty berieme ako terminál „NUMBER“.

Atribút `value` sa používa aj pri reprezentácii vnorených programov. Ako sme uviedli v úvode kapitoly 1, celý vnorený program, od otváracieho až po zatvárací oddeľovač, sa zabalí do jedného tokenu. Tokenizér vyprodukuje jediný token, ktorého `type` bude „EMBEDSTAT“ (*embedded statements*, vnorený zoznam príkazov), alebo iná podobná konštanta, a `value`

bude obsahovať celý abstraktný syntaktický strom (resp. iný výstup) z vnoreného parsera.

2.3 Stavba tokenizérov

Multiparser nemá zabudovaný systém na automatické generovanie tokenizérov. Jednotlivé tokenizéry musia byť napísané ručne. To je rozdiel oproti parserom – tie vie Multiparser generovať podľa vstupných gramatík. Preto je dôležité, aby bolo písanie tokenizérov dostatočne jednoduché.

Jeden zo štandardných spôsobov, ako v jazyku Python implementovať iterátor, je použiť takzvané *generátory*. Iterátor je z definície ľubovoľný objekt, čo má metódy `__iter__` a `__next__`, popísané vyššie. Generátory sú jeden zo spôsobov, ako sa taký objekt dá napísať.

Generátor sa píše ako normálna Pythonová funkcia, ktorá miestami obsahuje príkaz `yield`. Akonáhle funkcia obsahuje aspoň jeden `yield`, stáva sa generátorom. Zvnútra sa generátor správa ako funkcia, ktorej vykonávanie sa niekedy môže prerušiť. Príkaz `yield` vráti volajúcemu nejakú hodnotu a pozastaví vykonávanie celej funkcie, až kým ju zase nebude treba. Zvonka sa funkcia tiež správa inak. Keď ju niekto zavolá, nespustí sa jej ozajstné telo, ale volajúci iba dostane vhodný iterátor. Pôvodné telo funkcie sa spustí až vtedy, keď sa prvýkrát zavolá metóda `__next__` tohto iterátora, a beží dovtedy, kým sa nevykoná `yield`. Každé ďalšie volanie `__next__` ju spustí odtiaľ, kde skončila, až po ďalší `yield`. Python sa teda sám na pozadí postará o protokol iterátorov a definíciu metódy `__next__`, a programátor sa môže sústrediť na samotný algoritmus.

Jednoduchý generátor by mohol vyzerať napríklad nasledovne:

```
# example_generator.py

def my_generator(should_skip_20=False):
    print("begin")
    yield 10
    if not should_skip_20:
        yield 20
    yield 30
    print("end")

my_iterator = my_generator(True)
print(next(my_iterator))       # vypíše "begin" a 10
print(next(my_iterator))       # vypíše 30
print(next(my_iterator))       # vypíše "end" a vyhodí StopIteration
```

Všimnime si, že volanie `my_generator(True)` nevypíše „begin“ – telo funkcie sa spustí až pri prvom volaní funkcie `next`, resp. metódy `__next__`.

Generátory nezväčšujú „silu jazyka“. Každý generátor by sa dal napísať aj priamo ako trieda, čo ručne implementuje metódu `__next__`. Z argumentov a lokálnych premenných by sa stali atribúty. Aj *instruction pointer* (na ktorom riadku generátora práve sme) by sa stal

atribútom. Ale už pre tento jednoduchý generátor je táto cesta výrazne zložitejšia, a to ani neobsahoval žiadne cykly:

```
# example_generator2.py

class MyGenerator:
    def __init__(self, should_skip_20=False):
        self.should_skip_20 = should_skip_20
        self.last_result = None

    def __iter__(self):
        return self

    def __next__(self):
        if self.last_result == None:
            print("begin")
            self.last_result = 10
            return 10
        if self.last_result == 10 and not self.should_skip_20:
            self.last_result = 20
            return 20
        if self.last_result == 10 or self.last_result == 20:
            self.last_result = 30
            return 30
        if self.last_result == 30:
            print("end")
            raise StopIteration()
```

Generátory umožňujú vyjadriť ten istý algoritmus bez toho, že by si explicitne pamätali aktuálny krok výpočtu. A keďže tokenizéry sú v Multiparseri jednoducho iterátory, ktoré produkujú tokeny, oplatí sa písať ich pomocou generátorov.

Jednoduchý tokenizér by mohol vyzerať nasledovne:

```
def tokenizer(ctx, close_with=None):
    while True:
        loc = ctx.location

        if ctx.current in (' ', '\n', '\t'):   # biele znaky
            ctx.advance()           # posuň sa na ďalší znak
        elif ctx.current == '':               # koniec vstupu
            yield Token(loc, 'EOF', None)
        elif ctx.current in ('+', '-', '*', '/', '^', '(', ')', ';'):
            token_type = ctx.current
            ctx.advance()
            yield Token(loc, token_type, None)
        elif ctx.match(r'[0-9]+'):
            value = ctx.matched.group()
            yield Token(loc, 'NUMBER', int(value))
        else:
            raise ParseError(loc, "Unexpected %r" % ctx.current)
```

Tento ukážkový tokenizér dokáže rozoznať čísla v desiatkovej sústave a jednoznakové tokeny `+`, `-`, `*`, `/`, `^`, `(`, `)` a `;`. Číselné tokeny budú mať `type` rovný `'NUMBER'`, a na konci vstupu bude nekonečno tokenov typu `'EOF'`.

Tokenizér môže čítať vstup pomocou premennej `ctx.current`, ktorá obsahuje ďalší

neprečítaný znak (alebo prázdny reťazec, ak je na konci vstupu), a `ctx.advance()`, ktorá sa posunie o jeden alebo viacero znakov dopredu. Metóda `ctx.peek(n)` vráti ďalších `n` znakov bez toho, že by sa posunula za ne.

Metóda `ctx.match()` poskytuje pohodlnejší spôsob čítania vstupu, než znak po znaku. Ak na aktuálnej pozícii nájde hľadaný regulárny výraz, uloží zhodu do `ctx.matched`, posunie aktuálnu pozíciu za neho a vráti `True`. Inak vráti `False`. Tokenizéry „hračkárskych“ jazykov, ktoré uvidíme v kapitole 3, rozoznávajú vstup hlavne touto metódou.

2.4 Generátor LL parserov

Teoretická stavba LL(1) parserov bola popísaná v sekcii 1.2, a rozšírenie o podporu vnorených jazykov v sekcii 1.2.2. Hlavnou úlohou generátora je načítať bezkontextovú gramatiku, vypočítať pre ňu množiny $FIRST$ a $FOLLOW$, a na ich základe zostrojiť parsovaciu tabuľku.

Konštrukciu parsera má na starosti funkcia `make_ll_parser()` zo súboru `makell.py`. Táto funkcia dostane ako vstup bezkontextovú gramatiku (inštanciu triedy `Grammar`) a vráti vygenerovaný parser v podobe reťazca Pythonového kódu.

Trieda `Grammar` obsahuje zoznam gramatických pravidiel. Každé pravidlo má ľavú stranu (neterminál), pravú stranu (n-ticu symbolov) a akciu (reťazec Pythonového kódu, na ktorý sa pozrieme neskôr). Na rozdiel od formálnej definície bezkontextových gramatík v triede `Grammar` nie je explicitná množina neterminálov – za neterminál sa považuje čokoľvek, čo sa nachádza na ľavej strane nejakého pravidla. Gramatika navyše môže mať *hlavičku* – Pythonový kód, ktorý bude umiestnený na začiatok vygenerovaného parsera – a *vlastnosti* – Pythonový slovník dodatočných nastavení, ktoré môžu ovplyvňovať vygenerovaný parser.

Dôležitou vlastnosťou je `"default_start"`, ktorá určuje počiatočný neterminál gramatiky. Ak nie je definovaná, použije sa neterminál nazvaný „`start`“. Počiatočný neterminál je možné za behu zmeniť argumentom `start_nt`, ktorý bol popísaný v sekcii 2.1.

V triede `Grammar` sú aj metódy na výpočet množín $FIRST$ a $FOLLOW$. Pripomeňme, že množina $FIRST$ je definovaná nasledovne:

$$FIRST(w) = \{\, x \in T \mid \exists u \in (N \cup T)^* : w \Rightarrow^* xu \,\}$$

Niektoré zdroje používajú inú definíciu, v ktorej navyše $\varepsilon \in FIRST(w)$, ak $w \Rightarrow^* \varepsilon$. V Multiparseri obsahuje $FIRST(w)$ iba terminály. Informáciu, ktoré neterminály a vetné formy sú vymazávajúce, vypočítame samostatne.

Množinu vymazávajúcich neterminálov budeme cyklicky konštruovať vo viacerých iteráciach. Na začiatku bude prázdna. V každej iterácii cyklu sa pozrieme na každé pravidlo $\xi \to w$, a ak sa w skladá iba z vymazávajúcich neterminálov (vzhľadom na doteraz nájdenú

množinu), pridáme medzi vymazávajúce neterminály aj ξ. Skončíme vtedy, keď dosiahneme pevný bod a ďalšia iterácia cyklu nezmení našu množinu. Indukciou od hĺbky stromu odvodenia sa dá ľahko dokázať, že tento postup nájde správne neterminály.

V ďalšom cykle podobne nájdeme pre každý neterminál ξ množinu $FIRST(\xi)$. Na začiatku budú všetky $FIRST(\xi)$ prázdne. V každej iterácii cyklu sa opäť pozrieme na každé pravidlo $\xi \to w$. Nech $w = x_1 x_2 \ldots x_n$. Pre každé $1 \leq i \leq n$ sa pozrime na symbol x_i. Ak sa $x_1 x_2 \ldots x_{i-1}$ skladá iba z vymazávajúcich neterminálov, x_i môže byť prvým symbolom z w, a preto do množiny $FIRST(\xi)$ pridáme všetky prvky z $FIRST(x_i)$. (Ak je x_i terminál, do $FIRST(\xi)$ pridáme jeho samotného.) Opäť skončíme vtedy, keď po niektorej iterácii nenastane žiadna zmena.

Tento cyklus vypočítal množiny $FIRST(\xi)$ iba pre neterminály ξ, ale ak chceme vypočítať $FIRST(w)$ pre ľubovoľnú vetnú formu w, použijeme už známy postup. Pozrieme sa na jednotlivé symboly z $w = x_1 x_2 \ldots x_n$, a spravíme zjednotenie množín $FIRST(x_i)$ pre tie x_i, ktorým predchádzajú iba samé vymazávajúce neterminály.

Ďalej treba vypočítať množiny $FOLLOW(\xi)$. Jej definícia závisí od podporovaných zatváracích oddeľovačov. Ak ich množinu nazveme E, môžeme $FOLLOW$ definovať nasledovne:

$$FOLLOW(\xi) = \{\, x \in T \mid \exists u, v \in (N \cup T)^*, e \in (E \cup \{\$\}) : \sigma e \Rightarrow^* u \xi x v \,\}$$

Pre počiatočný neterminál inicializujeme množinu $FOLLOW(\sigma)$ na $E \cup \{\$\}$. (Ak chceme vyrobiť parser s viacerými počiatočnými neterminálmi, ktoré môžu byť použité v rôznych situáciach, všetky dostanú $E \cup \{\$\}$.) Ostatné množiny $FOLLOW(\xi)$ začnú prázdne.

V každej iterácii cyklu sa znova pozrieme na každé pravidlo $\xi \to w$ a rozdelíme ho na jednotlivé symboly $w = x_1 x_2 \ldots x_n$. Ak je x_i neterminál, zvyšok w nám hovorí, čo za ním môže nasledovať. Do množiny $FOLLOW(x_i)$ preto pridáme každý prvok z $FIRST(x_{i+1} \ldots x_n)$. A navyše, ak je $x_{i+1} \ldots x_n$ vymazávajúce (skladá sa iba z vymazávajúcich neterminálov), do $FOLLOW(x_i)$ pridáme aj $FOLLOW(\xi)$. Skončíme, keď po niektorej iterácii nenastane žiadna zmena. (Tento algoritmus predpokladá, že každý neterminál je dosiahnuteľný.)

Akonáhle poznáme množiny $FIRST$ a $FOLLOW$, môžeme v súbore `makell.py` zostrojiť parsovaciu tabuľku podľa kritéria v sekcii 1.2. Pre každý neterminál ξ a terminál x nájdeme také pravidlo $\xi \to w$, kde $x \in FIRST(w)$, alebo je w vymazávajúce a $x \in FOLLOW(\xi)$. Ak sú viaceré, generátor nahlási konflikt a zlyhá. Ak nie je žiadne také pravidlo, políčko parsovacej tabuľky necháme prázdne.

V praxi sa bohužiaľ objavia aj také gramatiky, ktoré miestami vyčnievajú z LL(1). Multiparser preto umožňuje manuálne nastaviť riešenia LL konfliktov. Ak sa parser pre neterminál ξ a terminál x nedokáže rozhodnúť medzi pravidlami $\xi \to r_1$ a $\xi \to r_2$, používateľ môže gramatike pridať vlastnosť `"llconflicts"`, v ktorej bude slovník s kľúčom (ξ, x) a hodnotou r_1 resp. r_2. Praktické použitie uvidíme nižšie, a tiež na jazyku ToyLua v sekcii 3.3.

Generátor LL parserov zoberie vypočítanú tabuľku a zoradený zoznam pravidiel, zapíše ich ako Pythonové literály, a pripojí funkciu `parse()`, ktorá implementuje samotný LL(1) automat. Jej kód bude rovnaký pre každú gramatiku, rozdiely budú iba v parsovacej tabuľke.

Kostra LL automatov bola opísaná v sekcii 1.2. Ale musíme ešte niečo pridať: generovanie výstupu. Parsery nemajú zistiť len to, čo je na vstupe korektný program alebo nie. Musia byť schopné vrátiť výsledok – najčastejšie ako abstraktný syntaktický strom. Každý terminál a neterminál v strome odvodenia bude mať nejakú „hodnotu“, a výsledkom bude hodnota koreňového neterminálu. Na to slúžia „akcie“ gramatických pravidiel, ktoré sme spomínali vyššie. Akcia pravidla je funkcia, ktorá dostane parsovací kontext a hodnoty všetkých symbolov na pravej strane, a vráti hodnotu pre neterminál na ľavej strane pravidla. Preto ak sa na pravej strane nachádza n symbolov, akcia musí akceptovať $n+1$ argumentov. Ak akcie potrebujú pomocné funkcie alebo konštanty, môžu byť definované v hlavičke gramatiky, ktorá sa pripojí pred vygenerovaný parser.

Aby si LL parser mohol pamätať hodnoty, pridáme mu okrem štandardného zásobníka ešte druhý, kde budú zhromaždené doterajšie výsledky. Vždy, keď dočítame nejakú pravú stranu pravidla, zo zásobníka výsledkov odoberieme daný počet hodnôt, spustíme akciu pre dané pravidlo a jej výsledok dáme na vrch. Aby sme vedeli, kedy to treba spraviť, na hlavný zásobník pod w^R (reverz pravej strany pravidla) pridáme značku „pravidlo skončilo, použi akciu“.

Keď k automatu pridáme túto funkcionalitu, dostávame nasledovný kód:

```
def parse(ctx, start_nt=None, close_with=None):
    stack = [close_with or 'EOF', start_nt or default_start] # LL zásobník
    results = []                                # zásobník výsledkov
    tok_iter = tokenizer(ctx, close_with)       # iterátor tokenov
    token = next(tok_iter)                      # prvý token

    while True:
        stack_top = stack[-1]

        if isinstance(stack_top, int):          # je to značka pre koniec pravidla?
            rule, action = RULES[stack_top]     # pravá strana a akcia
            stack.pop()
            n = len(rule)                       # zisti dĺžku pravej strany
            args = results[len(results) - n:] # zober vrchných n hodnôt
            del results[len(results) - n:]    # odstráň ich zo zásobníka
            results.append(action(ctx, *args)) # spusti akciu, pridaj výsledok

        elif stack_top not in TABLE:            # je to terminál?
            if token.type != stack_top:         # over, že na vstupe nasleduje on
                raise ParseError(token.location, "...")
            stack.pop()
            if not stack: break                 # prázdny zásobník -> skončili sme
            results.append(token)
            token = next(tok_iter)              # neprázdny zásobník -> čítaj ďalej

        else:                                   # je to neterminál?
            row = TABLE[stack_top]
```

```
            if token.type not in row:          # hľadaj terminál, čo je na vstupe
                raise ParseError(token.location, "...")
            rule_num = row[token.type]         # v tabuľke je číslo pravidla
            rule, action = RULES[rule_num]
            stack.pop()                        # odstráň zo zásobníka neterminál
            stack.append(rule_num)             # pridaj značku pre koniec pravidla
            stack.extend(reversed(rule))       # pridaj reverz pravej strany

    return results[0]                          # vráť hodnotu pre start_nt
```

Akcie sa pre vrcholy stromu odvodenia vždy spúšťajú v postorder poradí. Obvykle iba niečo spravia s argumentami a vrátia výsledok (či už abstraktný syntaktický strom, fragment bytekódu, alebo niečo iné). Ale nič nebráni tomu, aby boli imperatívne a mali vedľajšie účinky. To by sme mohli využiť napríklad pri implementácii tabuľky symbolov pre jazyk C, ako sme naznačili v sekcii 2.1.

Všimnime si, že keď zo zásobníka odstránime posledný prvok, z cyklu okamžite vyskočíme a nečítame žiaden ďalší token. To je nutné kvôli vnáraniu do iných jazykov – viď sekciu 1.2.2. Klasický automat by miesto toho mohol jednoducho použiť podmienku „`while len(stack) != 0`“.

Funkcionalitu generátora LL parserov, ako aj gramatické vlastnosti `"default_start"` a `"llconflicts"`, uvidíme na tomto malom príklade:

```
# example_ll.py

from grammar import Grammar
from makell import make_ll_parser

preface = "from toycalc.lex import tokenizer"

rules = {
    # stat -> HELLO | IF cond THEN stat optional_else
    "stat": [(("HELLO",), "lambda ctx, a: 'hello'"),
             (("IF", "cond", "THEN", "stat", "optional_else"),
              "lambda ctx, a, b, c, d, e: ('if', b, d, e)")],

    # cond -> TRUE | FALSE
    "cond": [(("TRUE",), "lambda ctx, a: True"),
             (("FALSE",), "lambda ctx, a: False")],

    # optional_else -> ELSE stat | epsilón
    "optional_else": [(("ELSE", "stat"), "lambda ctx, a, b: b"),
                      ((), "lambda ctx: None")],
}

properties = {
    "default_start": "stat",
    "llconflicts": {
        ("optional_else", "ELSE"): ("ELSE", "stat"),
    },
}

my_grammar = Grammar(preface, properties, rules)
print(make_ll_parser(my_grammar))
```

Tento príklad ukazuje, ako je možné manuálne zostrojiť objekt `Grammar` (v sekcii 2.6 uvidíme pohodlnejší spôsob). Hlavička gramatiky musí importovať alebo definovať funkciu `tokenizer`, ktorú potom zavolá vygenerovaná funkcia `parse` (viď vyššie). Tu používame `tokenizer` z modulu `toycalc.lex`, ktorý je založený na ukážkovom tokenizéri zo sekcie 2.3, a navyše každú skupinu písmen pokladá za terminál – vstup „`HELLO WORLD`“ rozdelí na token `HELLO` a token `WORLD`, napriek tomu, že v gramatike sa nikde nenachádza. Tokenizéry ozajstných jazykov používajú zoznamy rezervovaných kľúčových slov, ale v tomto príklade nám stačí `toycalc.lex`.

Pravidlá sú zoskupené podľa neterminálu. Pravá strana musí byť vždy n-tica, aj keď má len jediný prvok. Preto nestačí `"HELLO"`, treba použiť `("HELLO",)`. Akcia každého pravidla je uložená ako reťazec kódu, ktorého výsledkom je anonymná funkcia danej arity – okrem objektu `ParserContext` dostane jeden argument za každý symbol na pravej strane.

Táto gramatika je nejednoznačná. Vstupu „IF TRUE THEN IF FALSE THEN HELLO ELSE HELLO“ zodpovedajú viaceré stromy odvodenia – nie je jasné, či „ELSE“ patrí ku vonkajšiemu alebo vnútornému „IF“-u. Túto nejednoznačnosť by sme mohli vyriešiť zmenou gramatických pravidiel, ale tu na nej demonštrujeme použitie vlastnosti `"llconflicts"`.

Konflikt nastáva preto, že terminál „ELSE“ je aj v množine $FIRST$(ELSE stat), aj vo $FOLLOW$(optional_else). Keď parser číta neterminál optional_else a na vstupe nasleduje „ELSE“, nevie sa rozhodnúť, ktoré pravidlo použiť. Hodnota `"llconflicts"` vyššie hovorí, že v tomto prípade si má vybrať „optional_else $\rightarrow$ ELSE stat“. To spôsobí, že „ELSE“ sa vždy bude viazať ku najbližšiemu „IF“-u. Keby sme `"llconflicts"` vynechali, funkcia `make_ll_parser()` by zlyhala. Chybové hlásenie by hovorilo, kde nastal konflikt a medzi akými pravidlami sa treba rozhodnúť.

Podotknime, že ak by sme chceli opačný výsledok, museli by sme už zmeniť pravidlá gramatiky. Riešenie konfliktu `("optional_else", "ELSE"): ()` by nespôsobilo, že „ELSE“ sa bude viazať ku vonkajšiemu „IF“-u, ale len to, že pravidlo „optional_else $\rightarrow$ ELSE stat“ sa nepoužije nikdy.

Keď výstup skriptu vyššie uložíme do súboru `demo_out.py`, môžeme otestovať, že vzniknutý LL parser plní naše očakávania:

```
# example_lltest.py

from demo_out import parse
from context import ParserContext

print(parse(ParserContext("HELLO")))
# vypíše hello

print(parse(ParserContext("IF FALSE THEN HELLO")))
# vypíše ('if', False, 'hello', None)
# každý IF sa preloží na štvoricu ('if', podmienka, then-vetva, else-vetva)

print(parse(ParserContext("IF TRUE THEN IF FALSE THEN HELLO ELSE HELLO")))
```

```
# vypíše ('if', True, ('if', False, 'hello', 'hello'), None)
# čiže ELSE patrí ku vnútornému IF-u, vonkajší IF nemá žiadnu ELSE vetvu

print(parse(ParserContext("IF HELLO THEN TRUE")))
# vyhodí ParseError: "Unexpected 'HELLO', expected 'FALSE', 'TRUE'"
```

2.5 Generátor LR parserov

Zatiaľčo sekcia 1.3 predstavila teoretické základy LR parserov, tu si ukážeme postup ich štandardnej konštrukcie. Túto konštrukciu nemusíme príliš meniť – na podporu vnárania do iných jazykov stačí pridať možnosť začínať z rôznych začiatočných stavov podľa toho, ktorý zatvárací oddeľovač chceme.

Náš NKA pre LR(1) má stavy tvaru $(\xi \to u \bullet v, w)$. Aby sme ich odlíšili od DKA stavov, budeme ich volať *LR položky*. DKA stav je teda určený množinou svojich LR položiek.

Pri konverzii NKA na DKA musíme odstrániť ε-prechody. NKA má podľa sekcie 1.3 tieto: z $(\xi \to u \bullet \gamma v', w)$ do $(\gamma \to \bullet p, w')$ vedie ε-prechod práve vtedy, keď existuje pravidlo $\gamma \to p$ a w' je taký k-terminálový reťazec (teda pre LR(1) jeden terminál), ktorý môže nasledovať po γ, t.j. $w' \in FIRST(v'w)$. Množiny $FIRST$ vypočítame ako pri LL parseroch (sekcia 2.4).

Pre množinu položiek K_0 (ktorú budeme volať *jadro*) teda môžeme vypočítať všetky položky, do ktorých sa vieme na ε dostať z niektorej položky z K_0 (tzv. *uzáver* alebo *epsilonový chvost* množiny K_0).

Samotný prevod NKA na DKA potom spravíme prehľadávaním grafu DKA stavov, napríklad do šírky. Počas prehľadávania si budeme pamätať frontu stavov, ktoré sme ešte nenavštívili. Keď pri prehľadávaní navštívime ďalší stav, zistíme stavy, do ktorých sa z neho môžeme dostať, nájdeme ich uzávery, a ak sme ich ešte nevideli, pridáme ich do fronty. Každý stav je určený množinou položiek, ale aby sa nám s nimi jednoduchšie pracovalo, počas prehľadávania im v nejakom poradí priradíme čísla.

Pri prehľadávaní budeme robiť tabuľku DKA prechodov, alebo presnejšie, tabuľku akcií. Ak na vstupe vidíme terminál alebo neterminál x, a náš stav S obsahuje jednu alebo viacero položiek tvaru $(\xi \to u \bullet xv, w) \in S$, DKA má prejsť do stavu $closure(\{(\xi \to ux \bullet v, w) \mid (\xi \to u \bullet xv, w) \in S\})$ (t.j. $\bullet x$ sa zmení na $x\bullet$ a ostatné položky zmiznú) a posunúť sa na vstupe o symbol ďalej. Posunutie na vstupe voláme *shift*. (Ak sme tento stav ešte nevideli, dáme mu nové číslo a pridáme ho do fronty.) Ak sme v stave S, pričom $(\xi \to u\bullet, w) \in S$ (sme na konci pravidla $\xi \to u$), a na vstupe vidíme písmeno $x = w$, našli sme handle. NKA by v tomto momente prešiel do konečného stavu. My vtedy chceme „vyškrtnúť u zo vstupu, a dať tam ξ“. Táto akcia sa volá *reduce* podľa pravidla $\xi \to u$. V tabuľke akcií teda bude pre každý stav S a každý symbol x buď *shift* do iného stavu, alebo *reduce* podľa nejakého pravidla, alebo nič (ak v tomto stave nie je daný symbol povolený).

Parser končí vtedy, keď by redukoval špeciálnu položku $(\sigma\bullet, w)$. Túto akciu voláme *accept*. Naša implementácia generátora ale pre jednoduchosť dáva do tabuľky štandardný *reduce*, pretože špeciálne položky bez ľavej strany sú uložené, ako keby mala ľavá strana hodnotu `None`. Parser vie, že keď by mal redukovať `None` $\rightarrow \sigma$, môže skončiť.

Prehľadávanie začneme od všetkých stavov, v ktorých môže automat začať. Podľa sekcie 1.3.1, ak chceme čítať po terminál e (kde e môže byť buď zatvárací oddeľovač, alebo ozajstný koniec súboru „$“ resp. `'EOF'`), NKA má začať v stave $(\bullet\sigma, e)$, takže DKA má začať od $closure(\{(\bullet\sigma, e)\})$.

Pozrime sa teraz na implementáciu samotného automatu. Na konci sekcie 1.3 sme naznačili, že automat si bude pamätať zásobník doteraz prečítaných terminálov a stavov dosiahnutých počas čítania. Ale v skutočnosti nám stačí pamätať si jednotlivé stavy. Podľa terminálov, čo sme už prečítali, sa nikdy nerozhodujeme.

Na vrchu zásobníka je aktuálny stav. Automat podľa aktuálneho stavu S a nasledujúceho terminálu na vstupe x zistí, akú akciu má spraviť. Ak *shift* do stavu S', automat jednoducho pridá S' na zásobník (tým sa stane aktuálnym stavom) a prejde za terminál x (do premennej x načíta nový „ďalší neprečítaný terminál“). Ak *reduce* podľa pravidla $\xi \rightarrow u$, automat musí akoby vyškrtnúť zo vstupu znaky u, ktoré práve prečítal, a pridať tam ξ. Automat sa preto vráti o $|u|$ stavov dozadu (vyhodí vrchných $|u|$ stavov zo zásobníka), čím akoby vyškrtne u. Na vrchu zásobníka bude stav S'. Automat podľa stavu S' a neterminálu ξ nájde v tabuľke novú akciu, ktorú vykoná, akoby bol na vstupe ξ. Všimnime si, že to bude vždy *shift*. Zo zásobníku teda odbudne $|u|$ stavov a pribudne jeden nový stav.

V Multiparseri je tabuľka akcií definovaná pre terminály aj neterminály. Ale akciu, keď je „na vstupe“ neterminál, hľadáme iba počas spracovania akcie *reduce*. Klasické formulácie LR automatov preto zvyknú tabuľku akcií rozdeľovať na dve: tabuľka *action* určuje podľa stavu a nasledujúceho terminálu, čo máme robiť, a tabuľka *goto* určuje podľa stavu a práve zredukovaného neterminálu, aký nový stav pridať po zredukovaní na zásobník. Multiparser ich nerozdeľuje.

Podobne ako pri LL parseroch musíme ešte pridať možnosť počítať výsledky. Pre každý vrchol v strome odvodenia opäť vypočítame jeho „hodnotu“ – pre terminály to bude priamo daný token, a pre neterminály bude každé pravidlo $\xi \rightarrow u$ mať „akciu“ – funkciu, čo z kontextu a hodnôt synov vypočíta hodnotu pre vrchol ξ. (Akcia pravidla nie je to isté, ako akcia v LR automate.) Tieto hodnoty si budeme rovnako ako v LL parseroch pamätať na zásobníku výsledkov. Tentoraz bude priamočiaro odrážať zásobník stavov, až na to, že bude o prvok kratší (na začiatku bude prázdny, zatiaľčo zásobník stavov má na začiatku jeden prvok). Keď automat robí *shift*, na zásobník výsledkov jednoducho pridá aktuálny token, a keď robí *reduce*, zoberie vrchných $|u|$ hodnôt, zavolá akciu pravidla a na zásobník dá výsledok. Odbudne $|u|$ hodnôt a pribudne jedna, rovnako ako pre zásobník stavov.

Generovanie kódu prebieha rovnako, ako pri LL parseroch. Funkcia `make_lr_parser()` (súbor `makelr.py`) dostane objekt `Grammar`, vypočíta parsovaciu tabuľku a pripojí k nej funkciu `parse()`, v ktorej je implementovaný samotný parsovací automat. V prípade LR parserov vyzerá nasledovne:

```
def parse(ctx, start_nt=None, close_with=None):
    stack = [INITIAL[(start_nt or default_start,
                      close_with or 'EOF')]]  # zásobník LR stavov
    results = []                              # zásobník výsledkov
    tok_iter = tokenizer(ctx, close_with)     # iterátor tokenov
    token = next(tok_iter)                    # prvý token

    while True:
        # stack[-1] je vrchný stav, token.type je nasledujúci terminál
        action = TABLE[stack[-1]].get(token.type)

        if action is None:
            raise ParseError(token.location, "...")

        is_shift, action_value = action       # action je vždy dvojica

        if is_shift:                          # shifty sú (True, nový stav)
            results.append(token)
            stack.append(action_value)        # pridaj na zásobník nový stav
            token = next(tok_iter)            # načítaj ďalší token

        else:                                 # redukcie sú (False, ID pravidla)
            if action_value is None: break    # počiatočná položka -> akceptuj
            left, right, rule_action = RULES[action_value]
            n = len(right)

            args = results[len(results) - n:] # zober vrchných n hodnôt
            del results[len(results) - n:]    # odstráň ich zo zásobníka
            results.append(
                rule_action(ctx, *args))      # spusti akciu, pridaj výsledok

            del stack[len(stack) - n:]        # odstráň vrchných n stavov
            new_is_shift, new_state = TABLE[stack[-1]][left] # "prečítaj" left
            assert new_is_shift               # nová akcia je určite SHIFT
            stack.append(new_state)           # daj na zásobník nový stav

    return results[0]                         # vráť hodnotu pre sigmu
```

LR parser nemôže začínať od ľubovoľného neterminálu. Generátor musí pre každú povolenú dvojicu (počiatočný neterminál, zatvárací oddeľovač) vytvoriť iný počiatočný LR stav. Množina povolených dvojíc je nastavená v objekte `Grammar`.

2.5.1 Konflikty a priorita operátorov

Podobne, ako sme videli v sekcii 2.4, aj generátor LR parserov má prostriedky na riešenie konfliktov. Návrh systému je z veľkej časti prebratý z generátora parserov Bison [4]. Multiparser ho rozširuje o podporu viacerých precedenčných skupín a možnosť riešenia reduce-reduce konfliktov (viď nižšie).

V LR tabuľke sa pre daný stav a nasledujúci terminál môže nachádzať buď akcia *shift*, alebo akcia *reduce* podľa niektorého pravidla. Konflikt nastáva vtedy, keď má generátor viacero akcií, čo môže spraviť. Ak sa nevie rozhodnúť medzi shift a reduce, nazýva sa *shift-reduce konflikt*. Ak má na výber viaceré pravidlá, podľa ktorých redukovať, je to *reduce-reduce konflikt*.

Multiparser sa oddelene pozrie na každé pole tabuľky (teda konkrétny LR stav a nasledujúci terminál), a ak by malo viacero akcií, pokúsi sa tento konflikt vyriešiť. Multiparser nerobí žiadne implicitné rozhodnutia – každé riešenie konfliktu musí byť špecifikované vo vlastnostiach gramatiky.

Riešenie shift-reduce konfliktov je založené na koncepte priorít operátorov. Uvažujme gramatiku $G = \{\alpha \to x \mid \alpha + \alpha \mid \alpha * \alpha\}$ a vstup $x + x * x$. Chceli by sme dostať odvodenie $x + (x * x)$, ale gramatika G je nejednoznačná. Po prečítaní $x + x$ budeme v LR stave, ktorý obsahuje položku $(\alpha \to \alpha + \alpha\bullet, *)$ (zodpovedá odvodeniu $(x + x\bullet) * x$) aj položku $(\alpha \to \alpha \bullet * \alpha, \$)$ (zodpovedá odvodeniu $x + (x \bullet * x)$), takže nastáva shift-reduce konflikt. V tomto prípade chceme, aby parser spravil shift. Na vstupe $x * x + x$ by sme zase chceli, aby po prečítaní $x * x$ spravil reduce.

To nás vedie k nasledovnému algoritmu: gramatike umožníme vyjadriť „priority“ – čiastočné usporiadanie terminálov a pravidiel. Ak nastane konflikt medzi shiftnutím terminálu x a redukovaním pravidla $\xi \to w$, skúsime porovnať ich priority. Ak má väčšiu prioritu terminál, zabudneme na reduce a do parsovacej tabuľky zapíšeme shift. Ak má väčšiu prioritu pravidlo, zabudneme na shift a použijeme reduce.

Ak majú shiftovaný terminál aj redukované pravidlo rovnakú prioritu, umožníme nastaviť asociativitu. Pri ľavo asociatívnych operátoroch, ako je $+$, chceme $x + x + x$ prečítať ako $(x + x) + x$. Pri pravo asociatívnych operátoroch, ako napríklad umocňovanie, má $x \wedge x \wedge x$ znamenať $x \wedge (x \wedge x)$. Takže pravidlá budú nasledovné: Ak máme na výber medzi shiftom a redukciou rovnakej priority, a operátor používa ľavú asociativitu, použijeme reduce. Ak pravú asociativitu, použijeme shift. Ak má nastavené, že nie je asociatívny, z parsovacej tabuľky odstránime aj shift, aj reduce – použiť daný operátor viackrát za sebou bude syntaktická chyba.

V príklade vyššie teda terminálu $*$ dáme väčšiu prioritu, ako pravidlu $\alpha \to \alpha + \alpha$, a pravidlu $\alpha \to \alpha * \alpha$ väčšiu prioritu, ako terminálu $+$. Algoritmus to považuje za dva samostatné vzťahy, ale my väčšinou nechceme rozlišovať medzi „prioritou terminálu $+$“ a „prioritou pravidla $\alpha \to \alpha + \alpha$“. Preto, ak nemá pravidlo nastavenú vlastnú prioritu, bude jeho priorita rovná priorite posledného terminálu v ňom. Vďaka tomu nám v našom príklade stačí povedať, že $*$ má väčšiu prioritu, ako $+$, a prioritu pravidiel nemusíme zdôrazňovať.

V Multiparseri sa priority nastavujú pomocou gramatickej vlastnosti `"precedence"`, ktorá obsahuje zoznam precedenčných skupín. Každá precedenčná skupina určuje vzájomnú

prioritu medzi všetkými jej členmi. Ak by sme chceli, aby malo $+$ väčšiu prioritu, ako $*$, a $-$ väčšiu prioritu, ako $/$, ale navzájom sa neovplyvňovali, môžeme ich oddeliť do dvoch skupín.

Každá precedenčná skupina sa skladá z niekoľkých úrovní, zoradených od najnižšej k najvyššej priorite. Úroveň je reprezentovaná ako dvojica, ktorej prvý člen je typ asociativity (`'left'`, `'right'`, `'nonassoc'` alebo `'precedence'`), a druhý člen je zoznam položiek – terminálov alebo pravidiel. Pravidlo je reprezentované dvojicou neterminálu na ľavej strane a n-tice symbolov na pravej strane. Všetky položky na jednej úrovni majú rovnakú prioritu.

Typ `'nonassoc'` znamená, že operátor nie je možné použiť dvakrát za sebou – v parsovacej tabuľke nebude ani shift, ani reduce. Autor gramatiky vie o tom, že pravidlá gramatiky to nezakazujú, ale chce, aby také vstupy parser neakceptoval. Typ `'precedence'` znamená, že autor gramatiky určil iba prioritu, nie asociativitu. Autor gramatiky tvrdí, že priorita stačí na riešenie konfliktu – a ak napriek tomu dochádza k nejednoznačnosti, chce o tom vedieť.

Tento systém sa hodí nielen na priority operátorov, ale aj riešenie ľubovoľných shift-reduce konfliktov. Ak chceme rozhodnúť konflikt medzi terminálom x a pravidlom p, ale neovplyvniť pritom žiadne iné konflikty, môžeme pridať dvojčlennú precedenčnú skupinu, ktorá obsahuje iba terminál x a pravidlo p. To uvidíme na nasledovnom príklade. Vráťme sa ku „IF-THEN-ELSE“ parseru zo sekcie 2.4 a upravme ho na LR parser:

```
# example_lr.py

from grammar import Grammar
from makelr import make_lr_parser

preface = "from toycalc.lex import tokenizer"

rules = {
    # stat -> HELLO | IF cond THEN stat | IF cond THEN stat ELSE stat
    "stat": [(("HELLO",), "lambda ctx, a: 'hello'"),
             (("IF", "cond", "THEN", "stat"),
              "lambda ctx, a, b, c, d: ('if', b, d, None)"),
             (("IF", "cond", "THEN", "stat", "ELSE", "stat"),
              "lambda ctx, a, b, c, d, e, f: ('if', b, d, f)")],

    # cond -> cond AND cond | cond OR cond | TRUE | FALSE
    "cond": [(("cond", "AND", "cond"), "lambda ctx, a, b, c: (a, 'and', c)"),
             (("cond", "OR", "cond"), "lambda ctx, a, b, c: (a, 'or', c)"),
             (("TRUE",), "lambda ctx, a: True"),
             (("FALSE",), "lambda ctx, a: False")],
}

properties = {
    "default_start": "stat",
    "precedence": [
        # Prvá skupina: AND má vyššiu prioritu, ako OR.
        [
            ("left", ["OR"]),
            ("left", ["AND"]),
        ],
        # Druhá skupina: shiftnúť ELSE má vyššiu prioritu, ako redukovať
        # pravidlo "stat -> IF cond THEN stat". (Definovať asociativitu
```

```
        # nemusíme, a nechceme omylom maskovať prípadné ďalšie konflikty.)
        [
            ("precedence", [("stat", ("IF", "cond", "THEN", "stat"))]),
            ("precedence", ["ELSE"]),
        ],
    ],
}

my_grammar = Grammar(preface, properties, rules)
print(make_lr_parser(my_grammar))
```

Namiesto `[("stat", ("IF", "cond", "THEN", "stat"))]` by sme mohli napísať iba `["THEN"]` – pravidlá majú implicitne takú prioritu, ako má posledný terminál v nich.

Dostávame opäť parser, ktorý na vstupe „IF TRUE THEN IF FALSE THEN HELLO ELSE HELLO“ viaže „ELSE“ ku najbližšiemu „IF“-u. Ale keby sme priority vymenili, zmenili by sme výsledok shift-reduce konfliktu a dosiahli by sme opačné správanie. V LL parseri to nebolo triviálne dosiahnuť.

Mechanizmus priorít sa používa iba na riešenie shift-reduce konfliktov. Reduce-reduce konflikty totiž často svedčia o hlbšom probléme gramatiky, a lepšie je vyriešiť ich zmenou pravidiel. Ak ich autor gramatiky napriek tomu chce obísť, musí úmyselne použiť samostatnú gramatickú vlastnosť `"ignore_reduces"`. Ako už napovedá názov, ide o množinu redukcií, ktoré sa v prípade reduce-reduce konfliktov budú ignorovať. Ak je v množine trojica (a, b, c), LR položka $(a \to b\bullet, c)$ „prehrá“ všetky reduce-reduce konflikty, v ktorých sa zúčastní. Reduce-reduce konflikt je vyriešený práve vtedy, keď má najviac jedného víťaza. Praktické použitie uvidíme v sekcii 3.3.

2.6 Formát bezkontextových gramatík

Generátory LL a LR parserov pracujú na inštanciách triedy `Grammar`. V predošlých sekciách sme videli príklady, ktoré gramatiku konštruovali ručne a zoznam pravidiel reprezentovali ako Pythonový literál. Ale lepšie by bolo písať gramatiky vo formáte, ktorý je na to určený.

Multiparser používa vlastný formát nazvaný **mgr** („**m**ultiparser **gr**ammar“). Menná konvencia je, že gramatiky sú v súboroch `*.mgr`, a vygenerované Pythonové parsery sú v súboroch `*_out.py`.

Súbor **mgr** obsahuje zaradom hlavičku, vlastnosti a pravidlá gramatiky. Hlavička začína od prvého riadku súboru (aby vo výslednom parseri sedeli čísla riadkov) a končí riadkom, ktorý obsahuje iba text „`---`“. Po hlavičke nepovinne nasledujú vlastnosti gramatiky, zapísané ako Pythonový slovník uzavretý v spätných úvodzovkách (backtickoch). Ďalej nasleduje samotný zoznam pravidiel a ich akcií.

Základnú syntax ukazuje nasledujúci príklad:

```
# example_basicrules.mgr
```

```
from toycalc.lex import tokenizer
---
‘{ ’default_start’: ’program’ }‘

program = statement ";" program ‘lambda ctx, a, b, c: None‘
program =                       ‘lambda ctx: None‘

statement = sum                 ‘lambda ctx, e: print(e)‘

sum = sum "+" product           ‘lambda ctx, a, op, b: a + b‘
sum = sum "-" product           ‘lambda ctx, a, op, b: a - b‘
sum = product                   ‘lambda ctx, e: e‘

product = product "*" factor    ‘lambda ctx, a, op, b: a * b‘
product = product "/" factor    ‘lambda ctx, a, op, b: a // b‘
product = factor                ‘lambda ctx, e: e‘

factor = "-" factor             ‘lambda ctx, op, e: -e‘
factor = power                  ‘lambda ctx, e: e‘

power = value "^" factor        ‘lambda ctx, a, op, b: a ** b‘
power = value                   ‘lambda ctx, e: e‘

value = NUMBER                  ‘lambda ctx, t: t.value‘
value = "(" sum ")"             ‘lambda ctx, l, e, r: e‘
```

Symboly ako `NUMBER`, ktoré neobsahujú špeciálne znaky, môžeme písať bez úvodzoviek. Ako sme spomínali v sekcii 2.4, Multiparser pokladá všetko, čo sa vyskytuje na ľavej strane nejakého pravidla, za neterminál, a všetko ostatné za terminál.

Za každým pravidlom je v spätných úvodzovkách (backtickoch) Pythonový výraz pre akciu tohto pravidla. Akcia môže byť definovaná ako nepomenovaná lambda funkcia, alebo ju môžeme definovať v hlavičke a do backtickov napísať iba jej meno. Text v backtickoch pre jednoduchosť nemôže obsahovať ďalší backtick.

Tento príklad implementuje LR(1) gramatiku pre jednoduchú kalkulačku. Jej akcie nekonštruujú abstraktný syntaktický strom, ale hneď počítajú výsledky. Výpočet začína od pravidla `value = NUMBER`, ktorého akcia dostane token typu `NUMBER` (teda objekt `Token` s atribútmi `type`, `location` a `value`, viď sekcia 2.2), a zoberie z neho len jeho `value`. Výpočet pokračuje, až kým nepríde k neterminálu `statement`. Jeho akcia je imperatívna – vypíše hodnotu na výstup a vráti `None`. Akcie pre `program` už nemusia nič robiť. Parser postupne vyhodnotí akcie pre celý strom, aj keď ich rodič odignoruje.

Gramatické pravidlá sa dajú písať aj iným spôsobom, ktorý nazývame „rozšírený zápis“. Ten sa ľahšie píše, ale je to len syntaktický cukor – výsledkom budú stále len pravidlá a akcie, ktoré by sme mohli zapísať aj priamo.

Rozšírený zápis sa namiesto „=“ začína s „:“, aby sa dal odlíšiť od štandardného zápisu. Za „:“ môžu ísť viaceré pravidlá a akcie oddelené s „|“. Napísať `nt: a ‘...‘ | b ‘...‘` má ten istý význam, ako `nt: a ‘...‘` a vzápätí `nt: b ‘...‘`.

Pravidlá v rozšírenom zápise môžu obsahovať podpravidlá – výrazy v zátvorkách. Podpravidlo tvaru `(a|b|c)` znamená „čokoľvek z `a`, `b` alebo `c`“. `(a|b|c)?` znamená „nula alebo jeden raz“ a `(a|b|c)*` je „nula alebo viac ráz“. Tieto podpravidlá nie sú rovnocenné s normálnymi pravidlami, lebo možnostiam v zátvorkách nedávame akcie.

Podpravidlá sú len syntaktický cukor pre nový neterminál, ktorý má pravidlá `a`, `b` a `c`. Napríklad `alfa: a b (c|d (e)?) ‘...‘` sa preloží na pravidlá $\{\alpha \to ab\alpha_1, \alpha_1 \to c \mid d\alpha_2, \alpha_2 \to e \mid \varepsilon\}$ (ich akcie upresníme neskôr). Všetky tieto pravidlá by sme vedeli zapísať aj štandardným zápisom, ale podpravidlá sú často prehľadnejšie.

Na neterminály α_i sa môžeme odvolávať zápisom „`\i`“, kde `i` je číslo podpravidla (podľa otváracích zátvoriek). Rozšírený zápis `alfa: (a|b) \1 \1 ‘...‘` sa teda preloží na $\{\alpha \to \alpha_1\alpha_1\alpha_1, \alpha_1 \to a \mid b\}$. Všimnime si, že tento zápis nemá rovnakú sémantiku, ako spätné referencie v regulárnych výrazoch – regulárny výraz `(a|b)\1\1` popisuje iba slová `aaa` a `bbb`, zatiaľčo náš neterminál α popisuje celý jazyk $\{a,b\}^3$. Praktické využitie tejto syntaxe uvidíme v jazyku ToyPython v sekcii 3.2.

Ďalšou výhodou rozšíreného zápisu je jednoduchšie definovanie akcií. V štandardnom zápise musíme dávať pozor, aby pravá strana pravidla zodpovedala argumentom akcie. Na pohľad nemusí byť jasné, ktorý argument zodpovedá ktorému symbolu. V rozšírenom zápise sa mená premenných píšu priamo ku symbolom pravidla. Ak namiesto terminálu alebo neterminálu `foo` napíšeme `x=foo`, akcia uvidí jeho hodnotu v premennej `x`. Do backtickov sa už nepíše anonymná funkcia (resp. výraz, ktorého výsledkom je funkcia), ale iba jej telo. Takže pravidlo `sucet = sucet "+" sucin ‘lambda a, op, b: a + b‘` môžeme zapísať ako `sucet: a=sucet "+" b=sucin ‘a + b‘`. Tak hneď vidno, že `a` bude `sucet`, `b` bude `sucin`, a token `"+"` nás nezaujíma.

Toto „priraďovanie do premenných“ funguje aj v podpravidlách. Ak parser použije podpravidlo obsahujúce `x=foo`, akcia pravidla uvidí hodnotu symbolu `foo` v premennej `x`. Napríklad v pravidle `alfa: b (x=c | x=d) ‘spracuj(x)‘` dostane funkcia „spracuj“ hodnotu zistenú buď z `c`, alebo z `d`.

Keďže podpravidlá umožňujú opakovanie a vynechávanie, môže sa stať, že parser nájde viaceré „priradenia do `x`“. Vtedy akcia dostane tú poslednú hodnotu. Ak naopak nebudú žiadne, `x` bude mať hodnotu `None`. Napríklad akcia pravidla `alfa: (x=foo)* ‘x‘` jednoducho vráti hodnotu posledného `foo`, alebo `None`, ak parser žiadne `foo` nenašiel.

Keď akciu zaujímajú všetky hodnoty, môže použiť „zoznamovú premennú“. Ak miesto `x=foo` napíšeme `x+=foo`, akcia dostane v premennej `x` pole všetkých takto označených hodnôt. Napríklad ak chceme popísať čiarkami oddelený zoznam identifikátorov, môžeme napísať `ids: l+=ID ("," l+=ID)* ‘[tok.value for tok in l]‘`. Všetky tokeny `ID`, ten prvý aj tie z opakovaného podpravidla, budú zozbierané v poli `l`. Akcia pravidla už len prečíta `l` a z každého tokenu vyberie jeho `value`. V rámci pravidla musí každá premenná byť

buď normálna, alebo zoznamová – nemôžeme použiť aj `x=foo` aj `x+=foo`.

Akcie v rozšírenom zápise navyše môžu používať tri špeciálne premenné. `_ctx` je parsovací kontext. `_all` je zoznam hodnôt všetkých symbolov, nielen tých, čo majú „x=“. `_loc` je hodnota atribútu `location` prvého `Token`-u v `_all` (alebo `None`, ak v `_all` žiaden nie je). Lokácia prvého tokenu sa môže hodiť, ak vyrábame abstraktný syntaktický strom, v ktorom chceme uchovať ladiace informácie.

Pomocou rozšíreného zápisu môžeme predošlý príklad prerobiť na LL(1) gramatiku. Naša gramatika bohužiaľ často používa ľavú rekurziu, a jeden neterminál má viaceré pravidlá so spoločným prefixom – oboje v LL parseroch spôsobuje konflikty. Miesto toho použijeme podpravidlá. Napríklad reťaz sčítancov popíšeme ako `sum: product ("+" product)*`. Keď pridáme aj odčítanie, násobenie a delenie, dostávame nasledovný **mgr** súbor:

```
# example_extrules.mgr

from toycalc.lex import tokenizer

handlers = {
    "+": lambda l, r: l + r,
    "-": lambda l, r: l - r,
    "*": lambda l, r: l * r,
    "/": lambda l, r: l // r,
}

# Pomocná funkcia - zaradom na "l" aplikuje operácie z "ops" s hodnotami z "rs"
def left_associative(l, ops, rs):
    for op, r in zip(ops, rs):   # čiže (ops[0],rs[0]), (ops[1],rs[1]), ...
        handler = handlers[op.type]
        l = handler(l, r)
    return l
---
‘{ 'default_start': 'program' }‘

program: (statement ";")* ‘None‘

statement: e=sum ‘print(e)‘

sum: l=product ((ops+="+" | ops+="-") rs+=product)* ‘left_associative(l, ops, rs)‘

product: l=factor ((ops+="*" | ops+="/") rs+=factor)* ‘left_associative(l, ops, rs)‘

factor: "-" e=factor ‘-e‘
  | e=power ‘e‘

power: a=value ("^" b=factor)? ‘a if b is None else a ** b‘

value: t=NUMBER ‘t.value‘
  | "(" e=sum ")" ‘e‘
```

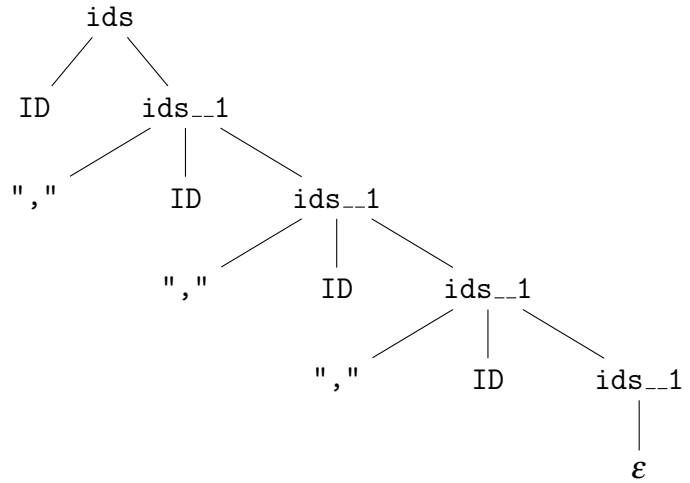
Funkciu `left_associative` stačí definovať raz, a môžeme ju použiť pre všetky ľavo asociatívne operátory. Tiež si môžeme všimnúť, ako pravidlo pre `power` využíva, že premenné majú predvolenú hodnotu `None`.

Zostáva ukázať, ako rozšírený zápis funguje a ako sa dá previesť na inštanciu triedy `Grammar` – teda vlastne na štandardný zápis. Každé pravidlo musí mať pravú stranu pozostávajúcu iba z niekoľkých terminálov alebo neterminálov, a musí mať akciu – Pythonovú funkciu konkrétnej arity.

Ako sme spomínali vyššie, pre podpravidlo vyrobíme nový neterminál. Zápis `(a|b|c)` nahradíme novým neterminálom α_i a do gramatiky pridáme nové pravidlá $\alpha_i \to a$, $\alpha_i \to b$, $\alpha_i \to c$ (skrátene povedané $\alpha \to a \mid b \mid c$). Ak je za zátvorkou otáznik, miesto toho pridáme pravidlá $\alpha_i \to a \mid b \mid c \mid \varepsilon$. A ak hviezdička, pridáme $\alpha_i \to a\alpha_i \mid b\alpha_i \mid c\alpha_i \mid \varepsilon$. (Všimnime si, že výsledok nepoužíva ľavú rekurziu, takže ho môžeme použiť v LL gramatikách.) Implementácia dá novému neterminálu α_i názov tvaru `abc__i`, kde `abc` je meno pôvodného neterminálu na ľavej strane (ktorý bol v **mgr** súbore pred dvojbodkou) a `i` je ďalšie voľné číslo v poradí. Zápis `\i` tiež prepíšeme na `abc__i`.

Rekurzívnym opakovaním tohto postupu sa zbavíme podpravidiel a dostaneme pravidlá, ktoré môžeme bez ďalších zmien použiť vo výslednej inštancii `Grammar`. Ale každému pravidlu ešte musíme priradiť akciu. Pôvodná akcia pre rozšírené pravidlo, ktoré sme dostali na vstupe, očakáva aj prístup ku premenným „`x=foo`“ definovaným v podpravidlách. Keďže podpravidlá sme zmenili na samostatné pravidlá, musíme im dať vhodné akcie.

Vráťme sa ku príkladu `ids: l+=ID ("," l+=ID)* ‘[tok.value for tok in l]‘`. Náš postup na odstránenie podpravidiel určí, že vo výslednej gramatike budú tri pravidlá $\{$`ids` $\to$ `ID ids__1`, `ids__1` $\to$ `"," ID ids__1`, `ids__1` $\to \varepsilon\}$. Každému pravidlu ešte musíme dať vhodnú akciu, aby neterminál `ids` nadobudol správnu hodnotu. Ak parsujeme napríklad vstup „`a,b,c,d`“, ktorý vhodným tokenizérom preložíme na terminály `ID,ID,ID,ID`, uvidíme nasledovný strom odvodenia:



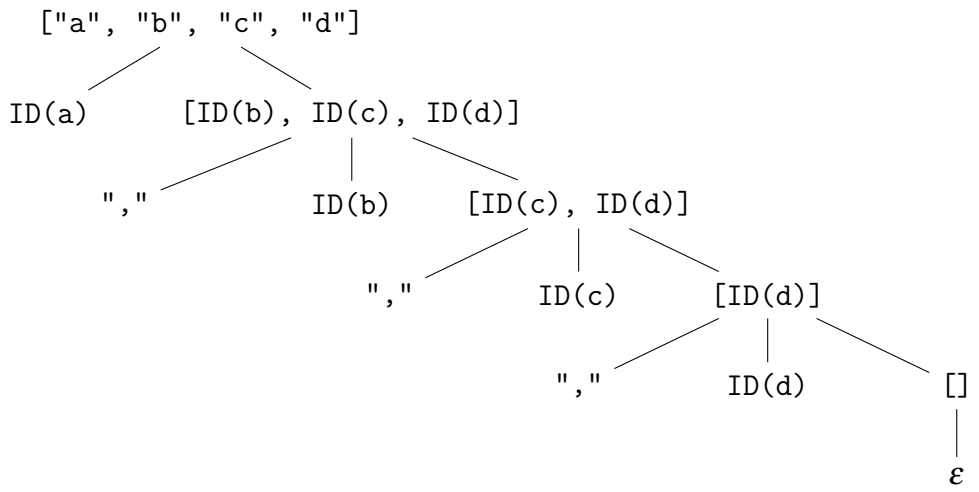
Vstupná akcia ‘`[tok.value for tok in l]`‘ očakáva pole `l` obsahujúce všetky tokeny typu `ID`. Problém je v tom, že akcie sú „postorder“ – spúšťajú sa, keď v strome odvodenia odchádzame z hlbšej vrstvy. Keby sme mali aj „preorder“ akcie, bolo by to jednoduché: pri vchode do neterminálu `ids` by sme v parseri na nejakom zásobníku vyrobili prázdne pole,

nejako by sme do neho postupne pridali jednotlivé `ID`, a pri odchode z `ids` by sme iba zobrali hotové pole a spustili akciu zadanú používateľom. Lenže „preorder“ akcie nemáme. Do LL parsera by sme ich ešte vedeli doplniť, ale LR parser to z princípu nedokáže. To, v akom je pravidle, sa dozvie, až keď ho zredukuje na neterminál. Takže akcia pre `ids__1` → `"," ID ids__1` nemôže len „pridať `ID` do nejakého poľa“ – žiadne pole vtedy ešte nemusí existovať.

Riešením by mohlo byť, aby akcia pre `ids__1` vždy vrátila čiastočné pole `l`. Keď chceme dlhšie pole, vyrobíme jeho kópiu a pridáme nové `ID`. Takže prevod nášho pravidla z rozšíreného na štandardný zápis by mohol použiť tieto akcie:

```
ids = ID ids__1          ‘lambda ctx, id, l: [tok.value for tok in [id]+l]‘
ids__1 = "," ID ids__1   ‘lambda ctx, op, id, l: [id]+l‘
ids__1 =                 ‘lambda ctx: []‘
```

V jednotlivých vrcholoch nášho stromu odvodenia by boli tieto hodnoty:

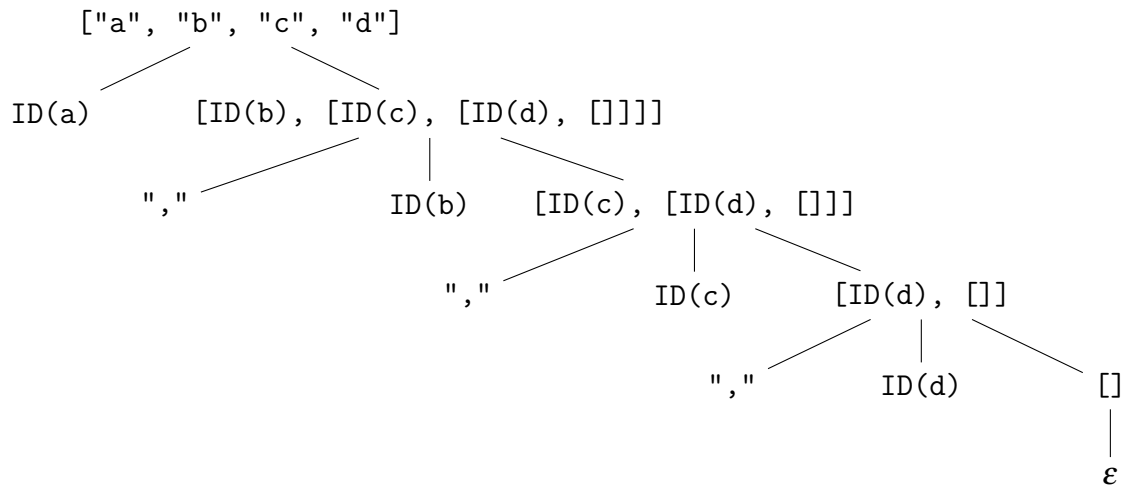


Toto riešenie má jeden problém: kvadratickú časovú zložitosť. Na každej úrovni kopírujeme celé pole `l` a o konštantu ho zväčšíme. To si pochopiteľne nemôžeme dovoliť.

Preto použijeme upravený postup: Kopírovať pole je síce pomalé, ale odkazovať sa na existujúce pole je zadarmo. Miesto toho, aby sme hodnoty synov spájali, vyrobíme vždy nové pole, ktoré ich iba obsahuje. Vrstva po vrstve vznikne stromová štruktúra, ktorej tvar lokálne odráža tvar stromu odvodenia. Na konci, keď príde čas spustiť pôvodnú akciu a dať jej pole `l`, naše vnorené pole už len sploštíme.

Pre túto variantu by gramatika a strom hodnôt mohli vyzerať nasledovne:

```
from context import flatten
---
ids = ID ids__1          ‘lambda ctx, id, l: [tok.value for tok in flatten([id, l])]‘
ids__1 = "," ID ids__1   ‘lambda ctx, op, id, l: [id, l]‘
ids__1 =                 ‘lambda ctx: []‘
```



Funkcia `flatten` dostala vstup `[ID(a), [ID(b), [ID(c), [ID(d), []]]]]` a vrátila sploštené pole `[ID(a), ID(b), ID(c), ID(d)]`. Pôvodná akcia potom toto pole tokenov previedla na pole reťazcov `["a", "b", "c", "d"]`.

Na tomto základe môžeme sformulovať všeobecný postup pre viacero premenných. Každý „ozajstný“ terminál alebo neterminál (ktorý bol aj na vstupe, nie je len odkazom na podpravidlo) uložíme ako dvojicu „(meno premennej, hodnota)“. Keď celú stromovú štruktúru sploštíme, dostaneme pole dvojíc. Ak tie dvojice roztriedime podľa mena premennej, dozvieme sa o každej premennej, aké hodnoty postupne nadobúdala. Pre zoznamové premenné je to presne to, čo potrebujeme, a pre normálne premenné len zoberieme poslednú hodnotu.

Pozrime sa na ďalší príklad. V gramatike vyššie sme videli pravidlo v rozšírenom zápise:

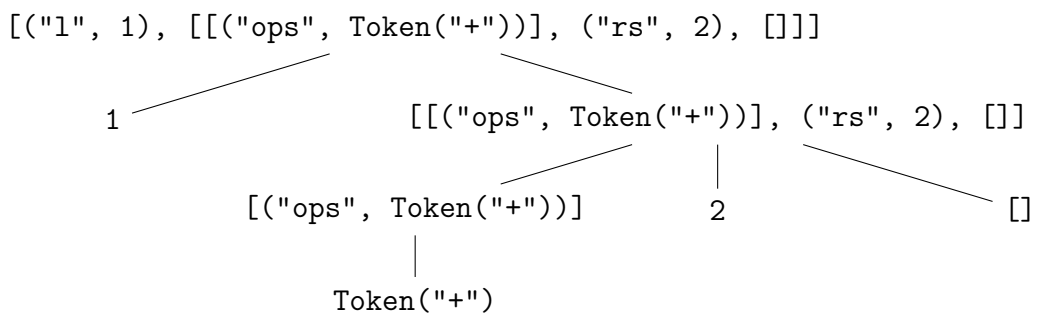
```
sum: l=product ((ops+="+" | ops+="-") rs+=product)* ‘left_associative(l, ops, rs)‘
```

Tento rozšírený zápis prevedieme na tieto pravidlá a akcie:

```
sum = product sum__1            ‘lambda ctx, v1, v2:
                                     nejako_vyhodnoť(flatten([("l", v1), v2]))‘
sum__1 = sum__2 product sum__1 ‘lambda ctx, v1, v2, v3: [v1, ("rs", v2), v3]‘
sum__1 =                       ‘lambda ctx: []‘
sum__2 = "+"                   ‘lambda ctx, v1: [("ops", v1)]‘
sum__2 = "-"                   ‘lambda ctx, v1: [("ops", v1)]‘
```

Všimnime si druhé pravidlo: Hodnoty `v1` a `v3` z umelých neterminálov `sum__2` a `sum__1` vraciame priamo, zatiaľčo hodnotu `v2` zabalíme do `("rs", v2)`, lebo `product` bol aj v rozšírenom zápise.

Pre vstup „`1 + 2`“ vytvoria jednotlivé akcie nasledovný strom hodnôt. (Strom odvodenia pre `product` zanedbáme. V koreni zatiaľ uvádzame len vstup pre funkciu `flatten`.)



Funkcia `flatten` „vymaže vnútorné hranaté zátvorky“ a vráti plochý zoznam dvojíc `[("l", 1), ("ops", Token("+")), ("rs", 2)]`. Fiktívna funkcia „nejako_vyhodnoť“ tento zoznam roztriedi podľa prvej položky každej dvojice, čím zistí, že `l` je `1`, `ops` je `[Token("+")]` a `rs` je `[2]`. (`ops` a `rs` sú zoznamové premenné.) S týmito hodnotami môže spustiť pôvodnú akciu `left_associative(l, ops, rs)`.

Ozajstnou obdobou fiktívnej funkcie „nejako_vyhodnoť“ je funkcia `combine_action()`, ktorá umožňuje používať rôzne zoznamové premenné a rôzne pôvodné akcie. Akcia pre pravidlo „`sum = product sum__1`“ z nášho príkladu bude v skutočnosti nasledovná:

```
sum = product sum__1 ‘combine_action(
        root=lambda ctx, v1, v2: [(’l’, v1), v2],
        user=lambda _ctx, _all, _loc, l, ops, rs: left_associative(l, ops, rs),
        normal_vars=[’l’],
        list_vars=[’ops’, ’rs’])‘
```

Funkcia `combine_action` podľa zadaných argumentov zostrojí akciu, ktorá spraví, čo sme opísali: sploští vstupné pole, určí hodnoty premenných a zavolá používateľom nastavenú akciu. Okrem toho poskytne používateľovej akcii aj premenné `_ctx`, `_all` a `_loc`.

Implementačné detaily rozšíreného zápisu sú takto odabstrahované aj od autora gramatiky, aj od generátorov parserov. Autor gramatiky môže používať podpravidlá a premenné, a generátor dostane normálnu bezkontextovú gramatiku.

Implementácia formátu **mgr** sa nachádza v adresári `mgr`. Formálna gramatika je definovaná v súbore `mgr/metagrammar.py`. Sama nepoužíva formát **mgr**, aby nebol závislý na sebe samom. Ale nie sú zapísané ani ako neprehľadný Pythonový literál. Inštancia `Grammar` je vytvorená na základe špeciálnej zjednodušenej verzie formátu **mgr**, ktorá je na tento účel definovaná v tom istom súbore.

Z tejto gramatiky vznikne LR parser `mgr/parser_out.py`. Tento parser je ďalej použitý v súbore `mgr/read.py`, ktorá riadi celý proces, a implementuje aj prevod rozšíreného zápisu na normálne pravidlá.

Súbory `*.mgr` je možné priamo použiť ako vstup pre skripty `makell.py` a `makelr.py`. Vygenerovanú gramatiku navyše môžeme zobraziť skriptom `mgr/dump.py`:

```
$ python3 -m mgr.dump example_extrules.mgr
$ python3 makelr.py example_basicrules.mgr example_basicrules_out.py
$ python3 makell.py toypython/grammar.mgr toypython/grammar_out.py
$ python3 runparser.py toypython.grammar_out mgr/makeparser.py
```

2.7 Generátor PEG parserov

Na rozdiel od LL a LR parserov Multiparser neobsahuje svoj vlastný generátor PEG parserov. Miesto toho používa externú Pythonovú knižnicu Grako [1], ktorú rozširuje o podporu vnorených jazykov. Použitie externej knižnice dosvedčuje, že Multiparser nie je uzavretý systém, ale dá sa prepojiť aj s inými generátormi.

Grako pre gramatiky parsovacích výrazov používa svoj vlastný formát nazvaný EBNF. (Tento názov je síce odvodený od *Extended Backus-Naur Form*, čo je spôsob zápisu bezkontextových gramatík, ale obsahom súboru je napriek tomu gramatika parsovacích výrazov.) Zo vstupného `*.ebnf` súboru vznikne Pythonový súbor implementujúci parser pre daný jazyk. Tento parser je implementovaný ako trieda, ktorá dedí z nadtriedy `grako.parsing.Parser`. Akcie pravidiel nie sú zapísané priamo v EBNF súbore, ale v samostatnej „sémantickej triede“. Dodatočné nastavenia, ktoré menia správanie parsera, sa tiež neurčujú v EBNF súbore, ale za behu. To je rozdiel oproti formátu **mgr** zo sekcie 2.6, ktorý obsahuje aj pravidlové akcie, aj gramatické vlastnosti.

Vygenerovaný Grako parser je možné použiť nasledovne:

```
from foo import FooParser, FooSemantics

class MySemantics(FooSemantics):
    # definujeme naše vlastné akcie...

buffer = grako.buffering.Buffer("vstupný text", dodatočné nastavenia...)
parser = FooParser()
result = parser.parse(buffer, "počiatočný_neterminál",
                      semantics=MySemantics(), dodatočné nastavenia...)
```

Trieda `Buffer` je obdobou Multiparserovej triedy `ParserContext` – pamätá si vstupný text a pozíciu v ňom.

Toto rozhranie musíme zabaliť do funkcie `parse()`, akú podľa sekcie 2.1 očakáva Multiparser. Táto funkcia dostane objekt `ParserContext`, ktorého pozícia nemusí byť na začiatku vstupu. Preto musí túto pozíciu odovzdať objektu `Buffer`, a keď PEG parser skončí, výslednú pozíciu zapísať naspäť do `ParserContext`. Obe triedy sú našťastie také podobné, že stačí priradiť `buffer.pos = ctx.pos`, respektíve `ctx.pos = buffer.pos`.

`Buffer` dokáže predspracovať vstup, aby tabulátory zmenil na medzery. Ale zmeny vstupného textu by spôsobili desynchronizáciu indexov `pos` medzi Grakom a Multiparserom. Multiparser preto používa svoju vlastnú podtriedu `Buffer`, nazvanú `IntegratedBuffer`, ktorá predspracovanie vynecháva. Tiež memoizuje niektoré výpočty, ktoré závisia iba od vstupného textu, takže vytvoriť viaceré `IntegratedBuffer` objekty z toho istého `ParserContext` objektu netrvá zbytočne dlho.

Ďalej doplníme podporu zatváracích oddeľovačov a argumentu `close_with`. Zo sekcie 1.4.1 vieme, že v PEG gramatikách sa koniec vstupu vyjadruje výrazom „!.“. Grako má

pre tento výraz vo svojom EBNF formáte vlastný syntaktický cukor – symbol „$“. Vo vygenerovanom parseri sa tento symbol preloží na volanie metódy `self._check_eof()`, ktorá za normálnych okolností spraví to, čo formálny výraz „!.“.

V Multiparseri použijeme podtriedu `FooParser`, v ktorej predefinujeme `_check_eof()`. Ak je `close_with` rovné `None`, použijeme pôvodné správanie. Ale ak má `close_with` inú hodnotu, metóda `_check_eof()` (a teda symbol `$` v EBNF súbore) zmení svoje správanie, a namiesto kontroly konca vstupu skontroluje prítomnosť daného zatváracieho oddeľovača. Efekt je rovnaký, ako keby sme v EBNF súbore nahradili výraz „!.“ výrazom pre zatvárací oddeľovač.

Trieda `Buffer` a metóda `Parser.parse()` majú v Graku mnoho nepovinných argumentov, ktoré môžu meniť správanie parsera – dôležitý je zvlášť argument `semantics`, ktorý určuje akcie pravidiel. Multiparser musí tieto dodatočné nastavenia podporovať, ale funkcia `parse()` musí mať iba argumenty `ctx`, `start_nt` a `close_with`.

Multiparser preto rozširuje syntax EBNF súborov, aby sa dodatočné nastavenia dali zapísať priamo v nich (podobne ako gramatické vlastnosti v `*.mgr` súboroch). Grako ignoruje komentáre uzavreté medzi `(*` a `*)`, ale Multiparser hľadá špeciálny komentár, ktorý je tvaru `(*multiparser_options(...)*)`. Ak taký existuje, v zátvorkách musia byť Pythonové pomenované argumenty. Tie sa odovzdajú konštruktoru `Buffer()` a metóde `Parser.parse()`.

Napríklad argument `whitespace=''` v Graku vypne automatické preskakovanie bieleho miesta. Ak v EBNF súbore napíšeme `(*multiparser_options(whitespace='')*)`, funkcia `parse()` vyrobí objekt `Buffer` s týmto argumentom.

Hodnoty môžu byť ľubovoľné Pythonové výrazy. Ak chceme určiť sémantickú triedu, môžeme sa pomocou funkcie `import_module` odvolať na iný modul:

```
(*multiparser_options(
    whitespace='',
    semantics=import_module("foo.semantics").MySemantics(),
)*)
```

Okrem nastavení pre Grako môžeme nastaviť aj `default_start="..."`, ktorým sa určí počiatočný neterminál v prípade, že `parse()` nedostane argument `start_nt`. Ak nie je prítomný ani `default_start`, použije sa neterminál nazvaný `"start"`.

Celé generovanie PEG parsera riadi súbor `makepeg.py`. Zo vstupného EBNF súboru vytvorí dva Pythonové súbory: v jednom bude výstup vygenerovaný Grakom, zatiaľčo v druhom bude funkcia `parse()` a všetko ostatné, čo je špecifické pre Multiparser. Tento druhý súbor bude mať približne nasledovný obsah:

```
from .foo_ebnf_out import FooParser        # importujeme vygenerovaný parser

class IntegratedBuffer(Buffer):
    def __init__(self, ctx, **kwargs):
        self.ctx = ctx                     # uložíme si náš ParserContext
        super().__init__(ctx.input, **kwargs) # nadtriede dáme vstupný text z ctx
```

```
    # (...ďalšie metódy na vypnutie predspracovania a pridanie memoizácie...)

class Parser(FooParser):
    def _check_eof(self):
        if self.close_with == None:        # ak je close_with rovné None,
            super()._check_eof()           # použijeme pôvodnú _check_eof()
        else:
            self._token(self.close_with)   # inak čítame close_with na vstupe

def parse(ctx, start_nt=None, close_with=None):
    parser = Parser()
    parser.close_with = close_with         # close_with uložíme v objekte Parser

    options = dict(...)                    # sem sa doplní multiparser_options
    default_start = options.pop('default_start', 'start')

    buffer = IntegratedBuffer(ctx, **options)
    buffer.pos = ctx.pos                   # povieme Graku začiatočnú pozíciu
    result = parser.parse(buffer, start_nt or default_start, **options)
    ctx.pos = buffer.pos                   # dozvieme sa od Graka koncovú pozíciu
    return result
```

2.8 Zložené parsery

Naším cieľom je parsovať vstupný text obsahujúci kombináciu viacerých jazykov. To dosiahneme kombináciu viacerých parserov pre jednotlivé jazyky. Výsledný kôd nazývame *zložený parser*.

Zložené parsery sú generované podľa konfiguračného súboru `config.py`. Tento súbor určuje, aké jazyky sa použijú, a ako sa do seba môžu vnárať. Pre každý jazyk je možné definovať viaceré páry oddeľovačov a viaceré počiatočné neterminály. Napríklad môžeme rozlišovať medzi vnoreným zoznamom príkazov a vnoreným výrazom.

Súbor `config.py` musí definovať tieto premenné:

- `languages`: slovník použitých jazykov. Každý kľúč je identifikátor jazyka, a každá hodnota je cesta ku `.mgr` alebo `.ebnf` súboru s gramatikou (viď sekcie 2.6 a 2.7).
- `root`: identifikátor jazyka najvyššej úrovne, od ktorého začne parsovanie vstupu.
- `embeds`: pole pravidiel povoleného vnárania. Každá položka je šestica (vonkajší jazyk, typ tokenu, vnútorný jazyk, počiatočný neterminál, otvárací oddeľovač, zatvárací oddeľovač). Ak parser vonkajšieho jazyka prečíta otvárací oddeľovač, začne parsovať vnútorný jazyk, začínajúc od daného počiatočného neterminálu, a skončí po zatváracom oddeľovači. Výsledok sa vo vonkajšom jazyku zabalí do tokenu daného typu.

Zložený parser pre jazyky Foo a Bar by sme mohli nastaviť nasledovne:

```
languages = {
    'foo': 'foo/foo.mgr',
    'bar': 'bar/bar.mgr',
}

root = 'foo'

embeds = [
    ('foo', 'BAR_CODE', 'bar', None, '<?bar', '?>'),
]
```

Tieto nastavenia umožňujú iba jednosmerné vnáranie Bar vnútri Foo. Ak tokenizér pre Foo prečíta text „<?bar“, spustí funkciu `parse(ctx, start_nt=None, close_with='?>')` pre jazyk Bar. Keď funkcia `parse` skončí, jej výsledok `x` sa zabalí do tokenu s typom `"BAR_CODE"` a hodnotou `x`. Pravidlá gramatiky `foo.mgr` môžu použiť terminál `BAR_CODE` a v akciách sa odkazovať na jeho `value`.

Ak máme viac ako dva jazyky, a chceme povoliť vnáranie každého do každého, môžeme pole `embeds` vytvoriť v cykle, alebo pomocou *list comprehension* notácie. Multiparser síce umožňuje aj vnárať jazyk do seba samého, ale väčšinou to nemá zmysel – rovnaký efekt sa dá dosiahnuť aj priamo gramatickými pravidlami.

Konštrukcia zloženého parsera je implementovaná skriptom `compose.py`. Tento skript načíta súbor `config.py` a vygeneruje v jeho adresári Pythonové súbory, ktoré spolu tvoria zložený parser. Všetky automaticky vygenerované súbory majú mená končiace na „`_out.py`“, aby sa dali ľahko rozoznať. Výsledný parser sa skladá z parserov pre jednotlivé jazyky a z hlavného súboru `composite_out.py`, ktorý ich spája.

Ak je v poli `languages` jazyk `foo`, jeho parser bude v `foo_out.py`. Typ parsera sa určí v prvom rade podľa prípony súboru s gramatikou: ak končí na `.ebnf`, ide o PEG parser, a ak končí na `.mgr`, je to LL alebo LR parser. Medzi týmito možnosťami sa skript `compose.py` rozhodne podľa gramatickej vlastnosti `"type": "LL"` resp. `"type": "LR"`. Každá `.mgr` gramatika, ktorú použijeme v `config.py`, musí mať nastavený `"type"`.

Zložený parser ešte potrebuje podporu pre otváracie oddeľovače. V prípade LL a LR parserov musíme rozšíriť každý tokenizér. Aby sme predišli cyklickým závislostiam, tokenizéry nebudú priamo volať funkcie `parse()` z iných jazykov. Miesto toho iba zavolajú metódu `embed_token()`, ktorá vyskúša všetky pravidlá vnárania, a spustí `parse()`, keď treba.

Do tokenizéru pre jazyk Foo teda stačí doplniť tento kód:

```
def tokenizer(ctx, close_with=None):
    while True:
        # ...ostatné pravidlá...

        e = ctx.embed_token('foo')   # Sme v jazyku 'foo' a chceme vedieť,
        if e:                        # či na vstupe nasleduje vnorený jazyk.
            yield e                  # Ak áno, vrátený Token odovzdáme parseru.
            continue

        # ...ostatné pravidlá...
```

Metóda `embed_token` je implementovaná v súbore `composite_out.py`:

```
class ParserContext(BaseParserContext):
    def embed_token(self, outer_language):
        loc = self.location
        # vyskúšame každé pravidlo v poli embeds...
        for outer, token_type, inner, start, opener, closer in embeds:
            # ak sedí vonkajší jazyk, a na vstupe nasleduje otvárací oddeľovač...
            if outer == outer_language and self.peek(len(opener)) == opener:
                # posunieme sa za otvárací oddeľovač
                self.advance(len(opener))
                # spustíme funkciu parse() pre vnorený jazyk
                parse = parsers[inner]
                inner_ast = parse(self, start, closer)
                # výsledok vrátime ako Token s daným typom a hodnotou
                return Token(loc, token_type, inner_ast)
```

PEG parsery nemajú samostatný tokenizačný krok, preto musíme použiť iný spôsob – viď sekcia 1.4.1. Pre každý „typ tokenu“, ktorý je nastavený v poli `embeds`, pridáme do PEG gramatiky špeciálny neterminál s predponou „`embed_`“. (Teda napríklad pre `"BAR_CODE"` môžu EBNF pravidlá použiť `embed_BAR_CODE`.) Multiparser tento neterminál definuje tak, že buď prečíta otvárací oddeľovač a za ním kód vo vnorenom jazyku, alebo neuspeje a kurzor nechá na pôvodnom mieste.

Tieto špeciálne neterminály sú implementované ako ďalšie metódy pridané do triedy `Parser` zo sekcie 2.7. Volanie funkcie `parse()` pre vnorený jazyk prebieha podobne, ako v metóde `embed_token`, ale navyše sa pred a po volaní `parse()` synchronizuje hodnota `poz` medzi objektmi `ctx` a `buffer`.

Grako z technických príčin vyžaduje, aby mal každý použitý neterminál nejaké pravidlo. Ak v EBNF súbore použijeme neterminál `embed_BAR_CODE`, musíme mu pridať umelé pravidlo „`embed_BAR_CODE = !() ;`“ – PEG výraz, ktorý nikdy neuspeje.

EBNF súbor pre jazyk Foo by mohol vyzerať napríklad nasledovne:

```
start = expr $ ;
expr = sum ;
sum = product ('+' product)* ;
product = atom ('*' atom)* ;
atom = 'x' | '(' expr ')' | embed_BAR_CODE ;
embed_BAR_CODE = !() ;   # táto definícia sa nepoužije, Multiparser ju prepíše
```

Neterminál `atom` v tomto príklade môže prečítať buď `'x'`, alebo výraz v zátvorkách, alebo ľubovoľný vnorený kód, ktorý má v poli `embeds` nastavený vonkajší jazyk `"foo"` a typ tokenu `"BAR_CODE"`. Neterminál `embed_BAR_CODE` sa postará aj o otváracie a zatváracie oddeľovače, EBNF súbor o nich nemusí vedieť.

Výstupom skriptu `compose.py` teda je zložený parser, ktorý podľa nastavení dokáže čítať vnorené programovacie jazyky. Jednotlivé parsery reagujú na otváracie oddeľovače a volajú parsovacie funkcie iných jazykov. Keď vnorený parser prečíta zatvárací oddeľovač, skončí a vráti kontrolu vonkajšiemu parseru. Výsledný systém je kompletnou implementáciou návrhu z kapitoly 1.

Kapitola 3

Ukážkové jazyky

Multiparser obsahuje tri ukážkové jazyky: ToyCalc, ToyPython a ToyLua. Spolu dokazujú, že Multiparser vie parsovať nielen triviálne príklady, ale aj komplexné programovacie jazyky, aké sa používajú v praxi.

Jazyky sa volajú „Toy...“, pretože nie sú určené na praktické použitie. Multiparser je zodpovedný iba za syntaktickú analýzu – dokáže zostrojiť abstraktný syntaktický strom, ktorý používa ToyPython aj ToyCalc, ale nesnaží sa ho nijako ďalej interpretovať alebo kompilovať. Tieto ukážkové jazyky nemajú žiaden kompilátor, ktorý by ich ďalej spracoval a preložil do spustiteľnej podoby.

3.1 ToyCalc

ToyCalc je najjednoduchší ukážkový jazyk Multiparsera. Je podobný kalkulačkovým príkladom zo sekcie 2.6 – s tým rozdielom, že výsledky nepočíta a nevypisuje hneď, ale konštruuje abstraktný syntaktický strom. Tento strom nie je veľmi zložitý, keďže jeho vrcholy sú iba `+`, `-`, `*`, `/`, čísla a vnorené výrazy.

ToyCalc nepozná „príkazy“. Má iba „výrazy“, ktoré vracajú výsledok. Keď do ToyCalcu vnárame iné jazyky, tiež sa musíme obmedziť iba na výrazy.

ToyCalc má vďaka svojej jednoduchosti tri rôzne implementácie: existuje pre neho LL parser, LR parser aj PEG parser. Zložené parsery si môžu vybrať, ktorý parser chcú použiť. Takto sa dá overiť, že všetky typy parserov dokážu kooperovať.

3.2 ToyPython

ToyPython je na rozdiel od ToyCalcu založený na netriviálnom praktickom jazyku – konkrétne na Pythone 3.4. Jeho gramatika vznikla z oficiálnej Pythonovej gramatiky, a výsledkom

parsera je abstraktný syntaktický strom kompatibilný s modulom `ast` zo štandardnej knižnice (okrem nových vrcholov `EmbedStat` a `EmbedExp`).

Python je známy tým, že bloky kódu sú určené odsadením riadkov. Väčšina ostatných jazykov používa biele znaky len na oddelenie jednotlivých tokenov, ale v Pythone majú význam.

Pythonový program nemusí používať jednotné odsadenie – rôzne bloky môžu byť odsadené rôznym počtom medzier či tabulátorov. Tokenizér si pamätá úrovne odsadenia všetkých blokov, v ktorých sa práve nachádza. Ak má nový riadok väčšie odsadenie, ako predošlý, tokenizér vyrobí tokeny „NEWLINE“ a „INDENT“, a zapamätá si nové odsadenie. Ak má nový riadok menej medzier, tokenizér zistí, koľko otvorených blokov kódu sa tým končí, a za „NEWLINE“ vyrobí dané množstvo tokenov „DEDENT“. Keď sa odsadenie zmenšuje, musí sa vrátiť na známu úroveň.

Niektoré konce riadkov sa „nepočítajú“. Ak sa znak konca riadku nachádza v zátvorkách (či už okrúhlych, hranatých alebo kučeravých), alebo je pred ním „\“, ďalší riadok sa pokladá za pokračovanie predošlého riadku a jeho úroveň odsadenia sa ignoruje. Ak je riadok prázdny (resp. obsahuje iba biele znaky alebo komentáre), nevznikne ani token „NEWLINE“.

Tokenizér pre ToyPython implementuje podobný algoritmus, ale oproti Pythonu má dva rozdiely. Za prvé pre jednoduchosť vôbec nepodporuje tabulátory, a za druhé vnoreným ToyPython programom dovoľuje, aby mal prvý riadok nenulové odsadenie. Ak je ToyPython program vnorený vo väčšej funkcii, ktorá už je odsadená, nebolo by vhodné, keby jeho odsadenie muselo začínať od nuly.

Z ostatných Pythonových tokenov podporuje ToyPython iba nutné minimum. Mená premenných nemôžu obsahovať neobvyklé Unicode znaky, číselné literály môžu byť len celé čísla v desiatkovej sústave, a reťazcové literály nemôžu obsahovať žiadne escape sekvencie ani používať predpony `b`, `r` a `u` (pre bajtové, surové a unikódové reťazce). Toto je hlavnou prekážkou pri čítaní existujúceho Pythonového kódu. Cieľom ToyPythonu je ukázať použitie existujúcej gramatiky, takže úplnosť tokenizéra nebola prioritou.

Gramatika ToyPythonu je založená na oficiálnej Pythonovej gramatike [7]. Jej formát sa na pohľad podobá na formát **mgr**:

```
dictorsetmaker: ( (test ':' test (comp_for | (',' test ':' test)* [','])) |
                  (test (comp_for | (',' test)* [','])) )
subscriptlist: subscript (',' subscript)* [',']
```

Ale ak by sme tieto pravidlá skúsili prepísať do **mgr** (nahradiť `[]` za `()?` a podobne) a spustili generátor parserov, zistili by sme, že výsledná gramatika nie je LL(1) ani LR(1). Python často používa čiarkami oddelené zoznamy prvkov, ktoré majú nepovinnú koncovú čiarku, ale naivný preklad týchto pravidiel do **mgr** spôsobí LL(1) aj LR(1) konflikt:

```
subscriptlist = subscript subscriptlist__1 subscriptlist__2
subscriptlist__1 = ',' subscript subscriptlist__1
subscriptlist__1 =
```

```
subscriptlist__2 = ','
subscriptlist__2 =
```

Python v skutočnosti nepoužíva LL(1) parser, ale variantu takzvaných LL(*) parserov. Hlavná myšlienka je nasledovná: parser Pythonu na rozdiel od formátu **mgr** nepoužíva žiadne pomocné neterminály. Miesto toho každé pravidlo interpretuje ako regulárny výraz, pre ktorý štandardnou konštrukciou zostrojí deterministický konečný automat. LL a LR parsery musia v `subscriptlist`-e tipovať, či vidia „vnútornú čiarku“ alebo „koncovú čiarku“, ale v konečnom automate je jednoducho stav „videl som čiarku, ale ešte neviem, či je vnútorná, alebo koncová“ – podobne, ako má deterministický konečný automat pre regulárny výraz „`a(ba)*(b)?`“ stav „videl som `b`, ale ešte neviem, či vnútorné, alebo koncové“. Ak sa konečný automat rozhoduje medzi viacerými neterminálmi, musia mať disjunktné množiny *FIRST*, aby ich automat mohol rozoznať podľa prvého terminálu.

Detaily Pythonového parsera pre nás nie sú podstatné. Dôležité je, že Multiparser podporuje iba LL(1), LR(1) a PEG parsery. Preto sa pravidiel, na ktoré LL(1) parser nestačí, musíme zbaviť. Tento proces by sa dal zrealizovať automaticky (vygenerované deterministické automaty sa dajú previesť na regulárne gramatiky, ktoré sú LL(1)), ale pre nás bude jednoduchšie a názornejšie pozmeniť problémové pravidlá ručne. Je ich len málo a sú pomerne jednoduché.

Neterminál `subscriptlist` môžeme zapísať nasledovne:

```
subscriptlist: subscript (',' (subscript \1)?)?
```

Čo je ekvivalentné tejto gramatike:

```
subscriptlist = subscript subscriptlist__1
subscriptlist__1 = ',' subscriptlist__2
subscriptlist__1 =
subscriptlist__2 = subscript subscriptlist__1
subscriptlist__2 =
```

Neterminály `subscriptlist__1` a `subscriptlist__2` zabezpečia, že `subscript`-y a čiarky sa budú striedať, ale zoznam môže kedykoľvek skončiť. Tieto pravidlá sú ekvivalentné pôvodnej definícii `subscriptlist`, ale nespôsobujú LL(1) konflikt.

Podobne môžeme pozmeniť väčšinu ostatných problémových pravidiel. Napríklad pravidlo pre `dictorsetmaker` upravíme nasledovne:

```
dictorsetmaker: test ( (':' test (comp_for | (',' (test ':' test \4)?)? )) |
                       comp_for | (',' (test \6)?)? )
```

Spoločný prefix `test` sme vyňali pred podpravidlo, a zoznamy s nepovinnou koncovou čiarkou sme upravili rovnako, ako v pravidle pre `subscriptlist`.

Zostáva definovať akcie gramatiky. V Pythone sa abstraktný syntaktický strom konštruuje v súbore `Python/ast.c`. Akcie gramatika ToyPythonu sú založené na tomto súbore, ale reimplementujú ho v Pythone namiesto C. Výsledkom ToyPython parsera je preto abstraktný syntaktický strom kompatibilný s výstupom Python parsera.

Pomocné funkcie definované v súbore `toypython/grammar.mgr` odrážajú rovnako nazvané funkcie v `ast.c`. Napríklad funkcia `set_context()` robí v oboch parseroch dodatočné kontroly, ktoré nie sú definované priamo v gramatike. Pythonový príkaz `del` je síce definovaný pravidlom „`del_stmt: 'del' exprlist`“, ale nedá sa použiť s každým výrazom. Môžeme vymazať premennú „`del x`“, ale nie „`del 1 + 1`“. Funkcia `set_context()` mimo iného kontroluje, že výrazy sú použité v správnom kontexte.

Už chýba len tá najdôležitejšia časť: vnáranie iných jazykov. To je vďaka návrhu Multiparsera triviálne – tokenizér ani gramatika nemusí nič vedieť o tom, aké jazyky vnárame a aké oddeľovače používame. ToyPython iba určuje, že existujú dva typy vnárania: token „EMBEDSTAT“ predstavuje vnorený zoznam nula alebo viac príkazov, a token „EMBEDEXPR“ predstavuje jeden vnorený výraz, ktorý musí vrátiť nejakú hodnotu. Vo výslednom abstraktnom syntaktickom strome sú reprezentované novými typmi „`EmbedStat`“ a „`EmbedExp`“. Gramatika ToyPythonu rozširuje neterminály „`small_stat`“ (príkaz) a „`atom`“ (atomický výraz) o nové pravidlá, ktoré priamočiaro premieňajú „EMBEDSTAT“ a „EMBEDEXPR“ na „`EmbedStat`“ a „`EmbedExp`“.

ToyPython predstavuje príklad, ako sa dá zobrať netriviálny parser napísaný v inom jazyku a používajúci svoje vlastné technológie, reimplementovať ho v Multiparseri, a doplniť možnosť vnárania iných jazykov.

3.3 ToyLua

ToyLua je založená na jazyku Lua 5.2. Tento minimalistický jazyk má relatívne malú gramatiku, ale v tejto gramatike sa nachádza známa nejednoznačnosť. ToyLua má za cieľ ukázať, ako sa vysporiadať s konfliktmi, keď nechceme meniť pravidlá.

ToyLua tokenizér podporuje na rozdiel od ToyPythonu aj viacriadkové reťazce a väčšinu escape sekvencií, ktoré pozná Lua. Jediné, čo oproti Lue chýba, sú desatinné čísla a číselné escape sekvencie `\xXX` a `\ddd`.

Manuál jazyka Lua [8] popisuje jeho formálnu gramatiku. Ale interpreter jazyka túto gramatiku v skutočnosti vôbec nepoužíva. Miesto toho obsahuje ručne písaný C parser, ktorý je založený na úplne inej gramatike.

ToyLua implementuje obe gramatiky. Súbor `toylua/lrgrammar.mgr` obsahuje LR gramatiku založenú na oficiálnom manuáli, zatiaľčo súbor `toylua/llgrammar.mgr` obsahuje LL gramatiku, ktorá postupuje tak, ako interpreter.

Gramatika uvedená v Lua manuáli je takmer LR(1). ToyLua mení iba niekoľko pravidiel:

1. Neterminál `parlist` je v Lua manuáli definovaný nasledovne:

```
parlist: namelist ("," "...")? | "..."
namelist: "NAME" ("," "NAME")*
```

Tieto pravidlá vedú k shift-reduce konfliktu. Ak po prečítaní `Name` nasleduje čiarka, nevieme, či ešte pokračuje `namelist`, alebo už končí a za čiarkou bude nasledovať „. . .“. ToyLua preto používa iné pravidlo:

```
parlist: "..." | "NAME" ("," parlist)?
```

(Pre čitateľnosť nebudeme uvádzať akcie ToyLua pravidiel.)

2. Neterminál `tableconstructor` má podľa manuálu nasledovný obsah:

```
tableconstructor: "{" (fieldlist)? "}"
fieldlist: field (fieldsep field)* (fieldsep)?
fieldsep: "," | ";"
```

Pravidlo definuje zoznam prvkov s oddeľovačmi, ktorý môže končiť dodatočným oddeľovačom. Použijeme riešenie, ktoré sme videli v sekcii 3.2:

```
tableconstructor: "{" (field (fieldsep \1)?)? "}"
```

3. Gramatika v Lua manuáli zanedbáva priority operátorov:

```
exp: "nil" | "false" | "true" | "NUMBER" | "STRING" | "..." | functiondef |
     prefixexp | tableconstructor | exp binop exp | unop exp

binop: '+' | '-' | '*' | '/' | '^' | '%' | '..' |
   '<' | '<=' | '>' | '>=' | '==' | '~=' |
   and | or

unop: '-' | not | '#'
```

ToyLua ruší neterminály `binop` a `unop`. Miesto nich dáva každému operátoru vlastné pravidlo, aby bolo možné použiť systém priorít operátorov zo sekcie 2.5.1.

```
exp
  : "nil" | "false" | "true" | "NUMBER" | "STRING" | "..."
  | functiondef | prefixexp | tableconstructor
  | exp "or"  exp | exp "and" exp | exp "<"   exp
  | exp ">"   exp | exp "<="  exp | exp ">="  exp
  | exp "~="  exp | exp "=="  exp | exp ".."  exp
  | exp "+"   exp | exp "-"   exp | exp "*"   exp
  | exp "/"   exp | exp "%"   exp | exp "^"   exp
  | "not" exp     | "#" exp       | "-" exp
```

Prioritu potom stačí nastaviť vo vlastnostiach gramatiky:

```
'precedence': [
    [
        ('left', ['or']),
        ('left', ['and']),
        ('left', ['<', '>', '<=', '>=', '~=', '==']),
        ('right', ['..']),
        ('left', ['+', '-']),
        ('left', ['*', '/', '%']),
        ('precedence', ['not', '#', ('exp', ('-', 'exp'))]),
        ('right', ['^']),
    ],
]
```

Príkazy v Lue nemusia byť nijako oddelené – ani bodkočiarkami, ani koncom riadku. Nie je nutné písať „`f(1); f(2); f(3)`“. Lua dokáže rozoznať, že „`f(1) f(2) f(3)`“ sú tri volania.

Volaná funkcia môže byť ľubovoľný výraz uzavretý v zátvorkách. „`(a or b)(1)`“ spôsobí, že vyhodnotíme výraz „`a or b`“, a získanú funkciu zavoláme s argumentom `1`.

Tieto dve vlastnosti vedú k nejednoznačnosti. Vstup „`(a)(b)(c)(d)`“ je možné interpretovať aj ako „`a(b); c(d)`“, aj ako „`((a(b))(c))(d)`“ – voláme funkciu `a` s argumentom `b`, tá vráti funkciu, ktorú zavoláme s argumentom `c`, a jej výsledok zavoláme s argumentom `d`. Podobne, vstup „`a=(b)(c)(d)`“ môžeme interpretovať ako dva príkazy „`a=b; c(d)`“, alebo ako „`a=((b(c))(d))`“. Lua manuál hovorí, že parser v oboch prípadoch preferuje druhú interpretáciu.

Nejednoznačnosť vzniká kvôli nasledovným pravidlám:

```
block: (stat)* (retstat)?
stat: functioncall | NAME "=" exp | ...
functioncall: prefixexp args | ...
prefixexp: var | functioncall | "(" exp ")"
args: "(" explist ")" | ...
exp: NAME | prefixexp | ...
```

Generátor LR parserov nahlási dva konflikty. Prvým je reduce-reduce konflikt medzi pravidlom `prefixexp` → `functioncall` a pravidlom `stat` → `functioncall`, keď na vstupe nasleduje „`(`“. Pri parsovaní vstupu „`(a)(b)(c)(d)`“ tento konflikt nastáva po tom, čo parser prečíta „`(a)(b)`“, a zredukuje ho na `functioncall`. Tento `functioncall` môže považovať za hotový `stat`, alebo ho môže pokladať za `prefixexp` ku väčšiemu volaniu. My chceme preferovať tú druhú možnosť. Preto do vlastností gramatiky pridáme:

```
'ignore_reduces': {
    # Ak na vstupe nasleduje "(", a pravidlo "stat -> functioncall" je
    # v konflikte s iným pravidlom, vyhrá to druhé pravidlo.
    ('stat', ('functioncall',), '('),
},
```

Druhým konfliktom je shift-reduce konflikt. Generátor nám oznámi, že parser môže buď redukovať pravidlo `exp` → `prefixexp`, alebo shiftnúť z (`args` → •`(explist)`) do (`args` → `(`•`explist`)). Pri parsovaní vstupu „`a=(b)(c)(d)`“ tento konflikt nastáva po tom, čo parser prečíta „`a=(b)(c)(d)`“ a zredukuje „`(b)`“ na `prefixexp`. Parser sa musí rozhodnúť, či spraví shift a začne čítať `args`, alebo túto pravú stranu priradenia uzavrie ako hotovú.

Lua manuál hovorí, že „`(c)(d)`“ má byť tiež súčasťou priradenia, takže ToyLua parser má urobiť shift. Preto nastavíme, že token `"("` má väčšiu prioritu, ako pravidlo `exp` → `prefixexp`:

```
'precedence': [
    [
        ('precedence', [('exp', ('prefixexp',))]),
        ('precedence', ['(']),
    ],
```

```
]
```

ToyLua gramatika už nemá žiadne ďalšie konflikty a generátor pre ňu dokáže vyrobiť LR parser.

V LL gramatike `toylua/llgrammar.mgr`, ktorá je založená na ručne písanom parseri z Lua interpretra, spôsobuje nejednoznačnosť podobný konflikt. Ale tu sa dá vyriešiť omnoho jednoduchšie.

LL gramatika má namiesto neterminálu `prefixexp` neterminál `suffixedexp`, čo je výraz, za ktorým môže ísť niekoľko „suffixov“ – volanie funkcií a metód, indexovanie polí a podobne.

```
suffixedexp: primaryexp (suffixedexp_suffix)*
suffixedexp_suffix: "." "NAME" | "[" exp "]" | ":" "NAME" args | args
```

Nejednoznačnosť gramatiky v nej spôsobí iba jediný LL konflikt: keď `suffixedexp__1` vidí ľavú zátvorku, môže buď vojsť do `args`, alebo ukončiť postupnosť suffixov a výjsť zo `suffixedexp`. V tomto prípade je zjavné, že chceme vojsť do `args` – inak by vôbec nebolo možné volať funkcie. Tým je nejednoznačnosť vyriešená.

Samozrejme, obe gramatiky sú rozšírené o podporu vnárania príkazov a výrazov. Zatiaľčo v Pythone sa každý výraz dal použiť ako príkaz, v Lue sú výrazne oddelenejšie – väčšina príkazov nie sú výrazy, a väčšina výrazov nie sú príkazy.

3.4 Ukážkové zložené parsery

Zložené parsery sme si predstavili v sekcii 2.8. Multiparser obsahuje tri ukážkové zložené parsery, nazvané `mix_alpha`, `mix_beta` a `mix_gamma`.

Parser `mix_alpha` nazýva jazyky ToyCalc, ToyPython a ToyLua skratkami „calc“, „py“ a „lua“. Vnorený zoznam príkazov môže byť oddelený buď s oddeľovačmi `<?foo ?>`, alebo s `%foo{ }` (kde `foo` je daná skratka). Vnorené výrazy sa píšu do `%foo( )`. Vnáranie je neobmedzené – každý jazyk môže byť vnorený v každom. Všimnime si, že viaceré zatváracie oddeľovače sú tokeny, ktoré už v daných jazykoch existujú – viď sekcia 1.2.2.

Tu je príklad viacjazyčného programu, ktorý vie `mix_alpha` prečítať:

```
function hello(values, n)
    my_string = [[5]]               -- v ToyLue je to reťazec
    <?py
        my_nested_array = [[5]]     # v ToyPythone je to pole
        if len(values) > 5:
            print("over 5 values")
            # ?> zatvárací oddeľovač v komentári sa nepočíta
        else: yield n
        %lua{
            print("back" .. " in " .. "lua")
        }
        assert True
```

```
    ?>
    result = %calc(5 + %py(10 ** 6) * 6)  # ToyPython v ToyCalcu v ToyPythone
end
```

Parser `mix_beta` ukazuje opačný extrém. Jeho podpora vnárania je veľmi obmedzená. Nielen že umožňuje vnárať iba výrazy, ale navyše sa ani nedá určiť, ktorý jazyk bude vnorený. V ToyPythone je vždy vnorená ToyLua, v ToyLue ToyCalc a v ToyCalcu zase ToyPython. Všetky prípady používajú oddeľovače `(: :)`. Tento jazyk samozrejme nie veľmi použiteľný, ale demonštruje, že tie isté oddeľovače môžu mať rôzny význam podľa jazyka, v ktorom sú použité.

Program pre `mix_beta` môžu vyzerať nasledovne:

```
print((:(:7:):))
a = (: [[lua string]] :)
b = (: x .. y :)
c = (: 1 + (: (: (yield 1) :) * 2 :) ^ -6 :)
```

Parser `mix_gamma` používa na bloky príkazov oddeľovače podobné na HTML – `<lua>` `</lua>` a `<python>` `</python>`. Vnorené výrazy používajú otvárací oddeľovač `@@lua` (resp. `@@python`, `@@calc`) a zatvárací oddeľovač `@@@`. Keďže hneď po oddeľovači začína vnorený jazyk, v `mix_gamma` môžu nastávať situácie, ktoré by za normálnych okolností tokenizéry nedovolili: `@@pythonrange(7)@@@` sa vyhodnotí ako volanie funkcie `range` napriek tomu, že s oddeľovačom tvorí jedno slovo.

To môže viesť aj ku takýmto extrémnym prípadom:

```
print(@@calc1+2*@@python3+@@lua4@@@@@@@@@@)
```

Ale aj z tohto riadku dokážeme vytvoriť abstraktný syntaktický strom.

Vidíme, že Multiparser podporuje veľmi rôzne druhy otváracích a zatváracích oddeľovačov. Zložené parsery môžu definovať vlastné pravidlá vnárania, od viacerých oddeľovačov pre každý jazyk až po jediný povolený oddeľovač s jediným povoleným poradím vnárania. Aj rozlišovanie medzi „príkazmi“ a „výrazmi“ je nastaviteľné – pre naše tri ukážkové jazyky je tento konkrétny rozdiel užitočný, ale závisí iba od jazykov, ktoré si vyberieme.

Záver

Navrhli sme systém Multiparser, v ktorom viaceré parsery kooperujú pri čítaní jedného vstupného textu. Preskúmali sme klasické modely parsovania a rozšírili sme ich o podporu otváracích a zatváracích oddeľovačov, ktoré umožňujú vnáranie jazykov.

Celý systém sme implementovali – od generátorov LL a LR parserov, cez vhodný formát zápisu gramatík a akcií, po pokročilé riešenie konfliktov a iné rysy užitočné pri implementácii praktických jazykov.

Systém Multiparser splnil ciele práce. Dokáže generovať zložené parsery, ktoré kombinujú viaceré formálne modely a aj viaceré rôzne pravidlá lexikálnej analýzy. Tiež sme ukázali, že Multiparser si poradí aj s komplikáciami, ktoré nastávajú pri parsovaní praktických programovacích jazykov Python a Lua.

Literatúra

[1] Thomas Bragg and Juancarlo Añez. Grako: A generator of PEG/packrat parsers from EBNF grammars. `https://pypi.python.org/pypi/grako/3.5.1`, 2015.

[2] Franklin L DeRemer. *Practical translators for LR (k) languages*. PhD thesis, Massachusetts Institute of Technology, 1969.

[3] Franklin L DeRemer. Simple LR(K) grammars. *Commun. ACM*, 14(7):453–460, July 1971.

[4] Charles Donnelly and Richard Stallman. *Bison: the Yacc-compatible parser generator*, volume 1. Free Software Foundation, 1993.

[5] Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time. 2002.

[6] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM SIGPLAN Notices*, volume 39, pages 111–122. ACM, 2004.

[7] Python Software Foundation. Python 3.4: Full grammar specification. `https://docs.python.org/3.4/reference/grammar.html`, 2014.

[8] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes. Lua 5.2 reference manual. `http://www.lua.org/manual/5.2/`, 2011.

[9] Donald E Knuth. On the translation of languages from left to right. *Information and control*, 8(6):607–639, 1965.

[10] Bernard Lang. *Deterministic techniques for efficient non-deterministic parsers*. Springer, 1974.

[11] Michael E Lesk and Eric Schmidt. Lex: A lexical analyzer generator, 1975.

[12] Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: Reconciling object, relations and xml in the .net framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 706–706, New York, NY, USA, 2006. ACM.

[13] Terence Parr. The definitive antlr reference: building domain-specific languages. 2007.

[14] Daniel J Rosenkrantz and Richard Edwin Stearns. Properties of deterministic top down grammars. In *Proceedings of the first annual ACM symposium on Theory of computing*, pages 165–180. ACM, 1969.

[15] Michel F Sanner et al. Python: a programming language for software integration and development. *J Mol Graph Model*, 17(1):57–61, 1999.