UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Aleš Lekše

Generator sintaksnih analizatorjev vrste LALR v programskem jeziku PHP

DIPLOMSKO DELO
NA VISOKOŠOLSKEM ŠTUDIJU

Mentor: pred. dr. Boštjan Slivnik

Ljubljana, 2011

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00048/2010
Datum: 08.11.2010

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ALEŠ LEKŠE**

Naslov: **GENERATOR SINTAKSNIH ANALIZATORJEV VRSTE LALR V PROGRAMSKEM JEZIKU PHP**

LALR PARSER GENERATOR IN PHP

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

Sestavite generator sintaksnih generatorjev vrste LALR v programskem jeziku PHP. Generator naj za vhodno gramatiko izračuna tabelo vrste LALR in jo združi z algoritmom za sintaksno analizo vrste LR. Dobljeni sintaksni analizator vrste LALR, ki naj bo prav tako napisan v programskem jeziku PHP, naj bo sposoben opraviti sintaksno analizo in sestaviti sintaksno drevo analiziranega vhoda.

Mentor:
B. Slivnik
pred. dr. Boštjan Slivnik

Fakulteta za računalništvo in informatiko
2
UNIVERZA V LJUBLJANI

Dekan:
prof. dr. Nikolaj Zimic

I Z J A V A O A V T O R S T V U

diplomskega dela

Spodaj podpisani/-a ALEŠ LEKŠE,

z vpisno številko 63040446,

sem avtor/-ica diplomskega dela z naslovom:

Generator sintaksnih analizatorjev vrste LALR v programskem jeziku PHP

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom (naziv, ime in priimek) pred. dr. Boštjan Slivnik

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne 8.5.2011 Podpis avtorja/-ice:

Zahvala

Za pomoč pri nastajanju diplomske naloge se zahvaljujem mentorju, pred. dr. Boštjanu Slivniku.

Zahvaljujem se mami Štefki, očetu Jožetu ter sestri Ireni za vzpodbudo ter pomoč v času študija.

Zahvaljujem se tudi sošolcem (Juretu, Alešu) za študijsko pomoč ter prijetna druženja.

Kazalo

Povzetek

Obstaja veliko orodij za strojno prevajanje programskih jezikov, vendar ne za jezik PHP. Zato smo se odločili, da izvedemo generator, ki bo za poljubno gramatiko generiral sintaksni analizator.

Možnosti uporabe takšnega orodja je več, izpostavili pa smo uporabo sintaksnega analizatorja za prevedbo poizvedb SQL med različnimi dialekti ter kot pomoč pri generiranju poizvedb v modelu EAV.

Primerjali smo različne metode za sintaksno analizo glede na kompleksnost izvedbe, zmogljivost analiziranja v smislu zahtevnosti gramatike ter hitrosti izvajanja same analize. Odločili smo se, da izvedemo analizator tipa LALR saj se je izkazalo, da ima najboljše razmerje med zmogljivostjo in hitrostjo.

V nalogi smo podrobno predstavili postopek s katerim iz formalnega opisa gramatike izdelamo prevajalno tabelo ter postopek analize vhodnega programa.

Analizator smo uspešno spojili s knjižnico Zend_Db ter ji dodali zmožnost pretvorb določenih delov poizvedbe SQL iz standardnega jezika v dialekt katerega razume ciljni SUPB.

Abstract

Many tools for machine software translations exists, but not for PHP. Therefore we decided to implement such tool that will be capable of generating analyzer for any suitable grammar.

There are many usages for such tool, such is translation between different SQL dialects and as helper for generating queries in EAV model.

We compared different analysis methods in regard to implementation complexity, ability to analyze complex grammars and analysis performance. Decision was made to implement analyzer of type LALR which happens to have best ability/performance ratio.

Detailed procedures for creating such tool from formal grammar description will be presented in this thesis and analysis process.

We successfully merged this tool with Zend_Db library and extended it with the ability to transform some parts of SQL queries from standard to vendor specific dialect.

1 Uvod

Prevajalnik je program, čigar naloge pravzaprav ni težko opisati: na vhodu dobi zapis nekega programa v določenem programskem jeziku, njegov izhod pa je prevod programa v strojni jezik nekega računalnika (lahko tudi drug programski jezik). Namen prevajalnika je sprostiti programerja od navezave na strojni jezik računalnika in mu omogočiti, da programe piše v programskem jeziku, ki je prilagojen človeškemu načinu izražanja in razmišljanja.

V splošnem so prevajalniki sestavljenih iz sledečih delov [1]:

- leksikalna analiza: ima nalogo, da vhodne simbole (črke) združuje v večje pomenske enote, kot so npr. imena, ključne besede, številčne konstante in posebni simboli. Tem enotam pravimo osnovni simboli, ki kasneje v sintaksni analizi nastopajo kot končni simboli.
- sintaksna analiza: je naslednja stopnja, ki na podlagi formalnega opisa jezika ter vhodnega programa (zaporedja končnih simbolov) določi sintaktično strukturo programa
- semantična analiza: tu se sestavi neka notranja predstavitev vseh objektov, ki so prisotni v izvornem programu (spremenljivke, tipi, …) in na podlagi takšne predstavitve se opravijo določena preverjanja, ali se stavki izvornega programa pravilno podrejajo semantičnim pravilom izvornega jezika
- optimizacija: izvedba transformacij programa, ki ohranja njegov pomen (učinek), izboljša pa njegove lastnosti, kot so npr. čas izvajanja ali poraba pomnilnika
- generator strojne kode: zadnji korak je prevod programa v strojni jezik računalnika

V okviru diplomskega dela smo implementirali leksikalni in generator sintaksnih analizatorjev v programskem jeziku PHP.

V nadaljevanju je opisana možna uporaba analizatorja za premoščanje razlik med dialekti poizvedovalnega jezika SQL (structured query language) sistemov za upravljanje s podatkovnimi bazami (SUPB).

1.1 Razlike med dialekti poizvedovalnega jezika SQL

Kadar pišemo aplikacijo, ki uporablja podatke iz SUPB (večina spletnih aplikacij jo) je lahko nevarno (v smislu priklopa na ponudnika [2]) vezati aplikacijo na specifičnega proizvajalca (Oracle, IBM DB2, MSSQL, …) . Kljub temu, da je najpogostejši dostop do podatkov preko poizvedovalnega jezika SQL (ki naj bi bil prenosljiv) ter zanj obstoji standard (SQL-86 do trenutno aktualnega SQL:2008) se proizvajalci le-tega ne držijo v celoti. Idealno bi sicer bilo pisati ter uporabljati zgolj po standardih opisano sintakso ter funkcionalnosti, vendar žal proizvajalci tega pogosto ne podpirajo v celoti. Če želimo narediti aplikacijo prenosljivo (v smislu podprtih SUPB), nam ne preostane drugega, kot da dostop do nje dodatno abstrahiramo. Pogosto je dovolj, da je ta vmesna plast sorazmerno tanka ter preprosta, vendar se moramo ves čas razvoja zavedati, katere dialekte SQL bomo podpirali. Prav tako se je potrebno zavedati, da s širjenjem kroga podprtih SUPB lahko izgubimo določene funkcionalnosti (tudi lastniške), saj smo prisiljeni v uporabo zgolj tistih, ki jih podpirajo vsi.

S pomočjo analizatorja lahko izvedemo transformacije nad standardnimi poizvedbami SQL ter jih pretvorimo v dialekt, ki ga razume ciljni SUPB.

Spodaj bom v nekaj točkah opisal pogoste probleme, s katerimi se lahko srečamo pri razvoju aplikacij, ki morajo delovati z različnimi SUPB-ji:

- MySQL
- Oracle
- MS SQL
- IBM DB2

1.1.1 Ostranjevanje rezultatov poizvedbe SQL

Dober primer je ostranjevanje rezultatov poizvedbe. V namiznih aplikacijah imamo lahko kazalec na rezultat poizvedbe odprt ves čas delovanja aplikacije (ali vsaj med dvema dostopoma do podatkov). Izvedemo poizvedbo ter se s pomočjo kazalca pomikamo po rezultatu poizvedbe z namenom uporabniku prikazati zgolj del podatkov. Pri spletnih aplikacijah pa tega nimamo na voljo, saj protokol HTTP nima stanja. Zato je potrebno SUPB povedati, katero stran podatkov želimo.

Primer:

Iz tabele A želimo pridobiti prvih 10 zapisov urejenih po atributu B.

```
-- Izvorna poizvedba
SELECT * FROM A ORDER BY B;

-- Oracle
SELECT * FROM A ORDER BY B WHERE rownum<=10;

-- MySQL
SELECT * FROM A ORDER BY B LIMIT 10;

-- MSSQL
SELECT TOP 10 * FROM A ORDER BY B;

-- DB2
SELECT * FROM A ORDER BY B FETCH FIRST 10 ROWS ONLY;
```

Gornji primer kaže najpogostejši problem, s katerim se srečujemo pri prenosljivosti poizvedb SQL. Večina knjižnic podpira prenosljiv način definiranja ter izvedbe poizvedbe z ostranjevanjem.

1.1.2 Razlike med skalarnimi funkcijami

Funkcija POSITION [3] vrne položaj nekega niza v drugem nizu. Različni SUPB-ji imajo različne funkcije z različno sintakso, kar je razvidno spodaj:

```
-- ANSI SQL
SELECT POSITION('bar' IN 'foobar');

-- Oracle
SELECT INSTR('foobar', 'bar');

-- MSSQL
SELECT CHARINDEX('bar', 'foobar');

-- MySQL
SELECT POSITION('bar' IN 'foobar');

-- DB2
SELECT POSITION('bar', 'foobar') FROM SYSIBM.SYSDUMMY1;
```

Funkcija CONCATENATE združuje 2 ali več nizov v enega. V standardu je za združevanje predviden poseben operator, različni SUPB-ji pa imajo to izvedeno bodisi po standardu (Oracle, DB2), z drugim operatorjem (MSSQL) ali pa s posebno funkcijo (MySQL).

```
-- ANSI SQL
SELECT 'foo' || 'bar';

-- Oracle
SELECT 'foo' ||  'bar' FROM DUAL;

-- MSSQL
SELECT 'foo' + 'bar';

-- MySQL
SELECT CONCAT('foo', 'bar');

-- DB2
SELECT 'foo' || 'bar' FROM SYSIBM.SYSDUMMY1;
```

1.1.3 Razlikovanje malih ter velikih črk imen

Razlikovanje malih ter velikih črk (case sensitivity): različni SUPB na različne načine obravnavajo razlikovanje imen tabel in atributov.

- ANSI SQL pri imenih razlikuje male/velike črke. Če imena ne ovijemo v dvojne navednice ("), ga SUPB spremeni v velike črke, sicer ga pusti nespremenjenega.

```
"foo" != "Foo" != "FOO"
foo = Foo = FOO
```

- Oracle in DB2 se držita standarda.

- MSSQL ne razlikuje med malimi ter velikimi črkami.
- MySQL pri imenih atributov ne razlikuje med malimi ter velikimi črkami, dočim je obnašanje pri imenih tabel odvisno od operacijskega sistema oz. bolj natančno od datotečnega sistema:
  - MS Windows: ne razlikuje med velikimi ter malimi črkami;
  - GNU Linux/Unix: razlikuje med velikimi ter malimi črkami.

1.1.4 Vstavljanje in posodabljanje z enim ukazom

Ta funkcionalnost sicer ni del standarda, je pa zelo uporabna v primerih, kadar želimo v določeno tabelo vstaviti zapis, če ne obstaja, sicer pa posodobiti zgolj določene atribute (t.i. upsert).

```
-- MySql
INSERT INTO A (UniqueAttib, OtherAttrib) (1, 3) ON DUPLICATE
KEY UPDATE OtherAttrib =  OtherAttrib + VALUES(OtherAttrib);

-- Oracle/DB2
MERGE INTO A USING DUAL ON ( "UniqueAttib"=1 )
WHEN MATCHED THEN UPDATE SET "OtherAttrib"="OtherAttrib" + 3
WHEN NOT MATCHED THEN INSERT ("UniqueAttib","OtherAttrib")
    VALUES (1, 3);
```

1.1.5 Druge razlike

„Dual“ tabela: sistemska tabela brez stolpcev ter zgolj eno vrstico. Ime te tabele lahko uporabimo, kadar želimo dostopati do nekih konstant npr. trenutnega časa na strežniku, vrednosti nastavitev SUPB, ...

```
-- Oracle
SELECT 'foobar' AS COL FROM DUAL;

-- MSSQL
SELECT 'foobar' AS COL;

-- MySQL
SELECT 'foobar' AS COL;

-- DB2
SELECT 'foobar' AS COL FROM SYSIBM.SYSDUMMY1
```

1.2 Generiranje poizvedb v modelu EAV

Model EAV (entity attribute value [4]) je uporaben kadar imamo definiranih veliko različnih atributov, posamezne entitete pa uporabljajo zgolj manjšo podmnožico le teh. Vrednosti atributov hranimo v posebnih tabelah glede na podatkovni tip atributa v strukturi analogni razpršeni matriki. Ročna gradnja poizvedb v tem modelu je nadvse težavna, saj je potrebno osnovno entiteto kartezično povezati z vsemi entitetami, ki vsebujejo atribute. Gradnjo poizvedb v tem modelu si lahko olajšamo z generatorjem poizvedb, ki poskrbi za vse povezave ter vsa potrebna preimenovanja tabel in atributov.

Na poenostavljenem primeru:

Entiteta Osebe

Atribut	Opis
IdOsebe	Enolično ime osebe

Entiteta OsebeVarChar

Atribut	Opis
IdOsebe	Enolično ime osebe
Atribut	Ime polja osebe (npr. Ime, Priimek)
Vrednost	Vrednost atributa Atribut za vrstico IdOsebe

Entiteta OsebeInt

Atribut	Opis
IdOsebe	Enolično ime osebe
Atribut	Ime polja osebe (npr. Starost, Višina)
Vrednost	Vrednost atributa Atribut za vrstico IdOsebe

Primer ročno napisane poizvedbe:

```
// ta poizvedba nam vrne vse osebe višje od 180 cm
// skupaj z enoličnim imenom, imenom in višino
SELECT
    O.IdOsebe,
    A1.Vrednost Ime,
    A2.Vrednost Visina
FROM
```

```
    Osebe O
LEFT JOIN
    OsebeVarChar A1
ON
    A1.IdOsebe = O.IdOsebe AND A1.Atribut = 'Ime'
LEFT JOIN
    OsebeInt A2
ON
    A2.IdOsebe = O.IdOsebe AND A2.Atribut = 'Visina'
WHERE
    A2.Atribut>180;
```

Z analizatorjem bi lahko prevedli poizvedbo nad navidezno tabelo v poizvedbo v kateri bi bile vse potrebne tabele že kartezično povezane.

Z uporabo hipotetičnega generatorja bi zgolj napisali naslednje:

```
// spodnji primer, bi generiral poizvedbo ekvivalentno
// zgornji
$eav->select()
    ->from(
         'Osebe', // ime relacije
         array('Ime', 'Visina') // atributi
    )
    ->where('Visina>180'); // pogoj
```

2 Leksikalna in sintaksna analiza

2.1 Leksikalna analiza

Leksikalni analizator deluje kot končni avtomat, pri čemer bere znake iz vhodnega zaporedja in jih združuje v osnovne simbole jezika.

Implementacija je osnovana na uporabi regularnih izrazov za prepoznavanje osnovnega simbola ter njegove vrednosti. Načeloma je lahko "ročno" zgrajen analizator hitrejši, saj je prepoznavanje regularnih izrazov sorazmerno potratno, vendar če se zavedamo, da so funkcije regularnih izrazov v jeziku PHP izvedene v jeziku C in se ne interpretirajo, je implementacija posledično hitrejša.

V izogib dopolnjevanju analizatorja ob vsaki spremembi gramatike (npr. dodatku novega operatorja) je kot vhodni parameter podan poleg samega programa tudi gramatika. To nam omogoča, da lahko preproste končne simbole dodajamo v gramatiko, ne da bi potrebovali spremeniti leksikalni analizator.

Osnovni leksikalni simboli so razdeljeni v razrede, ki se jih prepoznava v sledečem vrstnem redu:

1. preprosti končni simboli (razni operatorji, vsem je skupna konstantna vrednost): prioritetno se jih prepoznava glede na njihovo dolžino (daljše pred krajšimi);
2. simboli, ki nimajo nobenega vpliva na program (t.i. beli simboli – whitespaces);
3. manj preprosti končni simboli: kot so npr. besedilne in numerične konstante, imena, … (vsak izmed njih je definiran z enim ali več regularnih izrazov).

Primeri regularnih izrazov (v narečju PERL):

- Besedilne konstante:

```
// C
/^"(?<value>(?:[^"\\]|\\.)*)"/

// ANSI SQL
/^"(?<value>(?:[^"]|"")*)"/
```

- Numerične konstante:

```
/^(?<value>\d+(?:\.\d+)?(?:e-?\d+)?)/
```

- Imena

```
/^[a-z]\w*/i
```

2.2 Sintaksna analiza

Primerjali smo tri različne analizatorje (CYK, LL(1) ter LALR) glede na stopnjo zahtevnosti implementacije ter hitrostjo analiziranja.

Za začetek nekaj pojasnil:

- v gramatikah bodo z velikimi črkami označeni vmesni simboli, simbol S bo predstavljal začetni simbol;
- z majhnimi črkami ali z drugimi znaki kot so + in - bodo označeni končni simboli;
- simbol ε predstavlja prazno produkcijo;
- simbol $ predstavlja konec besede;
- majhne grške črke (npr. α, β) predstavljajo stavčne oblike sestavljene iz vmesnih in/ali končnih simbolov, lahko tudi dolžine nič.

Zaradi očitnih slabosti CYK in LL(1) analizatorja smo se na koncu odločili implementirati LALR analizator. V naslednjih točkah bodo vsi trije na kratko opisani.

2.3 Cocke–Younger–Kasami (CYK)

CYK je algoritem, s katerim lahko preverimo ali beseda ustreza kontekstno neodvisni gramatiki in kako je zgrajena. Obstaja razširitev osnovnega algoritma, s katero lahko zgradimo tudi abstraktno sintaksno drevo.

Sam algoritem je sorazmerno preprost za implementacijo, vendar ni pretirano hiter. Najslabši čas izvajanja ima sledečo časovno zahtevnost: $O(n^3 * |G|)$, kjer je n dolžina besede ter |G| velikost gramatike [5]. Ima tudi omejitev glede gramatike in sicer, da morajo biti vse produkcije oblike A->B C ali D -> a (kjer so A, B, C in D vmesni ali začetni simboli ter b končni simbol).

Postopek analize je sledeč:

1. Pripravimo matriko velikosti n+1 * n kjer je n dolžina vhodne besede (stolpce številčimo od 1 naprej, vrstice pa od 0 naprej).
2. V vrstico 0 vpišemo simbole vhodne besede.
3. V polja vrstice 1 vpišemo simbol A, če obstaja produkcija oblike A->a, kjer je a končni simbol nad poljem (torej simbole iz katerih lahko neposredno izpeljemo končne simbole).
4. Množica simbolov {C} v polju $P_{i, j}$, kjer je i številka vrstice in j številka stolpca ter velja $i > 1$ in $j \leq n - i + 1$, je določena tako, da za vsak k med 1 in i - 1 poiščemo v tabeli vse kombinacije simbolov A', B' v poljih $P_{k, j}$ in $P_{i-k, j+k}$, za katere obstaja produkcija oblike C -> A' B'.

Primer:

Gramatika:

```
S → S E'
  | T T'
```

```
  | n
E' → P T
T → T T'
  | n
T' → R Q
P → +
R → *
Q → n
```

Vhodna beseda:

```
2 + 4 * 3
```

Z leksikalno analizo prevedemo vhodno besedo 2 + 4 * 3 v niz končnih simbolov:

```
n + n * n
```

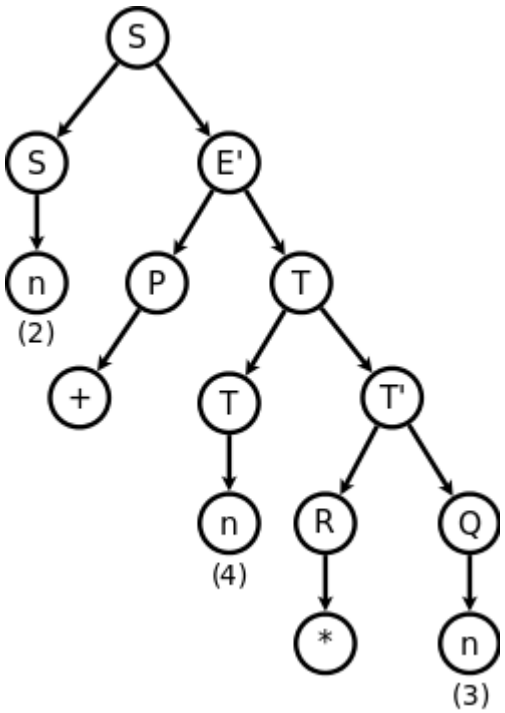
Analiza:

	1	2	3	4	5
0	n	+	n	*	n
1	S,T,Q	P	S,T,Q	R	S,T,Q
2	/	E'	/	T'	
3	S	/	S,T		
4	/	E'			
5	S				

Tabela 1: Analiza CYK

Simbol S je v polju $P_{3,1}$, ker obstaja kombinacija simbolov S E' iz polja $P_{1,1}$ ter polja $P_{2,2}$, ki se izpelje v simbol S iz produkcije S -> S E'.

Sintaksno drevo je prikazano na sliki 1.



*Slika 1: Drevo izpeljave besede n + n * n.*

2.4 LL(1) analiza

LL(1) analiza bere vhod od leve proti desni in deterministično sestavi skrajno levo izpeljavo vhodne besede [1]. Besedo analizira od zgoraj navzdol, tj. začne z začetnim simbolom ter ga postopoma razgrajuje do končnih simbolov.

LL(1) analizator spada v družino analizatorjev LL(k), kjer k predstavlja število simbolov, ki jih LL(1) analizator hrani v vhodnem oknu. Z višanjem števila k se sicer širi obseg podprtih gramatik v zameno za kompleksnejšo izvedbo.

Prednosti LL analize so predvsem hitra konstrukcija ter lažje razumevanje same analize, saj je bližje človeškemu razmišljanju, slabosti pa so:

- leva izpeljava, zaradi katere postanejo računski operatorji implicitno desno asociativni,
- ker leva rekurzija ni mogoča, postane sama gramatika slabše razumljiva (za odpravo je potrebno vpeljati vmesno produkcijo).

Omejitve LL(1) gramatik so sledeče:

- Nasprotje FIRST/FIRST nastane, kadar za poljuben par produkcij, ki imajo na levi strani isti simbol, A -> α in A -> β velja, da je presek množic FIRST(α) in FIRST(β) neprazna množica.

- Nasprotje FIRST/FOLLOW nastane, kadar za poljuben par produkcij, ki imajo na levi strani isti simbol, A -> α in A -> β velja, da je presek FIRST(α) ter FOLLOW(A), pri čemer FIRST(β) vsebuje ε, neprazna množica.
- Leva rekurzija: produkcije oblike A -> Aα vedno povzročijo FIRST/FIRST konflikte, razen če ne bi obstajala nobena druga produkcija z enako levo stranjo.

Primer gramatike z nasprotjem FIRST/FIRST:

```
S → A
A → a b
  | a c

FIRST(A → a b) = {a}
FIRST(A → a c) = {a}
```

Primer gramatike z nasprotjem FIRST/FOLLOW:

```
S → A B
A → a
  | ε
B → A a

FIRST(A → a) = {a}
FIRST(A → ε) = {ε} in FOLLOW(A) = {a}
```

Gradnja prevajalne tabele je srednje zahtevna, hitrost analiziranja pa neprimerno višja kot pri CYK algoritmu. Če želimo, lahko po končani analizi popravimo asociativnost računskih operatorjev z transformacijo drevesa.

Tabela je zgrajena iz stolpcev, ki so označeni s končnimi simboli (pri k=1) in iz vrstic, ki so označene z vmesnimi simboli. Tabelo zgradimo tako, da za vse vmesne simbole A ter vse končne simbole b poiščemo vse produkcije oblike A -> α pri katerih bodisi FIRST(α) vsebuje b ali FIRST(α) vsebuje ε ter FOLLOW(A) vsebuje b. V tabeli v presečišče stolpca označenega s simbolom b ter vrstice označene z simbolom A zapišemo produkcijo A-> α. Če v neko polje zapišemo več kot eno produkcijo, pomeni da smo naleteli na FIRST/FIRST ali FIRST/FOLLOW konflikt. Prazna polja so dovoljena.

Opis določevanja množic FIRST in FOLLOW bo opisan v točki 3.4. ter 3.5.

Primer:

Gramatika:

```
S → E
```

```
E → T G
G → + T G
  | ε
T → n U
U → * n U
  | ε
```

Množice FIRST in FOLLOW:

```
FIRST(S) = {n}          FOLLOW(S) = {$}
FIRST(E) = {n}          FOLLOW(E) = {$}
FIRST(G) = {+, ε}       FOLLOW(G) = {$}
FIRST(T) = {n}          FOLLOW(T) = {$, +}
FIRST(U) = {*, ε}       FOLLOW(U) = {$, +}
```

	+	**n**	*	**$**
S		E		
E		T G		
G	+ T G			ε
T		n U		
U	ε		* n U	ε

Tabela 2: Zgrajena LL(1) tabela

Analiza se vrši po sledečem postopku:

1. Na sklad damo začetni simbol S.
2. Iz vhoda preberemo prvi simbol b.
3. Iz sklada vzamemo simbol A; če ta ne obstaja, beseda ne ustreza gramatiki.
4. Če je simbol A enak simbolu b, potem iz vhoda preberemo nov simbol b ter nadaljujemo s točko 3.
5. V tabeli poiščemo produkcijo na presečišču stolpca označenega s simbolom b ter vrstice označene s simbolom A.
6. Na sklad damo v obratnem vrstnem redu vse simbole v produkciji (celotno desno stran), če je produkcija različna od ε.
7. Nadaljujemo s točko 3, dokler sklad ni prazen.

Spodaj bo predstavljen primer analize preproste besede, pri čemer sledimo gornjim pravilom. Tabela za vsak korak kaže vrednosti na skladu, del še neprebrane besede in trenutno izpeljavo besede.

Vhodna beseda:

```
2 + 4 * 3
```

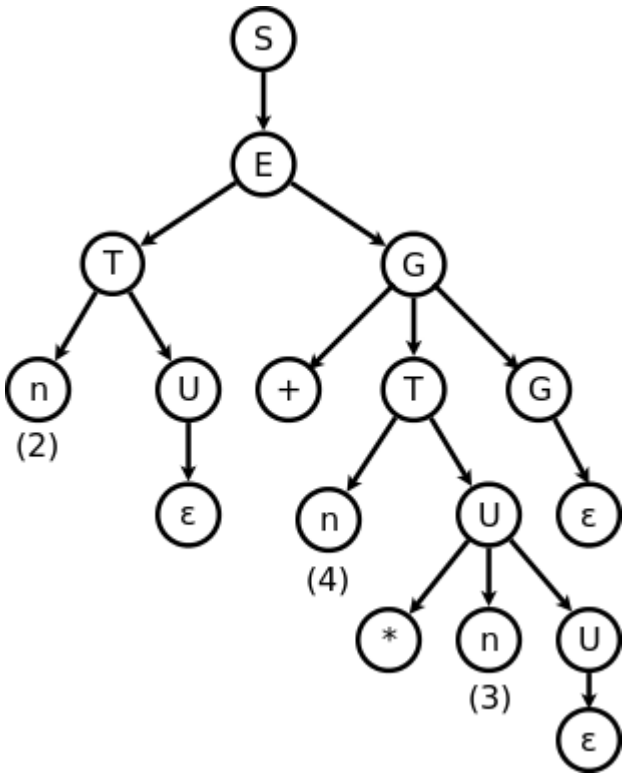
Z leksikalno analizo prevedemo vhodno besedo 2 + 4 * 3 v niz končnih simbolov:

```
n + n * n $
```

Postopek izpeljave:

Sklad	Beseda	Izpeljava
S	n + n * n $	S
E	n + n * n $	S(E)
G, T	n + n * n $	S(E(T G))
G, U, n	n + n * n $	S(E(T(n U) G))
G, U	+ n * n $	S(E(T(n U) G))
G	+ n * n $	S(E(T(n U()) G))
G T +	+ n * n $	S(E(T(n U()) G(+ T G)))
G T	n * n $	S(E(T(n U()) G(+ T G)))
G U n	n * n $	S(E(T(n U()) G(+ T(n U) G)))
G U	* n $	S(E(T(n U()) G(+ T(n U) G)))
G U n *	* n $	S(E(T(n U()) G(+ T(n U(* n U)) G)))
G U n	n $	S(E(T(n U()) G(+ T(n U(* n U)) G)))
G U	$	S(E(T(n U()) G(+ T(n U(* n U)) G)))
G	$	S(E(T(n U()) G(+ T(n U(* n U())) G)))
	$	S(E(T(n U()) G(+ T(n U(* n U())) G())))

*Tabela 3: Izpeljava besede n + n * n*



*Slika 2: Drevo izpeljave besede n + n * n.*

2.5 LALR analiza

LALR (lookahead, left to right) je sintaksni analizator vrste LR, s katerim lahko analiziramo podrazred kontekstno neodvisnih gramatik. LR analizatorji so zaradi velike moči in hitrosti prevajanja sila priljubljeni [6]. Osnovna ideja LR analize je preprosta: v besedi, ki jo želimo analizirati in ki ni nujno sestavljena le iz končnih simbolov, temveč je v splošnem neka stavčna oblika gramatike, poiščemo tisto podbesedo, ki je nastala v zadnjem koraku desne izpeljave. Ko jo odkrijemo, jo zamenjamo z levo stranjo ustrezne produkcije in postopek nadaljujemo, dokler nismo prvotne stavčne oblike prevedli na začetni simbol [1]. Rezultat je abstraktno sintaksno drevo z skrajno desno izpeljavo.

Kot že omenjeno spadajo LALR analizatorji v družino LR(k) analizatorjev, kjer k predstavlja število simbolov, ki jih mora analizator prebrati v naprej. Med pogostejšimi so:

- LR(0): omogoča analizo zgolj preprostih gramatik, najpreprostejši za izvedbo.
- LR(1): zmogljiv analizator, ki zmore analizirati širok razred gramatik, a je zaradi velikega števila stanj že pri sorazmerno preprostih gramatikah neuporaben v praksi.
- LALR(1): skoraj tako zmogljiv kot LR(1) vendar z bistvo manj stanji, zelo pogosta uporaba v praksi (npr. yacc [7] in bison [8]).

Grobo lahko razmerje med zgoraj opisanimi analizatorji zapišemo kot (v smislu širine razreda gramatik, ki jih lahko analizirajo): LR(0) < LALR(1) < LR(1).

Implementacija je zahtevna (dosti zahtevnejša od LL(1)), ima pa ta algoritem več prednosti pred LL(1) in sicer:

- leva rekurzija je dovoljena, kar poenostavi opis gramatike;
- skrajno desna izpeljava, kar nam pride prav, saj se asociativnost računskih operatorjev ohrani.

Analiza se vrši s pomočjo prevajalne tabele, ki jo zgradimo s pomočjo funkcij FIRST, FOLLOW in množice stanj LR(k).

Primer (postopek gradnje je podrobno opisan v poglavju 3):

Gramatika:

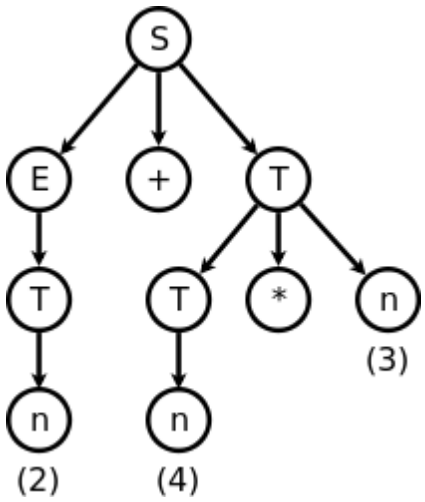
```
S → E
E → E + T
  | T
T → T * n
  | n
```

	Action				Goto		
Stanje	$	+	*	n	S	E	T
0				S1		2	3
1	R4	R4	R4				
2	ACCEPT	S4					
3	R2	R2	S5				
4				S1			6
5				S7			
6	R1	R1	S5				
7	R3	R3	R3				

Tabela 4: Zgrajena LALR tabela.

Stavek:

```
2 + 4 * 3
```



*Slika 3: Drevo izpeljave besede n + n * n.*

3 Generator LALR

LALR analiza stavka se vrši s pomočjo posebne prevajalne tabele, ki usmerja analizator. Izvedba samega analizatorja je sorazmerno preprosta, medtem ko konstrukcija te tabele žal ni.

Načinov gradnje LALR prevajalne tabele je več [9]:

- Osnovna ideja: zgradimo vse množice LR(1) ter nato združimo množice z enakim jedrom.
- Preprost algoritem: množice z enakim jedrom združimo že ob gradnji.
- "kanalni" algoritem, ki ga uporablja yacc [9].
- Gradnja LALR(1) kot nadgradnja LR(0).
- LALR(1) preko SLR(1) [9], [10].

Zaradi relativne enostavnosti smo se odločili, da implementiramo zadnji algoritem, ki v grobem sestoji iz sledečih korakov, ki bodo bolj podrobno opisani v nadaljevanju:

1. gradnja množice LR(0) stanj;
2. gradnja razširjene gramatike;
3. množice FIRST in FOLLOW;
4. določevanje prevedb;
5. gradnja tabele s pomočjo tabele prehodov ter prevedb.

Za osnovo smo vzeli gramatiko iz točke 3.2.c, ki smo jo nadgradili z združevanjem operacij z oklepaji ter dodali vse osnovne računske operatorje.

Gramatika:

```
0     S -> E
1     E -> E A T
2        | T
3     T -> T M F
4        | F
5     F -> ( E )
6        | n
7        | - F
8     A -> +
9        | -
10    M -> *
11       | /
```

Zgornji zapis predstavlja formalen zapis sintakse, pri čemer so:

- končni simboli (ne nastopajo na levi strani)
  - n število
  - +, -, *, / računski operatorji

    - (,) simbola s katerim so združene operacije
- vmesni simboli (nastopajo tudi na levi strani)
    - S ciljni simbol
    - E operacija prištevanja in odštevanja ali T
    - T operacija z množenjem in deljenjem ali F
    - F faktor, število, z oklepaji združene operacije ali negiran faktor
    - A operator za seštevanje in odštevanje
    - M operator za množenje in deljenje

Sama hierarhija produkcij zagotavlja, da se prioritete in asociativnost računskih operatorjev ohranijo.

3.1 Množica LR(0) stanj

Prvi korak pri gradnji tabele je določevanje množic LR(0). LR(0) množice predstavljajo vse možne prehode med produkcijami, glede na trenutni simbol.

Pri gradnji produkcija ni več definirana zgolj z simboli, temveč še z kazalcem na simbol ki sledi. Znak, ki to označuje je ·. Na primeru:

Naslednji simbol je C:

```
A -> B ·C D
```

Pravkar smo obiskali simbol D – produkcija je izčrpana:

```
A -> B C D·
```

Vsak element množice LR(0) stanj je sestavljen iz dveh delov: jedra in telesa. Jedro sestavljajo produkcije iz stanja iz katerega smo izpeljali to stanje, telo pa iz izpeljanih produkcij. Produkcijo izpeljemo tako, da dodamo produkcijo, na katero kaže kazalec, ter vse produkcije, na katere kažejo dodane produkcije rekurzivno dokler niso dodane vse produkcije in/ali vsi kazalci kažejo na končne simbole.

Gradnjo začnemo z stanjem I_0. Jedro stanja je produkcija začetnega simbola: S -> E. Kazalec postavimo pred prvi simbol.

Stanje I_0:

```
S -> ·E
+ E -> ·E A T
+ E -> ·T
+ T -> ·T M F
+ T -> ·F
```

```
+ F -> ·( E )
+ F -> ·n
+ F -> ·- F
```

Vsa naslednja stanja zgradimo tako, da za vsak simbol ali končni simbol poiščemo vse produkcije v prejšnjem stanju (I_0) katere kazalci kažejo na ta simbol. Za vsak simbol za katerega obstaja ena ali več takšnih produkcij ustvarimo novo stanje – v produkcijah prestavimo kazalec na naslednji simbol. Te produkcije tvorijo jedro novo ustvarjenih stanj. Telesa novih stanj dobimo z izpeljavo produkcij iz jedra.

Stanje I_1 (iz I_0 z uporabo simbola '('):

```
F -> ( ·E )
+ E -> ·E A T
+ E -> ·T
+ T -> ·T M F
+ T -> ·F
+ F -> ·( E )
+ F -> ·n
+ F -> ·- F
```

Stanje I_2 (iz I_0 z uporabo simbola n):

```
F -> n·
```

Stanje I_3 (iz I_0 z uporabo simbola -):

```
F -> - ·F
+ F -> ·( E )
+ F -> ·n
+ F -> ·- F
```

Stanje I_4 (iz I_0 z uporabo simbola E):

```
S -> E·
E -> E· A T
+ A -> ·+
+ A -> ·-
```

Stanje I_5 (iz I_0 z uporabo simbola T):

```
E -> T·
```

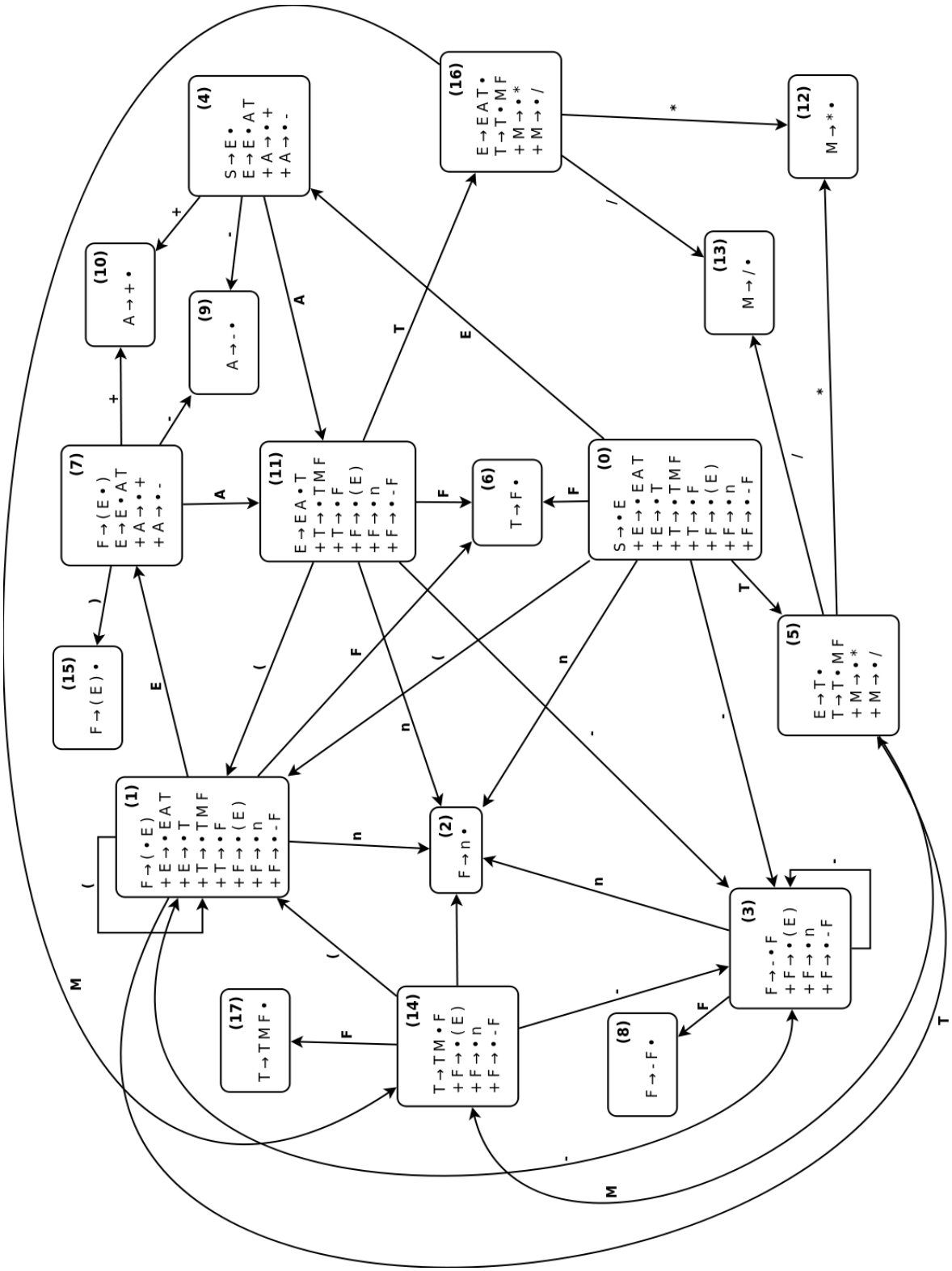
```
T -> T· M F
+ M -> ·*
+ M -> ·/
```

Stanje I_6 (iz I_0 z uporabo simbola F):

```
T -> F·
```

Ustvarili smo 6 stanj iz stanja I_0 ter izpeljali vse njegove naslednike. Sedaj postopek nadaljujemo nad stanji I_1 – I_6.

Po končanem postopku dobimo 17 stanj. Prehode med njimi si lahko ogledamo na sliki 4.



Slika 4: Prehodi med stanji

3.2 Tabela prehodov

Množica LR(0) stanj, katere smo generirali v prejšnji točki, so osnova za gradnjo tabele prehodov med stanji. Prehod med stanjem I_A in iz njega izpeljanim stanjem I_B je tisti simbol S, katerega smo uporabili za izpeljavo.

Iz slike povezav iz prejšnje točke je npr. razvidno, da iz stanja I_0 preidemo v stanje I_4 preko simbola E (viden na povezavi med stanjema).

Množica	(	)	n	-	+	*	/	S	E	T	F	A	M
0	1		2	3					4	5	6		
1	1		2	3					7	5	6		
2													
3	1		2	3							8		
4				9	10							11	
5						12	13						14
6													
7		15		9	10							11	
8													
9													
10													
11	1		2	3						16	6		
12													
13													
14	1		2	3							17		
15													
16						12	13						14
17													

Tabela 5: Tabela prehodov

3.3 Razširjena gramatika

Naslednji korak je generiranje razširjene gramatike. Tukaj vsak simbol v produkciji opremimo z številko stanja, iz katere izhaja p, ter številko naslednjega stanja n, torej $_pS_n$. To storimo tako, da se sprehodimo čez produkcije vseh stanj, pri katerih kazalec kaže na prvi simbol produkcije. V produkciji A -> ·B C iz stanja I_p označimo simbol A z številko stanja I_p ter številko stanja I_a v katerega prehaja stanje I_p preko simbola A – $_pA_a$. Simbol B označimo z številko stanja I_p ter številko stanja I_b v katerega prehaja stanje I_p preko simbola B – $_pB_b$. Vse nadaljnje simbole G označimo z številko ciljnega stanja prejšnjega simbola ter številko stanja v katerega prehaja ciljno stanje prejšnjega simbola preko G. Torej simbol C označimo z $_bC_c$, kjer je c številka stanja v katerega prehaja stanje I_b preko simbola C.

Tukaj si seveda pomagamo z tabelo prehodov iz prejšnje točke za določevanjem ciljnih stanj.

Za primer vzemimo stanje I_3:

Množica I_3

```
F -> - ·F
+ F -> ·( E )
+ F -> ·n
+ F -> ·- F
```

Produkcije `F -> - ·F` ne upoštevamo, saj kazalec ne kaže na prvi simbol.

Nadaljujemo z naslednjo produkcijo `F -> ·( E )`. Simbol F označimo z številko stanja I_3 ter z številko stanja I_8 v katerega stanje I_3 prehaja preko simbola F. Simbol (zopet označimo z številko stanja I_3 ter številko stanja I_1 v katerega prehaja stanje I_3 preko simbola (. Naslednji simbol E označimo z številko stanja I_1 (torej ciljnega stanja prejšnjega simbola) ter številko stanja I_7 v katerega prehaja stanje I_1 preko simbola E. Naslednji simbol) označimo z številko stanja I_7 ter številko stanja I_{15} v katerega prehaja stanje I_7 preko simbola). Izjema je začetni simbol S katerega označimo z $_0S_\$$. $ označuje konec stavka.

$$_3F_8 \rightarrow {}_3(_1\ {}_1E_7\ {}_7)_{15}$$

Postopek nadaljujemo s produkcijama `F -> ·n` ter `F -> ·- F` ter seveda s produkcijami ostalih stanj.

Končni rezultat je razširjena gramatika:

$$_0S_\$ \rightarrow {}_0E_4$$
$$_0E_4 \rightarrow {}_0E_4\ {}_4A_{11}\ {}_{11}T_{16}$$
$$_0E_4 \rightarrow {}_0T_5$$
$$_0T_5 \rightarrow {}_0T_5\ {}_5M_{14}\ {}_{14}F_{17}$$
$$_0T_5 \rightarrow {}_0F_6$$
$$_0F_6 \rightarrow {}_0(_1\ {}_1E_7\ {}_7)_{15}$$
$$_0F_6 \rightarrow {}_0n_2$$

```
0F6 -> 0-3 3F8
1E7 -> 1E7 7A11 11T16
1E7 -> 1T5
1T5 -> 1T5 5M14 14F17
1T5 -> 1F6
1F6 -> 1(1 1E7 7)15
1F6 -> 1n2
1F6 -> 1-3 3F8
3F8 -> 3(1 1E7 7)15
3F8 -> 3n2
3F8 -> 3-3 3F8
4A11 -> 4+10
4A11 -> 4-9
5M14 -> 5*12
5M14 -> 5/13
7A11 -> 7+10
7A11 -> 7-9
11T16 -> 11T16 16M14 14F17
11T16 -> 11F6
11F6 -> 11(1 1E7 7)15
11F6 -> 11n2
11F6 -> 11-3 3F8
14F17 -> 14(1 1E7 7)15
14F17 -> 14n2
14F17 -> 14-3 3F8
16M14 -> 16*12
16M14 -> 16/13
```

3.4 Množice FIRST

Množica FIRST produkcije A -> α je seznam končnih simbolov a, za katere obstajajo besede oblike aβ, ki se jih izpelje iz α. α in β so poljubne besede sestavljene iz vmesnih in končnih simbolov.

Pravila za gradnjo [11]:

1. FIRST(x) = {x}, če je x končni simbol.
2. Če obstaja produkcija X -> ε, potem je ε vsebovan v FIRST(X). ε predstavlja prazno produkcijo.
3. Če obstaja produkcija X -> α, je FIRST(α) vsebovan v FIRST(X).
4. FIRST(Y α) = FIRST(Y), če Y ne vsebuje ε, sicer je FIRST(Y α) = FIRST(Y) U FIRST(α). Če tudi α vsebuje ε potem tudi FIRST(Y α) vsebuje ε.

Množice FIRST gradimo iz razširjene gramatike. Spodaj je prikazan primer gradnje za $_0T_5$.

Produkcija

```
0T5  -> 0T5  5M14  14F17
0T5  -> 0F6
```

FIRST($_0T_5$) bo vseboval FIRST($_0T_5$ $_5M_{14}$ $_{14}F_{17}$) ter FIRST($_0F_6$). Ker $_0T_5$ ne vsebuje ε, postane FIRST($_0T_5$) = FIRST($_0T_5$) + FIRST($_0F_6$) oz. FIRST($_0T_5$) = FIRST($_0F_6$), saj vsak končni simbol šteje zgolj enkrat. Sedaj nadaljujemo izračun z določitvijo FIRST($_0F_6$).

```
0F6  -> 0(1  1E7  7)15
0F6  -> 0n2
0F6  -> 0-3  3F8
```

FIRST($_0F_6$) vsebuje FIRST($_0(_1$ $_1E_7$ $_7)_{15}$) + FIRST($_0n_2$) + FIRST($_0$-$_3$ $_3F_8$). Ker je v vseh treh produkcijah na prvem mestu končni simbol (in ne simbol, ki bi kazal na prazno produkcijo), je FIRST($_0F_6$) = {(, n, -}.

Torej je FIRST($_0T_5$) = {(, n, -}.

Če ponovimo postopek za vse simbole pridemo do sledečih množic FIRST:

```
FIRST(4A11) = {+, -}
FIRST(5M14) = {*, /}
FIRST(0F6) = {(, n, -}
FIRST(0T5) = {(, n, -}
FIRST(0E4) = {(, n, -}
FIRST(0S$) = {(, n, -}
FIRST(7A11) = {+, -}
FIRST(1F6) = {(, n, -}
FIRST(1T5) = {(, n, -}
FIRST(1E7) = {(, n, -}
FIRST(3F8) = {(, n, -}
FIRST(16M14) = {*, /}
FIRST(11F6) = {(, n, -}
FIRST(11T16) = {(, n, -}
FIRST(14F17) = {(, n, -}
```

3.5 Množice FOLLOW

Množica FOLLOW(A) simbola A je množica vseh končnih simbolov a, za katere se lahko iz začetnega simbola S izpelje beseda oblike αAaβ, kjer sta α in β poljubni (lahko tudi prazni)

besedi sestavljeni iz vmesnih in končnih simbolov.

Pravila gradnje [11]:

1. Konec vhoda $ je vsebovan v FOLLOW(S).
2. Če obstaja produkcija oblike A -> αBβ, kjer je α poljubna beseda, β pa neprazna beseda, potem je vse v FIRST(β) brez ε tudi v FOLLOW(B).
3. Če obstaja produkcija oblike A -> αB, potem je vse v FOLLOW(A) tudi v FOLLOW(B) [1].
4. Če obstaja produkcija oblike A -> αBβ in FIRST(β) vsebuje ε, potem je vse v FOLLOW(A) vsebovano tudi v FOLLOW(B).
5. FOLLOW(b) kjer je b končni simbol je prazna množica.

Množice FOLLOW gradimo iz razširjene gramatike. To je tudi bistvena razlika med LALR preko SLR od algoritma SLR, saj se večina konteksta ohrani.

Spodaj je prikazan primer gradnje za ${}_0T_5$.

Sprehodimo se čez vse produkcije, ki vsebujejo v telesu simbol ${}_0T_5$:

```
*    0E4 -> 0T5
**   0T5 -> 0T5 5M14 14F17
```

Pravila št. 2 na produkciji * ne moremo uporabiti, saj produkcija ni primerne oblike. Lahko pa upoštevamo pravilo št. 3, iz katerega sledi FOLLOW(${}_0T_5$) vsebuje FOLLOW(${}_0E_4$).

Pravila št. 3 na produkciji ** ne moremo uporabiti, lahko pa upoštevamo pravilo 2 iz katerega sledi: FOLLOW(${}_0T_5$) vsebuje FIRST(${}_5M_{14}\ {}_{14}F_{17}$) = FIRST(${}_5M_{14}$) = {*, /} (${}_5M_{14}$ ne vsebuje ε).

Nadaljujemo z izračunom FOLLOW(${}_0E_4$). Produkcije, ki vsebujejo v telesu simbol ${}_0E_4$, so:

```
***    0E4 -> 0E4 4A11 11T16
****   0S$ -> 0E4
```

Če upoštevamo pravilo št. 2 na produkciji ***, vidimo da FOLLOW(${}_0E_4$) vsebuje FIRST(${}_4A_{11}\ {}_{11}T_{16}$) = FIRST(${}_4A_{11}$) = {+, -} . Če upoštevamo pravilo št. 3 na produkciji ****, pa dobimo, da FOLLOW(${}_0E_4$) vsebuje FOLLOW(${}_0S_\$$), ki pa po pravilu št. 1 vsebuje $.

Ko zberemo skupaj vse simbole, dobimo sledečo množico FOLLOW:

1 Pravila 3. v resnici ne potrebujemo, saj je vsebovano v pravilu 4 (FIRST({}) = {ε}).

```
FOLLOW(0T5) = {*, /, +, -, $}
```

Če postopek ponovimo za vse simbole, dobimo sledeče množice:

```
FOLLOW(0S$) = {$}
FOLLOW(0E4) = {$, +, -}
FOLLOW(0T5) = {$, +, -, *, /}
FOLLOW(0F6) = {$, +, -, *, /}
FOLLOW(1E7) = {), +, -}
FOLLOW(1T5) = {), +, -, *, /}
FOLLOW(1F6) = {), +, -, *, /}
FOLLOW(11T16) = {$, +, -, ), *, /}
FOLLOW(11F6) = {$, +, -, ), *, /}
FOLLOW(14F17) = {$, +, -, *, /, )}
FOLLOW(3F8) = {$, +, -, *, /, )}
FOLLOW(4A11) = {(, n, -}
FOLLOW(5M14) = {(, n, -}
FOLLOW(7A11) = {(, n, -}
FOLLOW(16M14) = {(, n, -}
```

Pri gradnji množic FOLLOW smo naleteli na zanimiv problem krožne povezanosti, katerega bomo predstavili spodaj.

Hipotetični primer 3 krožno povezanih produkcij:

```
1. A -> aB
2. B -> bC
3. C -> cA
```

+ 3 dodatne produkcije

```
4. D -> Aa
5. D -> Bb
6. D -> Cc
```

Iz česar sledi:

```
a ∈ FOLLOW(A)
b ∈ FOLLOW(B)
c ∈ FOLLOW(C)

FOLLOW(B)⊂ FOLLOW(A)
FOLLOW(C)⊂ FOLLOW(B)
FOLLOW(A)⊂ FOLLOW(C)
```

Težava seveda nastopi, ko želimo izračunati množice FOLLOW za simbole A, B in C, saj so ti med seboj krožno povezani – FOLLOW(A) ne moremo izračunati, ne da bi določili FOLLOW(C), katerega zopet ne moremo določiti, ne da bi določili FOLLOW(B). Ko želimo določiti FOLLOW(B), pa naletimo na težavo, saj je le-ta odvisen od FOLOW(A), katerega smo želeli določiti na prvem mestu.

Težavo smo rešili z preprostim algoritmom, ki postopoma razpršuje FOLOW simbole:

Grob opis:

1. Za vsak simbol A določi, katere končne simbole zagotovo vsebuje množica FOLLOW, ter jih razprši po vseh produkcijah, katere množice FOLLOW vsebujejo FOLLOW(A).
2. Če je bil v prejšnjem koraku v katerokoli množico FOLLOW dodan kakšen končni simbol, nadaljuj postopek s točko 1.

Opis delovanje na zgornjem primeru:

1. korak: FOLLOW(A) zagotovo vsebuje a, FOLLOW(B) zagotovo vsebuje b ter FOLLOW(C) zagotovo vsebuje c. Ker je FOLLOW(A) vsebovan v FOLLOW(B), lahko dodamo simbol a v slednjega ter analogno simbol b v FOLLOW(C) ter c v FOLLOW(A).

```
FOLLOW(A)={a, c}
FOLLOW(B)={b, a}
FOLLOW(C)={c, b}
```

2. korak: ponovimo korak 1

```
FOLLOW(A)={a, c, b}
FOLLOW(B)={b, a, c}
FOLLOW(C)={c, b, a}
```

3. korak: ponovimo korak 1, ker nobena množica ne pridobi novega končnega simbola, je postopek končan – simboli so razpršeni po vseh množicah FOLLOW.

3.6 Prevajalna tabela

V tem zadnjem koraku bomo na podlagi tabele prehodov, množic FIRST in FOLLOW zgradili prevajalno tabelo, ki bo usmerjala sintaksni analizator.

Tabela sestoji iz akcijskega (ACTION) ter usmerjevalnega dela (GOTO). Stolpci v ACTION delu so končni simboli in $, stolpci v GOTO delu pa vmesni simboli. Vrstice so stanja LR.

Stanje	ACTION	GOTO
Št. stanja	Akcije pomika in prevedbe	Skoki

V ACTION delu ima lahko tabela sledeče tipe akcij:

- sprejem: ko analizator naleti na to akcijo ve, da stavek ustreza gramatiki – analiza se konča;
- sN: akcija pomika (N predstavlja št. stanja);
- rN: akcija prevedbe (N predstavlja št. produkcije).

Gradnja prevajalne tabele bo opisana v sledečih točkah.

3.6.1 Inicializacija tabele

Tabelo inicializiramo tako, da v GOTO del tabele prepišemo vse vmesne simbole ter pripadajoče prehode iz tabele prehodov. V ACTION del tabele v stolpec $ ter vrstico, ki označuje tisto stanje LR(0), ki vsebuje produkcijo začetnega simbola s kazalcem za zadnjim simbolom, akcijo sprejema. V našem primeru je to stanje I_4, saj ta vsebuje produkcijo S -> E·. V ACTION del prepišemo vse stolpce končnih simbolov iz tabele prehodov ter jih predznačimo z znakom s (shift).

Prevajalna tabela je prikazana v spodnji tabeli.

	ACTION								GOTO					
Št. st.	$	(	)	n	-	+	*	/	S	E	T	F	A	M
0		s1		s2	s3					4	5	6		
1		s1		s2	s3					7	5	6		
2														
3		s1		s2	s3							8		
4	acc.				s9	s10							11	
5							s12	s13						14
6														
7			s15		s9	s10							11	
8														
9														

	ACTION								GOTO					
10														
11		s1		s2	s3						16	6		
12														
13														
14		s1		s2	s3							17		
15														
16							s12	s13						14
17														

Tabela 6: Delno zgrajena prevajalna tabela

3.6.2 Tabela prevedb

V tem koraku bomo določili akcije prevedb s pomočjo množic FOLLOW ter razširjene gramatike.

Pripravimo si tabelo v kateri uparimo produkcije razširjene gramatike z pripadajočimi množicami FOLLOW. V pomoč lahko dodamo še stolpec z številko izvirne produkcije.

V tej tabeli poiščemo vse vrstice, katere razširjene produkcije izhajajo iz iste izvorne produkcije ter imajo ciljno številko zadnjega simbola v produkciji enako. Te vrstice lahko združimo ter sestavimo novo tabelo, ki bo vsebovala samo združene produkcije. Številka množice (tj. vrstica) kamor bomo zapisali združene produkcije, je določena z ciljno številko zadnjega simbola v produkciji.

Št. raz. prod.	Produkcija	Št. izvir. prod.	FOLLOW
0	${}_0S_{\$} \rightarrow {}_0E_4$	0	$
1	${}_0E_4 \rightarrow {}_0E_4\ {}_4A_{11}\ {}_{11}T_{16}$	1	$, +, -
2	${}_0E_4 \rightarrow {}_0T_5$	2	$, +, -
3	${}_0T_5 \rightarrow {}_0T_5\ {}_5M_{14}\ {}_{14}F_{17}$	3	$, +, -, *, /
4	${}_0T_5 \rightarrow {}_0F_6$	4	$, +, -, *, /
5	${}_0F_6 \rightarrow {}_0(_1\ {}_1E_7\ {}_7)_{15}$	5	$, +, -, *, /
6	${}_0F_6 \rightarrow {}_0n_2$	6	$, +, -, *, /

7	$_{0}F_{6}$ -> $_{0}$-$_{3}$ $_{3}F_{8}$	7	$, +, -, *, /
8	$_{1}E_{7}$ -> $_{1}E_{7}$ $_{7}A_{11}$ $_{11}T_{16}$	1	), +, -
9	$_{1}E_{7}$ -> $_{1}T_{5}$	2	), +, -
10	$_{1}T_{5}$ -> $_{1}T_{5}$ $_{5}M_{14}$ $_{14}F_{17}$	3	), +, -, *, /
11	$_{1}T_{5}$ -> $_{1}F_{6}$	4	), +, -, *, /
12	$_{1}F_{6}$ -> $_{1}(_{1}$ $_{1}E_{7}$ $_{7})_{15}$	5	), +, -, *, /
13	$_{1}F_{6}$ -> $_{1}n_{2}$	6	), +, -, *, /
14	$_{1}F_{6}$ -> $_{1}$-$_{3}$ $_{3}F_{8}$	7	), +, -, *, /
15	$_{3}F_{8}$ -> $_{3}(_{1}$ $_{1}E_{7}$ $_{7})_{15}$	5	$, +, -, *, /,)
16	$_{3}F_{8}$ -> $_{3}n_{2}$	6	$, +, -, *, /,)
17	$_{3}F_{8}$ -> $_{3}$-$_{3}$ $_{3}F_{8}$	7	$, +, -, *, /,)
18	$_{4}A_{11}$ -> $_{4}+_{10}$	8	(, n, -
19	$_{4}A_{11}$ -> $_{4}$-$_{9}$	9	(, n, -
20	$_{5}M_{14}$ -> $_{5}*_{12}$	10	(, n, -
21	$_{5}M_{14}$ -> $_{5}/_{13}$	11	(, n, -
22	$_{7}A_{11}$ -> $_{7}+_{10}$	8	(, n, -
23	$_{7}A_{11}$ -> $_{7}$-$_{9}$	9	(, n, -
24	$_{11}T_{16}$ -> $_{11}T_{16}$ $_{16}M_{14}$ $_{14}F_{17}$	3	$, +, -,), *, /
25	$_{11}T_{16}$ -> $_{11}F_{6}$	4	$, +, -,), *, /
26	$_{11}F_{6}$ -> $_{11}(_{1}$ $_{1}E_{7}$ $_{7})_{15}$	5	$, +, -,), *, /
27	$_{11}F_{6}$ -> $_{11}n_{2}$	6	$, +, -,), *, /
28	$_{11}F_{6}$ -> $_{11}$-$_{3}$ $_{3}F_{8}$	7	$, +, -,), *, /
29	$_{14}F_{17}$ -> $_{14}(_{1}$ $_{1}E_{7}$ $_{7})_{15}$	5	$, +, -, *, /,)
30	$_{14}F_{17}$ -> $_{14}n_{2}$	6	$, +, -, *, /,)
31	$_{14}F_{17}$ -> $_{14}$-$_{3}$ $_{3}F_{8}$	7	$, +, -, *, /,)
32	$_{16}M_{14}$ -> $_{16}*_{12}$	10	(, n, -
33	$_{16}M_{14}$ -> $_{16}/_{13}$	11	(, n, -

Primer:
V zgornji tabeli vidimo, da v vrstici 1 ter 8 produkciji izhajata iz iste izvirne produkciji ter imata enako ciljno številko zadnjega simbola - 16, zato ju združimo v:

Št. stanja	Št. prod. pred združitvijo	Produkcija	FOLLOW
16	1, 8	E -> E A T	$, +, -,)

Če ponovimo postopek za vse izvirne produkcije, dobimo sledečo tabelo prevedb:

Št. stanja	Št. prod. pred združitvijo	Produkcija	Št. izvirne prod.	FOLLOW
16	1, 8	E -> E A T	1	$, +, -,)
5	2, 9	E -> T	2	$, +, -,)
17	3, 10, 24	T -> T M F	3	$, +, -, *, /,)
6	4, 11, 25	T -> F	4	$, +, -, *, /,)
15	5, 12, 15, 26, 29	F -> (E)	5	$, +, -, *, /,)
2	6, 13, 16, 27, 30	F -> n	6	$, +, -, *, /,)
8	7, 14, 17, 28, 31	F -> - F	7	$, +, -, *, /,)
10	18, 22	A -> +	8	(, n, -
9	19, 23	A -> -	9	(, n, -
12	20, 32	M -> *	10	(, n, -
13	21, 33	M -> /	11	(, n, -

Tabela 7: Prevedbe

3.6.3 Končna prevajalna tabela

Končno tabelo zgradimo na podlagi delno zgrajene prevajalne tabele v točki a), v katero moramo vnesti še akcije prevedb, pri čemer si pomagamo s tabelo prevedb zgrajeno v točke b). Postopek je sledeč:

- sprehodimo se skozi tabelo prevedb,
- št. stanja iz tabele prevedb nam pove ciljno vrstico v prevajalni tabeli,

- za vsak končni simbol iz množice FOLLOW v ustrezen stolpec v prevajalni tabeli vnesemo akcijo prevedbe rN, kjer N ustreza št. izvorne produkcije

Na primeru:

Izsek iz tabele prevedb

Št. stanja	Št. prod. pred združitvijo	Produkcija	Št. izvirne prod.	FOLLOW
16	1, 8	E -> E A T	1	$, +, -,)

V stolpce $, +, - ter (, vrstica 16, vnesemo akcijo prevedbe r1.

Če postopek ponovimo za vse vrstice dobimo:

	ACTION								GOTO					
St.	$	(	)	n	-	+	*	/	S	E	T	F	A	M
0		s1		s2	s3					4	5	6		
1		s1		s2	s3					7	5	6		
2	r6		r6		r6	r6	r6	r6						
3		s1		s2	s3							8		
4	acc.				s9	s10							11	
5	r2		r2		r2	r2	s12	s13						14
6	r4		r4		r4	r4	r4	r4						
7			s15		s9	s10							11	
8	r7		r7		r7	r7	r7	r7						
9		r9		r9	r9									
10		r8		r8	r8									
11		s1		s2	s3						16	6		
12		r10		r10	r10									
13		r11		r11	r11									
14		s1		s2	s3							17		
15	r5		r5		r5	r5	r5	r5						
16	r1		r1		r1	r1	s12	s13						14
17	r3		r3		r3	r3	r3	r3						

Tabela 8: Prevajalno tabela

3.7 Sintaksna analiza

Ko imamo zgrajeno prevajalno tabelo, je postopek prevajanja sorazmerno preprost.

Postopek je sledeč:

1. Inicializiramo sklad stanj, tako da damo na vrh št. stanja 0, končne simbole iz vhodne besede damo v vrsto končnih simbolov ter inicializiramo izhod. Iz vrste končnih simbolov preberemo prvi končni simbol – ta postane trenutni simbol.

2. Glede na trenutni simbol ter stanja na vrhu sklada stanj iz prevajalne tabele preberemo akcijo.
3. Sedaj se ravnamo glede na tip akcije:
   a) Pomik sN: na vrh sklada damo stanje N ter iz vrste simbolov odstranimo 1 simbol.
   b) Prevedba rN: stanje N damo na konec izhoda, iz sklada stanj odstranimo toliko stanj, kolikor je število simbolov na desni strani produkcije N. Na vrhu sklada stanj imamo sedaj št. začasnega stanja M. Poiščemo produkcijo št. N: A -> α ter s pomočjo simbola A iz glave te produkcije v prevajalni tabeli za stanje M v GOTO delu prevajalne tabele odčitamo novo stanje ter ga damo na vrh sklada stanj.
   c) Sprejem: Beseda ustreza gramatiki, analiza se konča.
   d) Karkoli drugega: sintaksna napaka.
4. Nadaljujemo na korak 2.

Primer:

```
2 * (3 + 4)
```

Vhodno besedo pretvorimo v simbole:

```
n[2] * ( n[3] + n[4] ) $
```

Sam postopek bolj pregledno predstavimo v tabeli:

Vhodna beseda	Izhod	Sklad stanj	Naslednja akcija
n[2] * (n[3] + n[4]) $		0	s2[2]
* (n[3] + n[4]) $		0, 2	r6[3]
* (n[3] + n[4]) $	6	0, 6	r4[4]
* (n[3] + n[4]) $	6, 4	0, 5	s12
(n[3] + n[4]) $	6, 4	0, 5, 12	r10
(n[3] + n[4]) $	6, 4, 10	0, 5, 14	s1

2 Korak 1 – inicializacija. Iz prevajalne tabele odčitamo iz vrstice 0 ter stolpca n akcijo s2

3 Na vrh sklada stanj smo dali stanje 2 ter iz vhodne besede odstranili prvi simbol. Iz prevajalne tabele odčitamo glede na naslednji simbol * ter stanje na vrhu sklada 2 naslednjo akcijo – r6

4 Stanje akcije r6 damo na konec izhoda. Produkcija št. 6 je F -> n, ki ima 1 simbol na desni strani in tolikšno število stanj odstranimo iz sklada stanj. Glede na začasno stanje 0 ter začasni simbol iz glave produkcije 6 – simbol F v prevajalni tabeli poiščemo v GOTO delu naslednje stanje – stanje 6. V prevajalni tabeli za stanje 6 ter naslednji simbol * odčitamo naslednjo akcijo – r4

n[3] + n[4]) $	6, 4, 10	0, 5, 14, 1	s2
+ n[4]) $	6, 4, 10	0, 5, 14, 1, 2	r6
+ n[4]) $	6, 4, 10, 6	0, 5, 14, 1, 6	r4
+ n[4]) $	6, 4, 10, 6, 4	0, 5, 14, 1, 5	r2
+ n[4]) $	6, 4, 10, 6, 4, 2	0, 5, 14, 1, 7	s10
n[4]) $	6, 4, 10, 6, 4, 2	0, 5, 14, 1, 7, 10	r8
n[4]) $	6, 4, 10, 6, 4, 2, 8	0, 5, 14, 1, 7, 11	s2
) $	6, 4, 10, 6, 4, 2, 8	0, 5, 14, 1, 7, 11, 2	r6
) $	6, 4, 10, 6, 4, 2, 8, 6	0, 5, 14, 1, 7, 11, 6	r4
) $	6, 4, 10, 6, 4, 2, 8, 6, 4	0, 5, 14, 1, 7, 11, 16	r1
) $	6, 4, 10, 6, 4, 2, 8, 6, 4, 1	0, 5, 14, 1, 7	s15
$	6, 4, 10, 6, 4, 2, 8, 6, 4, 1	0, 5, 14, 1, 7, 15	r5
$	6, 4, 10, 6, 4, 2, 8, 6, 4, 1, 5	0, 5, 14, 17	r3
$	6, 4, 10, 6, 4, 2, 8, 6, 4, 1, 5, 3	0, 5	r2
$	6, 4, 10, 6, 4, 2, 8, 6, 4, 1, 5, 3, 2	0, 4	accept[5]

Če želimo iz izpisa rekonstruirati sintaksno drevo, moramo zgolj slediti prevedbam:

- r6: F(n[2])
- r4: T(F(n[2]))
- r10: M(*)
- r6: F(n[3])
- r4: T(F(n[3]))

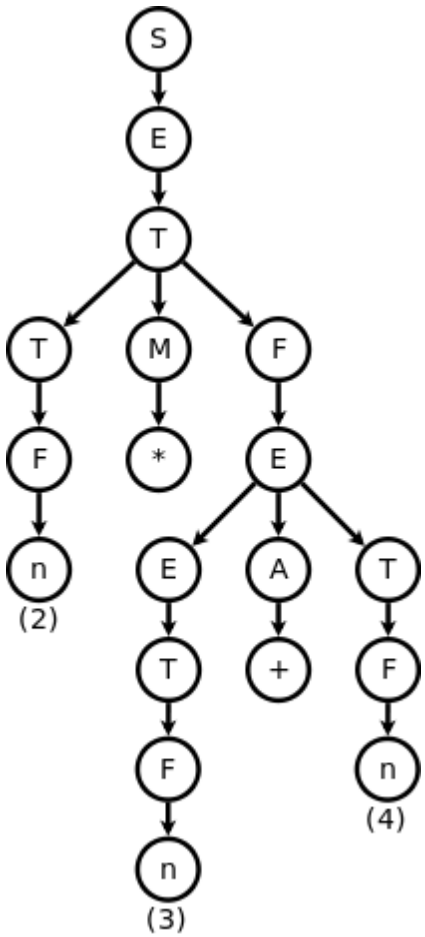
5 Ukaz sprejem nam pove, da vhodna beseda ustreza gramatiki – analiza se s tem konča.

- r2: E(T(F(n[3])))
- r8: A(+)
- r6: F(n[4])
- r4: T(F(n[4]))
- r1: E(E(T(F(n[3]))) A(+) T(F(n[4]))
- r5: F(E(E(T(F(n[3]))) A(+) T(F(n[4]))))
- r3: T (T(F(n[2])) M(*) F(E(E(T(F(n[3]))) A(+) T(F(n[4]))))
- r2: E(T (T(F(n[2])) M(*) F(E(E(T(F(n[3]))) A(+) T(F(n[4])))))

Iz česar dobimo končno izpeljavo:

```
S(E(T(T(F(n[2]))  M(*)  F(E( E(T(F(n[3])))  A(+)
T(F(n[4])))))))
```

Grafično je izpeljava predstavljena na sliki 5.



*Slika 5: Drevo izpeljave besede n * n + n*

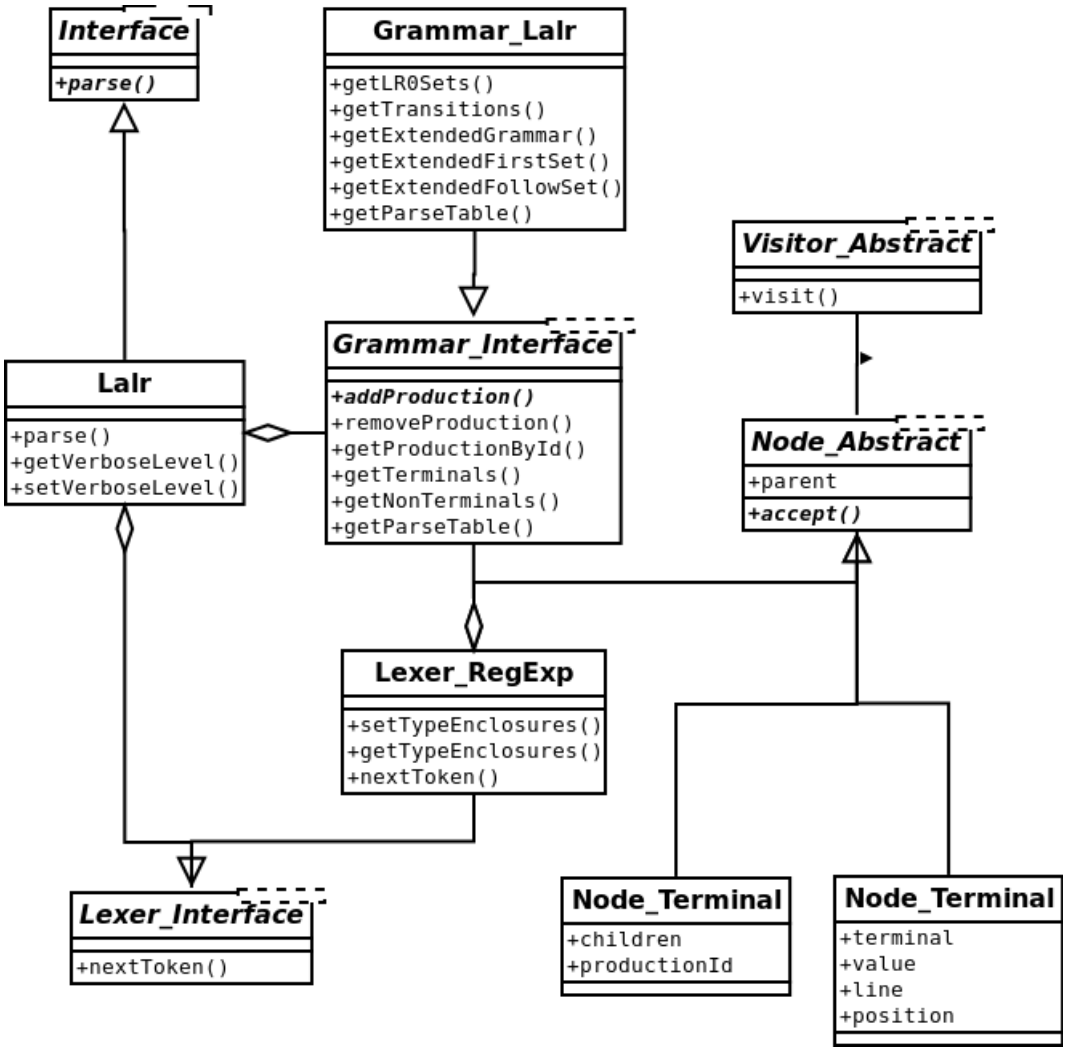
4 Zaključek

V okviru diplomskega dela je bil poleg generatorja LALR izveden tudi zametek transformatorja med standardnim SQL jezikom ter različnimi dialekti.

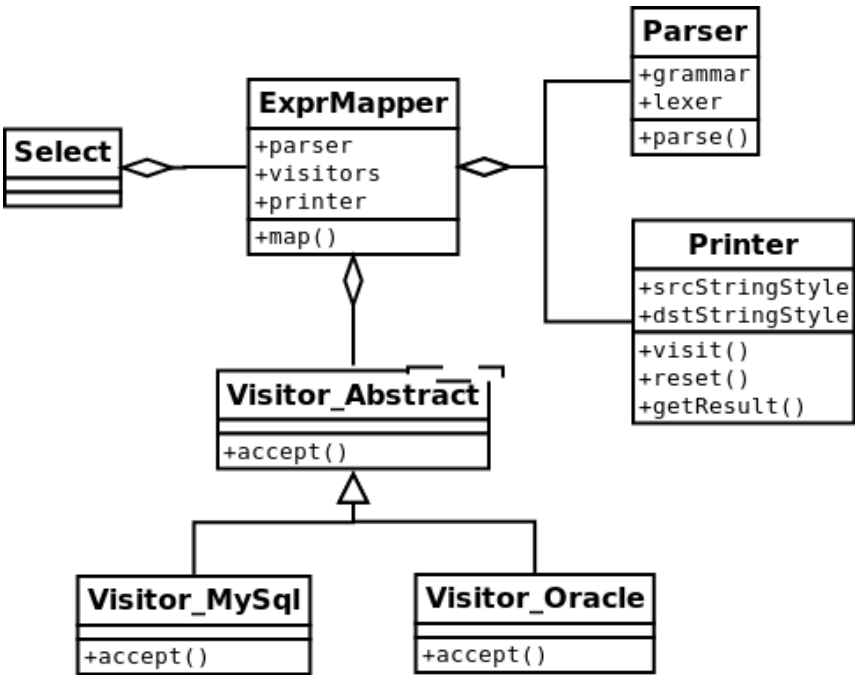
4.1 Izvedba

Pri izvedbi smo se držati dobrih praks objektnega programiranja, kot so: ločevanje odgovornosti, ohlapno vezani razredi, implementacija skrita pred uporabnikom, vstavljanje odvisnosti (dependeny injection) …

Pri tem sta nastala dva paketa: Util_Parse, kateri vsebuje logiko za leksikalno in sintaksno analizo ter Util_Db, ki razširja Zend_Db paket z logiko za prepisovanje fragmentov poizvedb. UML zgradba obeh paketov je razvidna iz spodnjih dveh slik.



Slika 6: Util_Parser UML diagram



Slika 7: Util_Db UML diagram

4.2 Zamenjava nepodprtih ANSI sql konstruktov

Cilj diplomskega dela je bil uporaba sintaktičnega analizatorja kot pomoč pri premoščanju razlik med različnimi SUPB. Rezultat analize je abstraktno sintaktično drevo AST, katero lahko uporabimo kot osnovo za zamenjavo različnih konstruktov. Na primer v ANSI standardu je operator, s katerim združujemo nize ||, ki pa ga MySql ne podpira oz. mu daje drugačen pomen (logični ali). Zamenjavo opravimo tako, da v AST poiščemo vse pojavitve oblike A_1 || A_2 || … || A_n ter jih zamenjamo s klicem funkcije CONCAT z argumenti A_1, A_2, …, A_n.

```
$db = Zend_Db::factory('Mysqli', array( /* params */));
// kreiranje select poizvedbe
$select = new Util_Db_Select($db);
$select->from('Uporabniki')
    ->where('FirstName || " " || LastName ="Janez Novak"')
;

// … izvede na SUPB sledečo poizvedbo
$sql = "SELECT `Uporabniki`.* FROM `Uporabniki` WHERE
( CONCAT ( `FirstName` , ' ', `LastName` ) = 'Janez
Novak' )";
```

Drug podprt primer je funkcija POSITION, ki jo Oracle podpira pod drugim imenom INSTR ter z zamenjanim vrstnim redom operandov.

```
$db = Zend_Db::factory('Oracle', array( /* params */));
// kreiranje select poizvedbe
$select = new Util_Db_Select($db);
$select->from('Uporabniki')
    ->where('POSITION("Novak" IN ImePriimek > 0')
;

// … izvede na SUPB sledečo poizvedbo
SELECT "Uporabniki".* FROM "Uporabniki" WHERE ( INSTR
( "ImePriimek" , "Novak" ) > 0 )
```

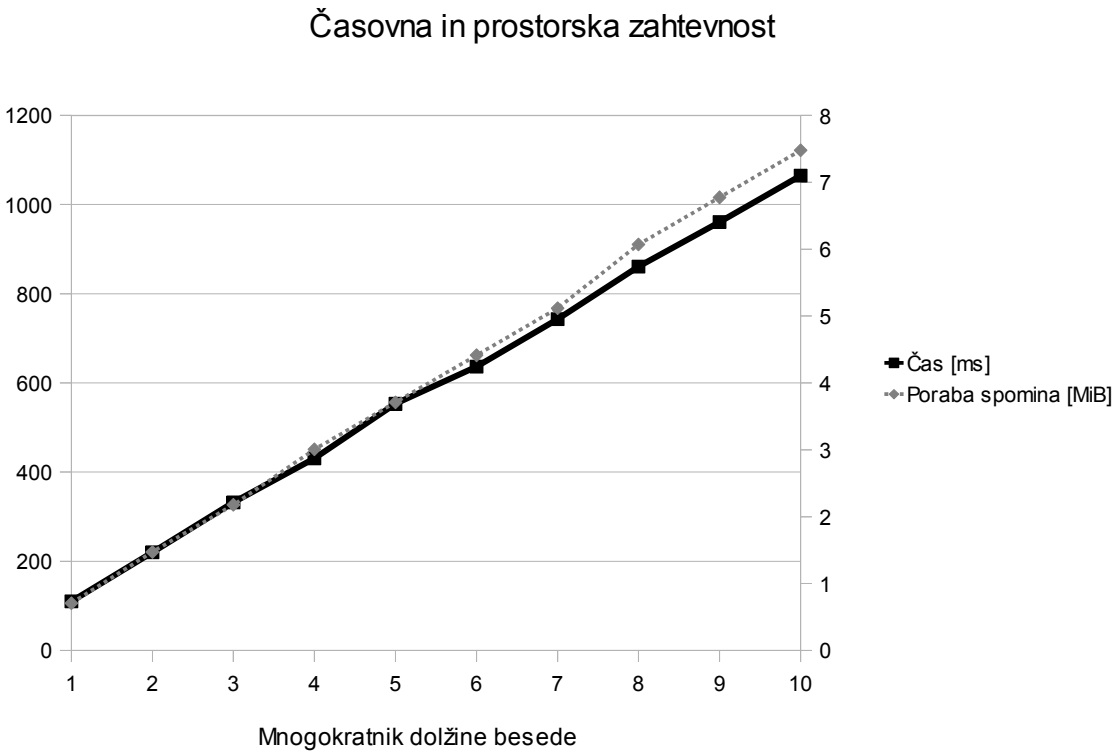
Zamenjava operatorja v obeh primerih ni povsem enostavna, saj je dobro zagotoviti, da popravljeno drevo ustreza vhodni gramatiki. Z drugimi besedami, po opravljeni zamenjavi, je zaželjeno, da drevo postane enako drevesu, katerega bi dobili, če bi analizirali stavek CONCAT (FirstName , " ", LastName) = "Janez Novak". Enakost zagotavlja, da lahko nad tem drevesom izvedemo še dodatne transformacije.

4.3 Časovna in prostorska zahtevnost LALR analize

Trditev, da je časovna zahtevnost algoritma O(n) ter da je odvisna zgolj od dolžine vhodne besede in ne od velikosti gramatike [6], smo preveriti empirično.

Zahtevnost smo testirali na gramatiki podobni tisti na primeru 2.5. Beseda je bila dolžine 180 simbolov, ki smo jo v vsakem koraku povečal za 180 simbolov. Vsako meritev smo izvedli 3 krat ter za vrednost vzeli povprečje vseh treh meritev. Merili smo hitrost analize in porabo spomina ter rezultate grafično predstavili na sliki 5.

Iz rezultatov sklepamo, da je trditev resnična, saj poraba časa in spomina raste enako hitro kot raste dolžine vhodne besede.



Slika 8: Rezulati meritev

4.4 Sklep

Med pisanjem diplomskega dela sta bila implementirana dva tipa sintaksnih analizatorjev: LL(1) ter LALR(1). Prvi je sicer res preprostejši za izvedbo, vendar je pisanje ustrezne gramatike toliko bolj zamudno, saj so nasprotja FIRST/FIRST in FIRST/FOLLOW precej pogosta. Zato sem se na pobudo, vzpodbudo in ob pomoči mentorja odločil, da izvedem LALR analizator, saj so le-ti dosti bolj robustni in zmogljivi. Kaj hitro pa sem ugotovil, da bo izvedba dosti zahtevna – v splošnem so prevajalniki tega tipa zaradi kompleksnosti generirani s pomočjo t.i. generatorjev prevajalnikom kot so yacc, bison ali antlr in redkeje zgrajeni ročno – naša implementacija seveda ni izjema.

Zastavljen cilj je dosežen, orodje omogoča poljubne transformacije vhodnega stavka, zaradi dinamične implementacije (nove produkcije v gramatiko se lahko dodajajo v času izvajanja).

Kljub temu, da sta izvedeni zgolj dve transformaciji, je manjkajoče stavčne oblike relativno preprosto dodati. Tukaj vidim, da se skriva še dosti dela, saj bi bilo potrebno za praktično uporabo v gramatiko dodati prav vse oblike, kot so:

- razni operatorji in določila: [expr] [op] INTERVAL [expr] [unit]
- klice funkcij s posebno sintakso: CAST(field AS type).

Ker ima večina funkcij ustaljeno obliko ([ime funkcije] ([arg], …), brez posebne sintakse) bi se sicer večina transformacij verjetno nanašala na preimenovanje funkcij ter morebitnega obračanja vrstnega reda argumentov.

Eden izmed pomislekov je tudi hitrost transformacij, saj je zadnja stvar, ki bi si jo želeli od njih, da opazno upočasnijo celotno aplikacijo. Ključna optimizacija bi bila pretvorba implementacije iz jezika PHP v jezik C, saj bi se tako analiza kot transformacije izvajale mnogo hitreje. Vmesna rešitev bi bila lahko tudi izvedba medpomnenja že analiziranih poizvedb.

A. Izvorna koda

```
/**
  * Util/Parser/Grammar/Lalr.php
  *
  * LALR grammar class.
  *
 * Handles construction of parse tables.
 *
 * @author devil
 */
class Util_Parser_Grammar_Lalr extends Util_Parser_Grammar_Abstract
    implements Util_Parser_Type_Lalr {

    /**
     * Returns LR0 sets
     *
     * @return array
     */
    public function getLR0Sets() {
        $this->_calcLR0Sets();
        return $this->_sets;
    }

    /**
     * Calculate LR0 sets if they are not already calculated
     *
     * @return void
     */
    public function _calcLR0Sets() {
        // skip calculation is sets already calculated
        if (null !== $this->_sets) {
            return;
        }

        $startSymbol = $this->getStartSymbol();

        // final sets structure
        $sets = array();

        // Contains all set candidates. Candidates are identified by:
        // - parent set id
        // - list: map between production id and cursor
        $stack = array(array(-1, array(), null, 0));

        // contains transitions between sets that can be determined
        // while building sets
        $transitions = array();

        // Contains pointers between source and yet unknown destination sets
        // (identified by processing id).
        // This structure will be used to finalize building transitions table
        $unmappedTransitions = array();

        // Map between processing id and set id
        $processingItemToSetId = array();

        // initialize stack
        foreach ($this->getProductionsByName($startSymbol) as $production) {
            $stack[0][1][ $production['id'] ] = 0;
        }

        $processingItems = $stack[0][1];
        while (sizeof($stack)>0) {
            $item = array_pop($stack);
            list($parentSetId, $list, $fromSymbol, $processingItemId) = $item;
```

```
/*
 * set is multidimensional array where first dimension
 * is production id and value is array of distinct cursors
 */
$set = array();

// is set final set
$isFinalSet = false;

// for each production id, cursor pair
foreach ($list as $id=>$cursor) {
    if (!isset($set[$id])) {
        $set[$id] = array();
    }
    $set[$id][] = $cursor;

    $production = $this->getProductionById($id);

    // list of all symbols yet to be expanded
    $unadded = array();

    // if production is not exausted yet and cursor
    // is before non terminal
    if ($cursor<sizeof($production['rule']) &&
        $this->isNonTerminal($production['rule'][$cursor])) {
        $unadded[] = $production['rule'][$cursor];
    }

    // final set is determined where left side of production is start
    // symbol and production is exausted
    if ($production['name']==$this->getStartSymbol() &&
        $cursor==sizeof($production['rule'])) {
        $isFinalSet = true;
    }

    // expand set body
    while (sizeof($unadded)>0) {
        $symbol = array_pop($unadded);

        foreach ($this->getProductionsByName($symbol) as $production) {
            $id = $production['id'];

            if (!isset($set[$id])) {
                $set[$id] = array();
            }

            if (!in_array(0, $set[$id])) {
                $set[$id][] = 0;

                // candidates can only be non exausted productions
                if (sizeof($production['rule'])>0) {
                    $symbol = $production['rule'][0];
                    if ($this->isNonTerminal($symbol)) {
                        $unadded[] = $production['rule'][0];
                    }
                }
            }
        }

        $unadded = array_unique($unadded);
    }
}

// this should always be true
if (false == in_array($set, $sets)) {
    $sets[] = array(
        'parent'=>$parentSetId,
        'symbol'=>$fromSymbol,
```

```
                'items'=>$set,
            );
            $processingItemToSetId[$processingItemId] = sizeof($sets)-1;

            // mark final set
            if ($isFinalSet) {
                $this->_finalSetId = sizeof($sets)-1;
            }

            if (null!==$fromSymbol) {
                if (!isset($transitions[$parentSetId])) {
                    $transitions[$parentSetId] = array();
                }
                $transitions[$parentSetId][$fromSymbol] = sizeof($sets)-1;
            }

            // build helper list of items to be processed
            $newItems = array();
            foreach ($set as $id=>$cursors) {
                $production = $this->getProductionById($id);

                foreach ($cursors as $cursor) {
                    if ($cursor<sizeof($production['rule'])) {
                        $symbol = $production['rule'][ $cursor ];
                        if (!isset($newItems[$symbol])) {
                            $newItems[$symbol] = array();
                        }
                        $newItems[$symbol][ $production['id'] ] = $cursor+1;
                    }
                }
            }

            foreach ($newItems as $symbol=>$list) {
                $key = array_search($list, $processingItems);

                // new items
                if (false === $key) {
                    $stack[] = array(
                        sizeof($sets)-1,
                        $list,
                        $symbol,
                        sizeof($processingItems),
                    );
                    $processingItems[] = $list;

                // item is connected to some set
                // already processed or on stack
                } else {
                    $unmappedTransitions[] = array(
                        sizeof($sets)-1,
                        $symbol,
                        $key

                    );
                }
            }
        }
    }

    // finish set's connections
    foreach ($unmappedTransitions as $transition) {
        list($from, $symbol, $processingItemId) = $transition;

        if (!isset($transitions[$from])) {
            $transitions[$from] = array();
        }

        $transitions[$from][$symbol] =
```

```
                $processingItemToSetId[ $processingItemId ];
        }

        $this->_transitions = $transitions;
        $this->_sets = $sets;
    }

    /**

     * Returns transitions between sets.
     * It is multidimensional array where keys are source set ids and value is
     * array where key is transition symbol and value is destination set id.
     *
     * @return array
     */
    public function getTransitions() {
        $this->_calcLR0Sets();

        return $this->_transitions;
    }

    /**
     * Returns extended grammar. Symbols in extended grammar are decorated with
     * source and destination set id.
     *
     * For example if we took production A -> · B C D from set X then extended
     * production would be A(X,Y) = B(X,Z) C(Z,V) D(V,W), where:
     * - Y is number of set where we would get from set X following symbol A
     * - Z is number of set where we would get from set X following symbol B
     * - V is number of set where we would get from set Z following symbol C
     * - W is number of set where we would get from set V following symbol D
     *
     * @return array
     */
    public function getExtendedGrammar() {
        $this->_calcExtendedGrammar();
        return $this->_extendedGrammar;
    }

    /**
     * Calculate extended grammar if not already calculated.
     *
     * @return void
     */
    protected function _calcExtendedGrammar() {
        // skip calculation is grammar already calculated
        if (null !== $this->_extendedGrammar) {
            return;
        }

        $sets = $this->getLR0Sets();
        $transitions = $this->getTransitions();

        // final grammar structure
        $extendedGrammar = array();

        // helper map between extended production group id and left symbol
        $lookup = array();

        // extened productions grouped by left symbol (keys are denoted by
        // production group id)
        $productionsByName = array();

        // map between extened production ids and ordinary production ids
        $extendedGrammarToPlain = array();

        // walking through each LR0 set and all productions where cursor is
        // before first symbol
```

```
foreach ($sets as $setId=>$set) {
    foreach ($set['items'] as $id=>$cursors) {
        foreach ($cursors as $cursor) {
            if ($cursor==0) {
                $production = $this->getProductionById($id);

                $symbol = $production['name'];
                $start = $setId;

                // production having on left side start symbol
                // is special because it lead to end
                if ($symbol==$this->getStartSymbol()) {
                    $end = -1;
                } else {
                    $end = $transitions[$setId][$symbol];
                }

                // left side
                $extendedName = array(
                    $start,
                    $production['name'],
                    $end,
                );

                // determing sources and destinations for all symbols
                // on right side
                $nextSetId = $setId;
                $extendedRule = array();
                foreach ($production['rule'] as $symbol) {
                    $end = $transitions[$nextSetId][$symbol];

                    $extendedRule[] = array(
                        $start,
                        $symbol,
                        $end,
                    );

                    $start = $end;
                    $nextSetId = $end;
                }

                $extendedGrammar[] = array(
                    'name'=>$extendedName,
                    'rule'=>$extendedRule,
                );

                $extendedId = sizeof($extendedGrammar)-1;
                $key = array_search($extendedName, $lookup);
                if (false === $key) {
                    $lookup[] = $extendedName;
                    $key = sizeof($lookup)-1;
                    $productionsByName[$key] = array();
                }
                $productionsByName[$key][] = $extendedId;

                $extendedGrammarToPlain[$extendedId] = $id;
            }
        }
    }
}

// determing all extended non-terminals that will be used later on
$nonTerminals = array();
foreach ($extendedGrammar as $production) {
    if (!in_array($production['name'], $nonTerminals)) {
        $nonTerminals[] = $production['name'];
    }
}
```

```
        // storing results
        $this->_extendedGrammar = $extendedGrammar;
        $this->_extendedNonTerminals = $nonTerminals;
        $this->_extendedGrammarLookup = $lookup;
        $this->_extendedGrammarByName = $productionsByName;
        $this->_extendedGrammarToPlain = $extendedGrammarToPlain;
    }

    /**
     * Returns all extended production having on left side specified extended
     * symbol.
     *
     * @param array $extendedSymbol
     * @return array
     */
    public function getExtendedProductionsByName($extendedSymbol) {
        $this->_calcExtendedGrammar();

        $key = array_search($extendedSymbol, $this->_extendedGrammarLookup);
        $ids = $this->_extendedGrammarByName[$key];

        return array_intersect_key(
                    $this->getExtendedGrammar(),
                    array_fill_keys($ids, 1)
        );
    }

    /**
     * Returns list of all extended non-terminals
     *
     * @return array
     */
    public function getExtendedNonTerminals() {
        $this->_calcExtendedGrammar();
        return $this->_extendedNonTerminals;
    }

    /**
     * Resolves cylcles in first|follow sets by carfully propagating
     * terminals to each depended set.
     *
     * @param array $sets
     * @return array
     */
    protected function _resolveSetCylcles(&$sets) {
        // helper structure used to quickly determine which set is
        // depended on which
        $propagatedLookup = array();
        foreach ($sets as $srcSetId=>$srcSet) {
            $startSymbol = $srcSet['symbol'];
            $propagatedLookup[$srcSetId] = array();

            foreach ($sets as $setId=>$set) {
                if (in_array($startSymbol, $set['set'])) {
                    $propagatedLookup[$srcSetId][$setId] = 1;
                }
            }
        }

        // cycling until no more changes are made
        do {
            $anyChange = false;

            foreach ($sets as $srcSetId=>$srcSet) {
                $startSymbol = $srcSet['symbol'];

                // assumed set is set without dependecies to other sets
```

```
            $assumedSet = array();
            foreach ($srcSet['set'] as $symbol) {
                if (!is_array($symbol)) {
                    $assumedSet[] = $symbol;
                }
            }
            $assumedSet = array_unique($assumedSet);

            if (sizeof($assumedSet)==0) {
                continue;
            }

            // traversing all dependent sets
            foreach (array_intersect_key($sets, $propagatedLookup[$srcSetId])
                                        as $setId=>$set) {

                if (in_array($startSymbol, $set['set'])) {
                    // only append new terminals if this will actualy
                    // change it (so we can know when to stop
                    if (sizeof(array_intersect($assumedSet, $set['set']))
                        != sizeof($assumedSet)) {

                        $anyChange = true;
                        $sets[$setId]['set'] =
                            array_merge($set['set'], $assumedSet);
                    }
                }
            }
        }
    } while ($anyChange);


    foreach ($sets as $setId=>$set) {
        // remove unnecessery markers from set
        $sets[$setId]['set'] = array_unique(
                array_diff($set['set'], array(array())));
    }
}



/**
 * Calculate extended first sets.
 *
 * @return void
 */
protected function _calcExtendedFirstSets() {
    if (null!==$this->_extendedFirstSets) {
        return;
    }

    // final first set structure
    $firstSets = array();
    // helper map to quickly access first sets by symbol
    $firstSetsLookup = array();
    // stack of unprocessed symbols

    $stack = $this->getExtendedNonTerminals();

    while (sizeof($stack)>0) {
        $startSymbol = array_pop($stack);
        $firstSet = array();

        // walking through all productions that have symbol on the left side
        foreach ($this->getExtendedProductionsByName($startSymbol)
                                                as $production) {

            $allEps = true;
            foreach ($production['rule'] as $symbol) {
```

```
                // if symbol is terminal all symbols following must be
                // disregarded
                if ($this->isTerminal($symbol[1])) {
                    $firstSet[] = $symbol[1];
                    $allEps = false;
                    break;

                // if symbol A is non-terminal then we FIRST(A) to first set
                // if already calculated, otherwise we add A to first set
                // (non-terminals in first set are markers that we use later
                // on to determine final first sets
                } else {
                    $key = array_search($symbol, $firstSetsLookup);
                    if (false === $key) {
                        $firstSet[] = $symbol;
                    } else {
                        $firstSet = array_merge($firstSet,
                                array_diff(
                                    $firstSets[$key]['set'],
                                    array(self::EMPTY_PRODUCTION)
                                )
                        );
                    }

                    // if it is known, that FIRST(A) does not contain eps then
                    // all following symbols must be disregarded
                    if (!$this->_hasSymbolEps($symbol)) {
                        $allEps = false;
                        break;
                    }
                }
            }

            // if all symbols in productions can or do contain eps then
            // we add it to first set
            if ($allEps) {
                $firstSet[] = self::EMPTY_PRODUCTION;
            }
        }

        $firstSetsLookup[] = $startSymbol;
        $firstSets[] = array(
            'symbol'=>$startSymbol,
            'set'=>$firstSet,
        );
    }

    // resolve cylcles
    $this->_resolveSetCylcles($firstSets);

    // storing results
    $this->_extendedFirstSets = $firstSets;
    $this->_extendedFirstSetsLookup = $firstSetsLookup;
}

/**
 * Calculating first set for specified symbols
 *
 * @param array $symbols
 * @return array
 */
public function getExtendedFirstSet($symbols) {
    $this->_calcExtendedFirstSets();

    $allWithEps = true;
    $firstSet = array();

    foreach ($symbols as $symbol) {
```

```
            if ($this->isTerminal($symbol[1])) {
                $firstSet[] = $symbol[1];
                $allWithEps = false;
                break;
            } else {
                $key = array_search($symbol, $this->_extendedFirstSetsLookup);
                $firstSet = array_merge($firstSet,
                        array_diff(
                            $this->_extendedFirstSets[$key]['set'],
                            array(self::EMPTY_PRODUCTION)
                        )
                );

                if (!$this->_hasSymbolEps($symbol)) {
                    $allWithEps = false;
                    break;
                }
            }
        }

        if ($allWithEps) {
            $firstSet[] = self::EMPTY_PRODUCTION;
        }

        return array_unique($firstSet);
    }

    /**
     * Return all symbols A where FIRST(A) contain eps
     *
     * @return array
     */
    protected function _getSymbolsWithEps() {
        if (null === $this->_symbolsWithEps) {
            $grammar = $this->getExtendedGrammar();
            $symbolsWithEps = array();

            // we determine symbols that first set will contain eps
            // by propagating eps
            do {
                $lastCount = sizeof($symbolsWithEps);

                foreach ($grammar as $production) {
                    $allWithEps = true;
                    foreach ($production['rule'] as $symbol) {
                        if (!in_array($symbol, $symbolsWithEps)) {
                            $allWithEps = false;
                        }
                    }

                    if ($allWithEps &&

                        !in_array($production['name'], $symbolsWithEps)) {

                        $symbolsWithEps[] = $production['name'];
                    }
                }

            // we are done, when no new symbol with eps in first set is found
            } while (sizeof($symbolsWithEps)!=$lastCount);

            $this->_symbolsWithEps = $symbolsWithEps;
        }

        // storing and returning results
        return $this->_symbolsWithEps;
    }
```

```
/**
 * Returns true if FIRST($symbol) contains eps
 *
 * @param array $symbol
 * @return bool
 */
protected function _hasSymbolEps($symbol) {
    return in_array($symbol, $this->_getSymbolsWithEps());
}

/**
 * Calculate extended follow sets.
 *
 * @return void
 */
protected function _calcExtendedFollowSets() {
    // skip calculation if already calculated
    if (null !== $this->_extendedFollowSets) {
        return;
    }


    $startSymbol = array(
        0,
        $this->getStartSymbol(),
        -1
    );


    // initialize follow set of start symbol with eof marker
    $followSets = array(array(
        'symbol'=>$startSymbol,
        'set'=>array(self::EOF)
    ));
    $followSetsLookup = array($startSymbol);


    $grammar = $this->getExtendedGrammar();
    $stack = array();

    // preparing helper lookup structer to quickly access all production
    // that have specified symbol on the right side
    $rightSideLookup = array();
    $rightSideProductions = array();
    foreach ($grammar as $id=>$production) {
        if (!in_array($production['name'], $stack) &&

            $production['name']!=$startSymbol) {
            $stack[] = $production['name'];
        }

        foreach ($production['rule'] as $symbol) {
            if ($this->isNonTerminal($symbol[1])) {
                $key = array_search($symbol, $rightSideLookup);
                if (false === $key) {
                    $rightSideLookup[] = $symbol;
                    $key = sizeof($rightSideLookup)-1;
                    $rightSideProductions[$key] = array();
                }
                $rightSideProductions[$key][] = $id;
            }
        }
    }


    // we build follow sets for all exteneded non-terminals
    while (sizeof($stack)>0) {
        $startSymbol = array_pop($stack);
```

```
            $followSet = array();

            // find all productions that have specified symbol on the right side
            $key = array_search($startSymbol, $rightSideLookup);
            $ids = $rightSideProductions[$key];

            foreach ($ids as $id) {
                $production = $grammar[$id];

                // finding position of our symbol on the right side
                $pos = array_search($startSymbol, $production['rule']);
                // ... and storing only symbols to the left (can be empty)
                $symbols = array_slice($production['rule'], $pos+1);

                // calculate first set
                $firstSet = $this->getExtendedFirstSet($symbols);
                $hasEps = in_array(self::EMPTY_PRODUCTION, $firstSet);
                if ($hasEps) {
                    // removing eps from first set
                    $firstSet = array_diff($firstSet, array(self::EMPTY_PRODUCTION));
                }
                $followSet = array_merge($followSet, $firstSet);

                // if first set contains eps, then we need to add FOLLOW of
                // production left side to this follow set
                if ($hasEps) {
                    $key = array_search($production['name'], $followSetsLookup);

                    // if follow set was already calculated then we just append it
                    if (false === $key) {
                        $followSet[] = $production['name'];

                    // ... otherwise we just put symbol (which servers as marker)
                    } else {
                        $followSet = array_merge($followSet, $followSets[$key]['set']);
                    }
                }
            }

            $followSets[] = array(
                'symbol'=>$startSymbol,
                'set'=>$followSet,
            );
            $followSetsLookup[] = $startSymbol;
        }

        // resolving set dependecies
        $this->_resolveSetCylcles($followSets);

        // storing results
        $this->_extendedFollowSets = $followSets;
        $this->_extendedFollowSetsLookup = $followSetsLookup;
    }

    /**
     * For $symbol A return it's FOLLOW(A)
     *
     * @param array $symbol
     * @return array
     */
    public function getExtendedFollowSet($symbol) {
        $this->_calcExtendedFollowSets();

        $key = array_search($symbol, $this->_extendedFollowSetsLookup);
        return $this->_extendedFollowSets[$key]['set'];
    }


    /**
```

```
     * Calculate LR parse table
     *
     * @return void
     */
    protected function _calcParseTable() {
        // already calculated
        if (null !== $this->_parseTable) {
            return;
        }


        // Reductions are calculated from extended grammar left sides
        // where final set is determined by final set number of last symbol
        // on the left side, symbols by follow set of left side and reduction
        // target by original production id
        $reductions = array();
        foreach ($this->getExtendedGrammar() as $extenedId => $production) {
            // skiping start production (will be set to accept action)
            if ($production['name'][1]==$this->getStartSymbol()) {
                continue;
            }

            // original production id
            $id = $this->_extendedGrammarToPlain[$extenedId];

            // determing final set number for non-empty rules
            if (sizeof($production['rule'])>0) {
                $lastSymbol = end($production['rule']);
                $end = $lastSymbol[2];

            // for empty rules this is an exceptions, because empty rules
            // point to original set of left side symbol
            } else {
                $end = $production['name'][0];
            }

            // initializing structure
            if (!isset($reductions[$end])) {
                $reductions[$end] = array();
            }
            if (!isset($reductions[$end][$id])) {
                $reductions[$end][$id] = array();
            }

            // merging follow sets
            $reductions[$end][$id] = array_unique(
                    array_merge(
                        $reductions[$end][$id],
                        $this->getExtendedFollowSet($production['name']
                    ))
            );
        }


        $transitions = $this->getTransitions();
        $sets = $this->getLR0Sets();

        // initializing parse table
        $parseTable = array();
        foreach (array_keys($sets) as $setId) {
            $parseTable[$setId] = array(
                'action'=>array(),
                'goto'=>array(),
            );
        }

        // we use transitions for determing goto and shift actions (non-terminals
        // results in goto's and terminal in shift actions
```

```
        foreach ($transitions as $setId=>$symbols) {
            foreach ($symbols as $symbol=>$destSetId) {
                if ($this->isTerminal($symbol)) {
                    $parseTable[$setId]['action'][$symbol] = array(
                        'action'=>'shift',
                        'id'=>$destSetId,
                    );
                } else {
                    $parseTable[$setId]['goto'][$symbol] = $destSetId;
                }
            }
        }

        // we apply all reductions as reduce actions
        foreach ($reductions as $setId=>$idFollow) {
            foreach ($idFollow as $id=>$follow) {
                foreach ($follow as $symbol) {
                    // if there is already action placed in specified cell
                    // this means only one thing: shift/reduce conflict or
                    // reduce/reduce conflict
                    if (isset($parseTable[$setId]['action'][$symbol])) {
                        switch ($parseTable[$setId]['action'][$symbol]['action']) {
                            case 'shift':
                                throw new Util_Parser_Grammar_Exception_ShiftReduce(
                                    "Shift/reduce conflict at set #$setId for symbol $symbol"

                                );
                                break;

                            case 'reduce':
                                throw new Util_Parser_Grammar_Exception_ReduceReduce(
                                    "Reduce/reduce conflict at set #$setId for symbol $symbol"

                                );
                                break;
                        }
                    }

                    $parseTable[$setId]['action'][$symbol] = array(
                        'action'=>'reduce',
                        'id'=>$id,
                    );
                }
            }
        }

        $parseTable[$this->_finalSetId]['action'][self::EOF] = array(
            'action'=>'accept',
        );

        // storing results
        $this->_parseTable = $parseTable;
    }
}
```

```
/**
  * Util/Parser/Lalr.php
  *
  * Lalr parse implementation
  *
 * @author devil
 */
class Util_Parser_Lalr implements Util_Parser_Interface, Util_Parser_Type_Lalr {

    public function setVerboseLevel($level) {
        $this->_verboseLevel = $level;
    }

    public function getVerboseLevel()  {
        return $this->_verboseLevel;
    }

    protected function _msg($msg, $level=0) {
        if ($this->getVerboseLevel()>Util_Parser_Interface::VERBOSE_QUIET &&
            $level<=$this->getVerboseLevel()) {
            echo "$msgn";
        }
    }

     public function parse(Util_Parser_Grammar_Interface $grammar, Util_Parser_Lexer_Interface
$lexer) {
        /*
         * TODO: add error handling
         */
        $this->_msg('Parsing started', Util_Parser_Interface::VERBOSE_CHATTERBOX);

        // load parse table from grammar
        $parseTable = $grammar->getParseTable();

        $stack = array(0);
        $symbolStack = array();

        $currentToken = $lexer->nextToken();
        $currentTerminal = $currentToken->getTerminal();

                                    $this->_msg("Read    first    terminal    $currentTerminal",
Util_Parser_Interface::VERBOSE_CHATTERBOX);

        while (true) {
            $setId = end($stack);

            if (!isset($parseTable[$setId]['action'][$currentTerminal])) {
                $this->_msg(
                    "No action found at $setId for terminal $currentTerminal ",
                    Util_Parser_Interface::VERBOSE_MEDIUM
                );
                return null;
            }

            $action = $parseTable[$setId]['action'][$currentTerminal];

            if ($action['action']=='shift') {
                $stack[] = $action['id'];
                $symbolStack[] = $currentToken;

                $currentToken = $lexer->nextToken();
                $currentTerminal = $currentToken->getTerminal();

                $this->_msg("Read terminal $currentTerminal moving to {$action['id']}",
                        Util_Parser_Interface::VERBOSE_CHATTERBOX);
            } else if ($action['action']=='reduce') {
```

```
                $productionId = $action['id'];
                $production = $grammar->getProductionById($productionId);
                $childSymbol = new Util_Parser_Node_NonTerminal($productionId);

                foreach ($production['rule'] as $rule) {
                    if ($rule!=Util_Parser_Grammar_Abstract::EMPTY_PRODUCTION) {
                        $childSymbol->addChild(array_pop($symbolStack));
                        array_pop($stack);
                    }
                }
                $symbolStack[] = $childSymbol;

                $tempSetId = end($stack);

                $stack[] = $parseTable[$tempSetId]['goto'][$production['name']];

                $this->_msg("Production #$productionId '".
                    $grammar->printProduction($productionId)."' moving to ".end($stack),
                    Util_Parser_Interface::VERBOSE_CHATTERBOX);
            } else if ($action['action']=='accept') {
                break;
            }
        }
        return $symbolStack[0];
    }
}
```

B. Vhod/izhod

Datoteka s gramatiko:

```
#       grammar.txt
#
#       Sample grammar
#       Terminal n is recognized as integer value
#
S -> E
E -> E + T
  | T
T -> T * n
  | n
```

Program lahko uporabljamo na dva načina:

- izpis vhodne gramatike, LR množic, razširjene gramatike, FIRST in FOLLOW množic ter prevajalne tabele;

Klic PHP interpreterja:

```
$ php -f cli.php grammar.txt
```

Program izpiše diagnostiko gramatike:

```
Grammar:
--------------------------------------
0.   S → E
1.   E → E + T
2.     | T
3.   T → T * n
4.     | n

LR(0) sets:
--------------------------------------
SET 0
=================
S → · E
E → · E + T
E → · T
T → · T * n
T → · n

SET 1 (from 0 by n)
================
```

```
T → n ·

SET 2 (from 0 by T)
==================
E → T ·
T → T · * n

SET 3 (from 2 by *)
==================
T → T * · n

SET 4 (from 3 by n)
==================
T → T * n ·

SET 5 (from 0 by E)
==================
S → E ·
E → E · + T

SET 6 (from 5 by +)
==================
E → E + · T
T → · T * n
T → · n

SET 7 (from 6 by T)
==================
E → E + T ·
T → T · * n


Extended grammar:
----------------------------------------
0.   0S-1 → 0E5
1.   0E5 → 0E5 5+6 6T7
2.   0E5 → 0T2
3.   0T2 → 0T2 2*3 3n4
4.   0T2 → 0n1
5.   6T7 → 6T7 7*3 3n4
6.   6T7 → 6n1

First sets:
----------------------------------------
FIRST(0S-1) = [n]
FIRST(0E5) = [n]
```

```
FIRST(0T2) = [n]
FIRST(6T7) = [n]

Follow sets:
--------------------------------------
FOLLOW(0S-1) = [$]
FOLLOW(0E5) = [$, +]
FOLLOW(0T2) = [*, $, +]
FOLLOW(6T7) = [*, $, +]

Parse table:
---------------------------------------
|     |      Action       |   Goto    |
---------------------------------------
| Set | $  | +  | *  | n  | S | E | T |
---------------------------------------
|  0  |    |    |    | s1 |   | 5 | 2 |
---------------------------------------
|  1  | r4 | r4 | r4 |    |   |   |   |
---------------------------------------
|  2  | r2 | r2 | s3 |    |   |   |   |
---------------------------------------
|  3  |    |    |    | s4 |   |   |   |
---------------------------------------
|  4  | r3 | r3 | r3 |    |   |   |   |
---------------------------------------
|  5  | a  | s6 |    |    |   |   |   |
---------------------------------------
|  6  |    |    |    | s1 |   |   | 7 |
---------------------------------------
|  7  | r1 | r1 | s3 |    |   |   |   |
---------------------------------------
```

- Analiza vhodnega programa

Klic PHP interpreterja:

```
$ php -f cli.php grammar.txt "2+4*3"
```

Sintaksno drevo:

```
E → E + T (#1)
   E → T (#2)
      T → n (#4)
         2 (n)
```

```
+
T → T * n (#3)
   T → n (#4)
      4 (n)
   *
   3 (n)
```

Slike

Tabele

Literatura

[1] Boštjan Vilfan, "Prevajanje programskih jezikov", (Fakulteta za računalništvo in informatiko, 2004)

[2] "Vendor lock-in", http://en.wikipedia.org/wiki/Vendor_lock-in

[3] Kevin Kline with Daniel Kline, Ph.D., "SQL In A Nutshell", (O'Reilly Media, 2000)

[4] "Entity-attribute-value model", http://en.wikipedia.org/wiki/Entity-attribute-value_model

[5] "CYK algorithm", http://en.wikipedia.org/wiki/CYK_algorithm

[6] "LALR parser", http://en.wikipedia.org/wiki/LALR_parser

[7] "yacc", http://en.wikipedia.org/wiki/Yacc

[8] "bison", http://www.gnu.org/software/bison/

[9] Dick Grune, Ceriel J.H. Jacobs, "Parsing Techniques - A Practical Guide ", (Ellis Horwood, Chichester, England, 1990)

[10] "A Tutorial Explaining LALR(1) Parsing", http://web.cs.dal.ca/~sjackson/lalr1.html

[11] "First and Follow Sets", http://www.jambe.co.nz/UNI/FirstAndFollowSets.html