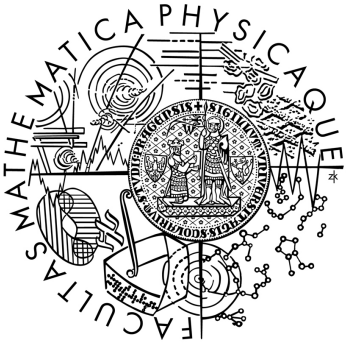
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Pavel Šašek

Rozšiřování syntaxe za běhu

Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Michal Žemlička, Ph.D.
Studijní program: Informatika, Softwarové systémy

2007

Rád bych poděkoval svému vedoucímu RNDr. Michalovi Žemličkovi, Ph.D. za soustavné vedení a cenné rady, které výrazně přispěly k vytvoření této diplomové práce.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 6. srpna 2007 Pavel Šašek

Obsah

Název práce: Rozšiřování syntaxe za běhu
Autor: Pavel Šašek
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Michal Žemlička, Ph.D.
e-mail vedoucího: zemlicka@ksi.mff.cuni.cz

Abstrakt: Rozšiřitelné jazyky jsou v současné době stále populárnější a přinášejí mnoho výhod. Takové jazyky mají relativně malé jádro, které se programátor snadno naučí, a lze je lépe přizpůsobit konkrétní řešené úloze – program je pak čitelnější, snáze laditelný a udržovatelný. Tato práce se zabývá možností rozšiřování syntaxe jazyka během analýzy vstupního textu, který může obsahovat pokyny pro taková rozšíření. Rozšíření může být buď permanentní (platí až do konce vstupního textu), nebo lokální (platí pouze dočasně). Práce přináší implementaci rozšiřitelného analyzátoru, založeného na přívětivých gramatikách, který podporuje permanentní i lokální rozšíření.

Klíčová slova: Za běhu rozšiřitelný analyzátor, lokální rozšíření, permanentní rozšíření, přívětivé gramatiky.

Title: Run-time syntax extensions
Author: Pavel Šašek
Department: Department of software engineering
Supervisor: RNDr. Michal Žemlička, Ph.D.
Supervisor's e-mail address: zemlicka@ksi.mff.cuni.cz

Abstract: Extensible languages are more and more popular now and bring many advantages. Such languages contain a relatively small base which a programmer can learn easily and are better adjustable for a particular problem – the code is then more legible, easier to debug and maintain. This work deals with the possibility of syntax extensions during the input text analysis, the input text can contain instructions for such extensions. An extension can be either permanent (valid to the input text end) or local (valid just temporarily). The work brings an implementation of an extensible parser based on kind grammars which supports permanent and local extensions.

Keywords: Run-time extensible parser, local extensions, permanent extensions, kind grammars.

Kapitola 1

Úvod

Tato práce se zabývá možností rozšiřování syntaxe jazyka za běhu, tzn. během překladu vstupního textu, který může obsahovat pokyny pro taková rozšíření. Rozšíření může být buď permanentní (platí až do konce vstupního textu), nebo lokální (platí pouze dočasně). Hlavním cílem je vytvořit aplikaci, která bude podporovat jak permanentní, tak i lokální rozšíření, a porovnat ji s existujícími pracemi.

1.1 Motivace[1]

V současné době existuje mnoho programovacích jazyků. Některé z nich jsou považovány za obecně použitelné (např. C++ nebo Java), jiné se naopak specializují na konkrétní účel (např. XPath pro adresování částí XML dokumentů). Problémem „velkého“ jazyka je rozsáhlost, programátor pak může mít potíže pamatovat si a zvládnout všechny jeho konstrukty; často se omezí jen na nějakou podmnožinu, kterou sám používá. Navíc i přes snahu o obecnou použitelnost se najde úloha, na kterou se takový jazyk prostě nehodí. Obdobná poznámka přirozeně platí i pro specializované jazyky.

Programátor je tak nucen jazyky střídat – pokud daný jazyk nezná, musí se jej naučit, nový jazyk s sebou případně přináší také nové vývojové prostředí. Takové změny samozřejmě stojí čas a zvyšuje se pravděpodobnost výskytu chyb.

Řešením naznačeného problému mohou být rozšiřitelné jazyky. Rozšiřitelný jazyk má relativně malé jádro, tedy malou část nutnou k zapama-

[1] Motivace pro rozšiřitelné jazyky je podrobněji rozebrána např. v pracech [Žem96, Žem06, Jan03].

tování, a dále se rozšiřuje podle potřeb řešené úlohy. Programátor si sám zvolí notaci a konstrukty, které se pro úlohu nejlépe hodí. Program je tak dobře čitelný, a tedy snáze laditelný a udržovatelný. Rozšiřitelný jazyk má šanci být skutečně univerzální. Programátor může získat jazyk, který přesně odpovídá potřebám konkrétní úlohy.

Myšlenka rozšiřitelnosti není zcela nová a stává se stále populárnější. Příkladem mohou být TEX a LATEX, SGML, XML, aplikační moduly (plug-ins) a další. Rozšiřitelnosti je dosahováno různými prostředky a na různých úrovních (vlastní identifikátory pro proměnné a funkce, knihovny, makra, přetěžování operátorů, šablony, ...), v této práci se budeme zabývat rozšiřitelností (zejména) syntaxe jazyka.

Translátory dnes nacházejí uplatnění v komplexních informačních systémech implementovaných softwarovými konfederacemi [KŽ05]. Zde v roli předřazených bran zajišťují převod problémově orientovaného rozhraní přístupného službám a uživatelům na implementačně orientované rozhraní aplikačních služeb (aplikací zajišťujících základní funkce systému). V případě, že zprávy jsou kódovány pomocí XML, je možné použít XSLT. V opačném případě je třeba využít nástroje založené na klasické syntaktické analýze, mezi něž patří i zde představený ExTra.

1.2 Zadání

> *Vytvořte za běhu rozšiřitelný přívětivý analyzátor podporující lokální rozšíření (s platností odpovídající konstrukcím daného jazyka) a porovnejte jej s existujícími implementacemi ([Žem02b] a [Žem94]) podporujícími permanentní rozšíření (to jsou taková, která platí od daného místa dále).*

Tato práce přímo navazuje na implementaci přívětivého analyzátoru KindTran [Žem02b], který je připraven na permanentní rozšíření, to zde však ještě není plně podporováno. Jak bylo zmíněno výše, hlavním cílem je tak dokončení permanentního rozšíření a přidání lokálního rozšíření.

S rozšířením syntaxe souvisí také opačný proces – redukce syntaxe. Práce pohlíží na rozšíření obecně, a to jako na změnu jazyka (gramatiky), podle kterého se provádí analýza vstupního textu. Tato změna může přidávat nové syntaktické konstrukty, ale zrovna tak je může i odebírat. Redukce a rozšíření jsou tedy speciální případy obecné změny jazyka, kterou se práce zabývá.

Kromě splnění hlavního cíle je snahou práce doplňovat původní implementaci [Žem02b] i v ostatních směrech tak, aby se postupně co nejvíce přibližovala reálně použitelnému translátoru.

1.3 Struktura práce

Text této práce je rozdělen do šesti kapitol. Následující kapitola obsahuje teoretické poznatky použité pro implementaci rozšiřitelného analyzátoru, a to přívětivé gramatiky a přívětivou analýzu. Mimo jiné je zde také základní pojmový aparát. Kapitola 3 popisuje vytvořený rozšiřitelný analyzátor ExTra z uživatelského pohledu (odpovídá uživatelské dokumentaci), implementační pohled lze nalézt v kapitole 4 (odpovídá vývojové dokumentaci). Kapitola 5 zmiňuje jiné práce zabývající se podobným tématem. V závěrečné šesté kapitole jsou shrnuty výsledky a možnosti dalšího rozvoje.

Součástí práce je CD, které obsahuje implementaci ExTra , včetně zdrojových kódů.

K práci jsou ještě přiloženy tři dodatky – v prvním je podrobněji rozepsáno, co přinesla tato práce, ve druhém obsah CD a ve třetím příklad ukazující možnosti analyzátoru.

Pro získání základních informací, jak se ExTra používá, stačí prostudovat kapitolu 3 – zejména její úvod a podkapitoly Spuštění, Definice jazyků, Konfigurační soubor (k základnímu pochopení není třeba znát všechna nastavení konfiguračního souboru) a Použití rozšíření a redukce. Nápomocné mohou být také ukázkové příklady, které jsou v kapitole 3 a v dodatku C, další pak nalezne čtenář na přiloženém CD.

Kapitola 2

Přívětivá analýza

Tato kapitola představuje třídu přívětivých gramatik a její souvislost s jinými známými třídami gramatik, dále pak ukazuje konstrukci přívětivého analyzátoru pro přívětivé gramatiky, nakonec se zaměří na problematiku rozšíření a redukce takového analyzátoru.

Přívětivé analyzátory dovolují snadnou rozšiřitelnost, která způsobí jen lokální změny v jejich struktuře a umožní dále pokračovat bez nutnosti restartu celé analýzy. Navíc jsou dostatečně rychlé, kompaktní a snadno modifikovatelné. Poradí si s levou rekurzí v pravidlech gramatiky nebo třeba se stejným prefixem pravých stran. Jsou tak vhodným kandidátem na implementaci za běhu rozšiřitelného analyzátoru, který podporuje i lokální rozšíření (tj. takové, které platí jen po určitou dobu, ne nutně až do konce překladu).

Celá kapitola vychází zejména z prací [Žem06, Žem02c, Žem96]; definice v podkapitole Základní pojmy byly s drobnými úpravami převzaty z [AU72, Chy84].

2.1 Základní pojmy

Zde budou uvedeny základní pojmy z teorie jazyků a překladačů, které jsou potřebné pro popsání přívětivých gramatik a přívětivé analýzy.

Definice 2.1.
Je-li Σ libovolná konečná množina (*abeceda*), pak Σ^+ označuje množinu všech konečných neprázdných posloupností utvořených z prvků množiny Σ. Nechť dále λ označuje prázdnou posloupnost. Pak můžeme definovat $\Sigma^* =$

$\Sigma^+ \cup \{\lambda\}$.

Posloupnost $a_1, \ldots, a_n \in \Sigma^*$ ($a_i \in \Sigma$) budeme zapisovat $a_1 \ldots a_n$. Každou takovou posloupnost nazýváme *slovem* v abecedě Σ. λ nazýváme *prázdným slovem*.

Definice 2.2 (Gramatika)**.**
Gramatika je čtveřice $G = (N, T, S, P)$, kde N a T jsou dvě disjunktní konečné abecedy, $S \in N$, P je konečná množina uspořádaných dvojic $\alpha \to \beta$ takových, že $\alpha, \beta \in (N \cup T)^*$ a α obsahuje alespoň jeden symbol z abecedy N.

N se nazývá množina *neterminálů*, T množina *terminálů*, S (*startovní*) *počáteční symbol* a P je množina (*přepisovacích*) *pravidel*.

Poznámka 2.1 (Značení)**.**
V práci je použita následující konvence:

- $a, b, c, \ldots$, reprezentují terminály,
- $A, B, C, \ldots$, reprezentují neterminály,
- $\ldots X, Y, Z$, reprezentují terminály nebo neterminály,
- $\alpha, \beta, \gamma, \ldots$, reprezentují slova složená z terminálů i neterminálů,
- $\ldots x, y, z$, reprezentují slova složená z terminálů.

Definice 2.3.
Nechť $G = (N, T, S, P)$ je gramatika, $\pi, \rho \in (N \cup T)^*$.

1. Řekneme, že π *se přímo přepíše na* ρ ($\pi \Rightarrow_G \rho$), právě když existují slova $\alpha, \beta, \gamma, \delta \in (N \cup T)^*$ taková, že $\pi = \gamma\alpha\delta$, $\rho = \gamma\beta\delta$ a $\alpha \to \beta$ je přepisovací pravidlo z P.

2. Řekneme, že π *se přepíše na* ρ ($\pi \Rightarrow_G^* \rho$), právě když existuje posloupnost
$$\pi_1, \pi_2, \ldots, \pi_n \ (n \geq 1)$$
slov z $(N \cup T)^*$ takových, že
$$\pi = \pi_1 \Rightarrow_G \pi_2 \Rightarrow_G \ldots \Rightarrow_G \pi_n = \rho.$$
Posloupnost $\pi_1, \pi_2, \ldots, \pi_n$ se nazývá *derivace* (odvození) slova ρ ze slova π.
Je-li z kontextu zřejmé, že se jedná o gramatiku G, můžeme psát $\Rightarrow$, $\Rightarrow^*$.

Definice 2.4 (Levá a pravá derivace)**.**
Je-li v derivaci přepisován vždy nejlevější neterminál, nazveme takovou derivaci *levou derivací*. Značíme $\Rightarrow^*_{G\ l}$ (nebo $\Rightarrow^*_l$, je-li z kontextu zřejmé, že se jedná o gramatiku G).

Je-li v derivaci přepisován vždy nejpravější neterminál, nazveme takovou derivaci *pravou derivací*. Značíme $\Rightarrow^*_{G\ r}$ ($\Rightarrow^*_r$).

Definice 2.5 (Jazyk generovaný gramatikou)**.**
Jazyk $L(G)$ generovaný gramatikou G je definován jako:

$$L(G) = \{w \mid w \in T^* \wedge S \Rightarrow^*_G w\}.$$

Je tedy tvořen všemi slovy v terminální abecedě, která lze odvodit z počátečního symbolu.

Definice 2.6 (Bezkontextová gramatika)**.**
Gramatika $G = (N, T, S, P)$ se nazývá *bezkontextová*, jestliže P obsahuje pouze pravidla typu

$$A \to \alpha,$$

kde $A \in N$ a $\alpha \in (N \cup T)^*$.

Definice 2.7 (A-pravidlo)**.**
A-pravidlo bezkontextové gramatiky je pravidlo tvaru

$$A \to \alpha,$$

kde $A \in N$ a $\alpha \in (N \cup T)^*$. A je tedy přepisovaný neterminál.

Definice 2.8 (First)**.**
Pro bezkontextovou gramatiku $G = (N, T, S, P)$ definujeme

$$FIRST_{k\ G}(\alpha) = \{x \mid \alpha \Rightarrow^*_l x\beta \wedge |x| = k \text{ nebo } \alpha \Rightarrow^* x \wedge |x| < k\}.$$

Je-li z kontextu zřejmé, že se jedná o gramatiku G, můžeme označení gramatiky vynechat a psát $FIRST_k(\alpha)$; je-li navíc $k = 1$, píšeme $FIRST(\alpha)$.

Definice 2.9 (Follow)**.**
Pro bezkontextovou gramatiku $G = (N, T, S, P)$ definujeme

$$FOLLOW_{k\ G}(\beta) = \{w \mid S \Rightarrow^* \alpha\beta\gamma \text{ a } w \in FIRST_{k\ G}(\gamma)\}.$$

Je-li z kontextu zřejmé, že se jedná o gramatiku G, můžeme označení gramatiky vynechat a psát $FOLLOW_k(\beta)$; je-li navíc $k = 1$, píšeme $FOLLOW(\beta)$.

Definice 2.10 (LL(k) gramatika)**.**
Řekneme, že bezkontextová gramatika $G = (N, T, S, P)$ je $LL(k)$, jestliže pro každé dvě levé derivace

1. $S \Rightarrow_l^* wA\alpha \Rightarrow_l w\beta\alpha \Rightarrow^* wx$ a
2. $S \Rightarrow_l^* wA\alpha \Rightarrow_l w\gamma\alpha \Rightarrow^* wy$

takové, že $FIRST_k(x) = FIRST_k(y)$, platí, že $\beta = \gamma$.

Definice 2.11 (Silná LL(k) gramatika)**.**
Řekneme, že bezkontextová gramatika $G = (N, T, S, P)$ je *silná* $LL(k)$ ($SLL(k)$), pokud platí: Jestliže $A \to \beta$ a $A \to \gamma$ jsou různá A-pravidla, pak

$$FIRST_k(\beta\, FOLLOW_k(A)) \cap FIRST_k(\gamma\, FOLLOW_k(A)) = \emptyset.$$

Definice 2.12 (LR(k) gramatika)**.**
K bezkontextové gramatice $G = (N, T, S, P)$ definujeme gramatiku $G' = (N \cup \{S'\}, T, S', P \cup \{S' \to S\})$. Řekneme, že gramatika G je $LR(k)$, jestliže z následujících tří podmínek

1. $S' \Rightarrow_{G'r}^* \alpha Aw \Rightarrow_{G'r} \alpha\beta w$,
2. $S' \Rightarrow_{G'r}^* \gamma Bx \Rightarrow_{G'r} \alpha\beta y$,
3. $FIRST_k(w) = FIRST_k(y)$

plyne, že $\alpha Ay = \gamma Bx$ (tj. $\alpha = \gamma$, $A = B$ a $x = y$).

Definice 2.13 (Zásobníkový automat)**.**
Zásobníkový automat je sedmice

$$(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F),$$

kde

1. Q je neprázdná konečná množina stavů,
2. Σ je neprázdná konečná vstupní abeceda,
3. Γ je neprázdná konečná zásobníková abeceda,
4. δ je zobrazení $Q \times (\Sigma \cup \{\lambda\}) \times \Gamma$ do konečných podmnožin $Q \times \Gamma^*$,

5. $q_0 \in Q$ je počáteční stav,

6. $Z_0 \in \Gamma$ je počáteční zásobníkový symbol,

7. $F \subseteq Q$ je množina koncových stavů.

2.2 Přívětivé gramatiky

Nyní definujme přívětivé gramatiky; k tomu je však potřeba zavést ještě několik dalších pojmů.

Definice 2.14 (Pravidla bez levé rekurze (NLRP[1])).

$$NLRP_G(A) = \{A \to \alpha \mid A \to \alpha \in P_G \ \wedge \ (\forall \beta)\alpha \not\Rightarrow_G^* A\beta\}$$
$$NLRP_G = \bigcup_{A \in N_G} NLRP_G(A)$$

Definice 2.15 (Pravidla s přímou levou rekurzí (DLRP[2])).

$$DLRP_G(A) = \{A \to A\alpha \mid A \to A\alpha \in P_G \ \wedge \ \alpha \neq \lambda\}$$
$$DLRP_G = \bigcup_{A \in N_G} DLRP_G(A)$$

Definice 2.16 (Následníci mimo levou rekurzi (NLRF[3])).
Nechť má gramatika G pravidla pouze z $NLRP$ a $DLRP$. Množina slov terminálů délky k, která se mohou objevit přímo za daným neterminálem A mimo levou rekurzi, se nazývá *následníci A mimo levou rekurzi* ($NLRF_k(A)$).

Definice 2.17 (Následníci v levé rekurzi (DLRF[4])).
Nechť má gramatika G pravidla pouze z $NLRP$ a $DLRP$. Množina slov terminálů délky k, která se mohou objevit přímo za daným neterminálem A v levé rekurzi, se nazývá *následníci A v levé rekurzi* ($DLRF_k(A)$).

Definice 2.18 ((Silná) k-přívětivá gramatika).
Bezkontextová gramatika $G = (N, T, S, P)$, která má pouze pravidla bez levé

[1]NLRP = Non-left-recursive productions
[2]DLRP = Directly left-recursive productions
[3]NLRF = Non-left-recursive follow
[4]DLRF = Directly left-recursive follow

rekurze nebo s přímou levou rekurzí, se nazývá *k-přívětivá* (*k-kind*), značení K(k), jestliže pro každý neterminál $A \in N$ platí následující dvě podmínky:

1. $DLRF_k(A) \cap NLRF_k(A) = \emptyset$;
2. pro každá dvě pravidla, která jsou buď obě z $NLRP(A)$ ($A \to \alpha\beta$, $A \to \alpha\gamma$ a α je nejdelší společný prefix jejich pravých stran) nebo obě z $DLRP(A)$ ($A \to A\alpha\beta$, $A \to A\alpha\gamma$ a $A\alpha$ je nejdelší společný prefix jejich pravých stran), platí

$$FIRST_k(\beta\, FOLLOW_k(A)) \cap FIRST_k(\gamma\, FOLLOW_k(A)) = \emptyset.$$

Definice 2.19 (Přívětivá gramatika)**.**
Bezkontextová gramatika G se nazývá *přívětivá* (*kind*), jestliže je k-přívětivá pro nějaké $k > 0$.

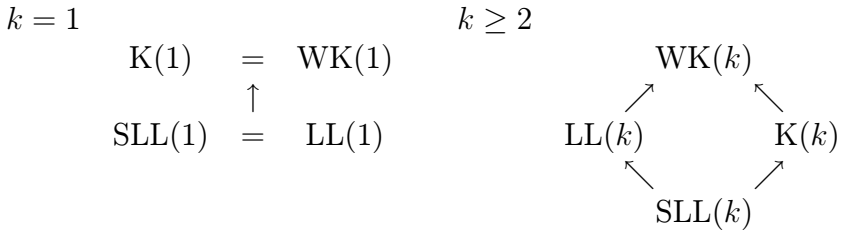
Podobně jako silné LL(k) gramatiky mohou být i (silné) k-přívětivé gramatiky rozšířeny na tzv. *slabé* k-přívětivé gramatiky, a to změnou podmínky 2 z definice:

Definice 2.20 (Slabá k-přívětivá gramatika)**.**
Bezkontextová gramatika $G = (N, T, S, P)$, která má pouze pravidla bez levé rekurze nebo s přímou levou rekurzí, se nazývá *slabá k-přívětivá* (*weak k-kind*), značení WK(k), jestliže pro každý neterminál $A \in N$ platí následující dvě podmínky:

1. $DLRF_k(A) \cap NLRF_k(A) = \emptyset$;
2. jestliže existuje $w \in T^*, \delta \in (N \cup T)^*$ takové, že $S \Rightarrow_l^* wA\delta$, pak platí: pro každá dvě pravidla, která jsou buď obě z $NLRP(A)$ ($A \to \alpha\beta$, $A \to \alpha\gamma$ a α je nejdelší společný prefix jejich pravých stran) nebo obě z $DLRP(A)$ ($A \to A\alpha\beta$, $A \to A\alpha\gamma$ a $A\alpha$ je nejdelší společný prefix jejich pravých stran), že

$$FIRST_k(\beta\delta) \cap FIRST_k(\gamma\delta) = \emptyset.$$

Třída přívětivých gramatik je ostře podtřídou LR gramatik a zároveň ostře nadtřídou LL gramatik. Platí, že k-přívětivé gramatiky generují LL(k) jazyky. Bližší vztah k-přívětivých gramatik k LL(k) gramatikám ukazuje obrázek 2.1. Podrobnosti a formální důkazy nalezne čtenář v [Žem06].



→ znamená „je vlastní podtřída“

Obrázek 2.1: Vztah k-přívětivých gramatik k LL(k) gramatikám

Přívětivé gramatiky mají mnoho dobrých vlastností jako LL(k) gramatiky, navíc připouštějí levou rekurzi (lze tedy použít např. standardní gramatiku aritmetického výrazu, která obsahuje pravidla s levou rekurzí). Omezení pravidel pouze na ty bez levé rekurze nebo s přímou levou rekurzí pomáhá její celkové přehlednosti – taková gramatika je pak mnohem lépe čitelná a srozumitelná. V podkapitole 2.5 Rozšíření a redukce bude ukázán další velmi podstatný rys přívětivých gramatik, a to snadná rozšiřitelnost jejích analyzátorů – rozšíření znamená jen lokální změny ve struktuře a analýza může pokračovat bez nutnosti restartu – proto je výhodné využít je pro implementaci za běhu rozšiřitelného analyzátoru.

2.3 Zásobníkový automat s výhledem

V této podkapitole představíme nový typ automatu, který bude základem pro přívětivý analyzátor – zásobníkový automat s výhledem. Základní myšlenka zní: dovolíme zásobníkovému automatu, aby se díval místo na právě čtený symbol vstupní abecedy na (nejvýše) k dosud nepřečtených symbolů.

Definice 2.21 (Zásobníkový automat s výhledem)**.**
Zásobníkový automat s výhledem k je osmice

$$(k, Q, \Sigma, \Gamma, \delta, q_0, Z_0, F),$$

kde

1. přirozené číslo $k \geq 1$ udává délku výhledu,
2. Q je neprázdná konečná množina stavů,

3. Σ je neprázdná konečná vstupní abeceda,

4. Γ je neprázdná konečná zásobníková abeceda,

5. δ je zobrazení $Q \times (\Sigma \cup \{\lambda\})^k \times \Gamma$ do konečných podmnožin $Q \times \Gamma^* \times \{0, \ldots, k\}$ (číslo mezi 0 a k určuje počet vstupních symbolů, které budou přečteny),

6. $q_0 \in Q$ je počáteční stav,

7. $Z_0 \in \Gamma$ je počáteční zásobníkový symbol,

8. $F \subseteq Q$ je množina koncových stavů.

Zásobníkový automat s výhledem rozpoznává právě bezkontextové jazyky. Výpočetní síla zásobníkového automatu s výhledem a „klasického“ zásobníkového automatu je stejná, jeden může být simulován druhým (důkaz v [Žem02c, Žem06]), pro $k = 1$ jsou dokonce oba shodné. Výhled má však vliv na to, jakou má automat strukturu – na počet stavů, na jeho čitelnost a přehlednost – čehož později využijeme při konstrukci a modifikacích našeho rozšiřitelného translátoru.

2.4 Principy přívětivé analýzy

Principy přívětivé analýzy a konstrukce analyzátoru pro přívětivou gramatiku vysvětlíme na příkladu gramatiky aritmetického výrazu (viz příklad 2.1).

Příklad 2.1 (Gramatika aritmetického výrazu)**.**
$G = \{N_G, T_G, S_G, P_G\}$, kde
$N_G = \{S, E, T, F\}$,
$T_G = \{id, num, (,), +, *\}$,
$S_G = S$,
$P_G = \{\ S \to E,$
$E \to E + T,$
$E \to T,$
$T \to T * F,$
$T \to F,$
$F \to id,$

$F \to num$,
$F \to (E)$
}.

□

Jak již bylo naznačeno v podkapitole 2.2 Přívětivé gramatiky, gramatika aritmetického výrazu je přívětivá (v [Žem06] je dokázáno, že je 1-přívětivá). Snadno se nahlédne, že však není LL(k) pro žádné k, protože obsahuje pravidla s levou rekurzí (podle [AU72, str. 344]): $E \to E+T$ a $T \to T*F$. Bylo by možné převést tuto gramatiku na jinou, která bude generovat stejný jazyk a nebude obsahovat pravidla s levou rekurzí – ale LL(k) gramatika, která vznikne takovým převodem, ztrácí původní přehlednost. Přívětivý analyzátor levou rekurzi připouští, přepis tedy není nutný. Navíc tato technika dovoluje umisťovat sémantické akce i jinam než na konec pravidel; mohou tedy být na místě, kam logicky patří. Přívětivý analyzátor tak umožňuje zachovat původní ráz gramatiky, takže lze snadno popsat zadaný jazyk pomocí formální gramatiky.

Nyní popíšeme způsob, jak k zadané přívětivé gramatice vytvořit přívětivý analyzátor.

Na každou množinu pravidel přívětivé gramatiky G se lze dívat jako na les *stromů pravidel*. Takový les stromů pravidel je pak snadno převeditelný na datovou strukturu, která definuje (rozšiřitelný) analyzátor přijímající jazyk $L(G)$. Strom pravidla je strom ve smyslu teorie grafů, jeho konstrukci vysvětlíme v následujících odstavcích na příkladu gramatiky aritmetického výrazu.

Pravidla nejprve rozdělíme do skupin podle neterminálu na levé straně a v rámci těchto skupin ještě podle toho, jestli je pravidlo levě rekurzivní (tj. jestli pravá strana začíná stejným neterminálem, jako je na levé straně). Výsledek pro gramatiku aritmetického výrazu je vidět na obrázku 2.2. Každý řádek obsahuje pravidla se stejným neterminálem na levé straně; v levém sloupci jsou pravidla bez levé rekurze, v pravém sloupci s levou rekurzí.

V dalším kroku oddělíme neterminál z levé strany do zvláštního (prvního) sloupce. Nadále tak není nutné psát levé strany pravidel, protože to, jaký tam patří neterminál, plyne z pozice (řádku) v tabulce. Obdobně lze vynechat neterminály na začátku pravých stran levě rekurzivních pravidel; neterminál opět plyne z pozice (řádku) v tabulce (a to, že je pravidlo levě rekurzivní, dává jeho přítomnost v posledním sloupci). Popsanou změnu zahrnuje obrázek 2.3.

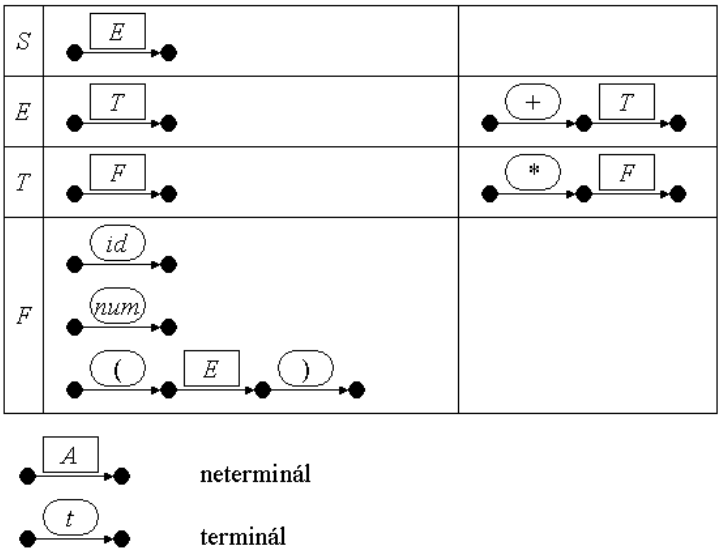
Pravidla	
bez levé rekurze	**s levou rekurzí**
$S \rightarrow E$	
$E \rightarrow T$	$E \rightarrow E + T$
$T \rightarrow F$	$T \rightarrow T * F$
$F \rightarrow id$ $F \rightarrow num$ $F \rightarrow (E)$	

Obrázek 2.2: Pravidla rozdělená podle neterminálu na levé straně a přítomnosti levé rekurze

Neterminál	**Pravé strany pravidel**	
	bez levé rekurze	**s levou rekurzí**
S	E	
E	T	$+T$
T	F	$*F$
F	id num (E)	

Obrázek 2.3: Zkrácený popis pravidel, která jsou rozdělená podle neterminálu na levé straně a přítomnosti levé rekurze

Tutéž věc je možné zapsat také jinak. Ke každému pravidlu přiřadíme cestu v orientovaném grafu. Hrany této cesty jsou ohodnoceny symboly (neterminály, terminály) z daného pravidla. Stejně jako na obrázku 2.3 neterminály z levých stran a neterminály ze začátku pravých stran levě rekurzivních pravidel plynou z pozice v tabulce, není tedy třeba jejich hrany a odpovídající uzly psát. Pravidla jako cesty v orientovaném grafu ukazuje obrázek 2.4.

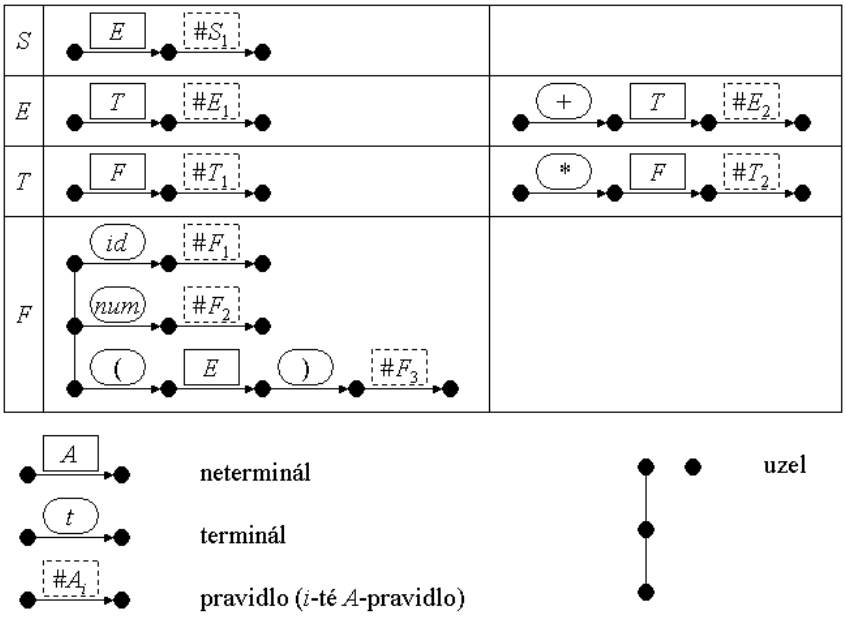


Obrázek 2.4: Pravidla jako cesty v acyklickém orientovaném grafu

Z praktických důvodů je výhodné přidat k cestám na jejich konec ještě jeden uzel, který identifikuje dané pravidlo (respektive jej identifikuje vstupující hrana do tohoto uzlu a její ohodnocení). To je důležité zejména tehdy, když pravá strana nějakého pravidla je prefixem pravé strany jiného pravidla. Hrana je ohodnocena neterminálem z levé strany pravidla a indexem, protože může být více pravidel se stejným neterminálem na levé straně (tento index se určil podle pořadí, v jakém byla pravidla uvedena na obrázku 2.2).

Ohodnocení hrany může být i jiné než zmíněný terminál, neterminál nebo pravidlo. V praxi se využije zejména ohodnocení sémantická akce nebo výstupní terminál.

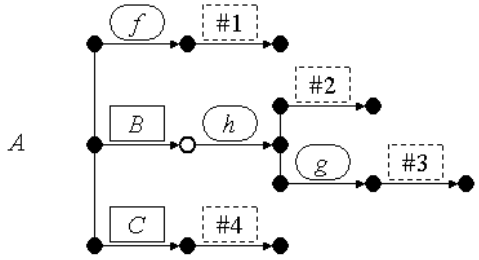
Nyní nic nebrání spojení stejných začátků všech cest pravidel (tj. stejných prefixů pravých stran pravidel) tak, aby utvořily stromy – stromy pravidel. Výsledek s identifikací pravidel a spojením cest zobrazuje obrázek 2.5.



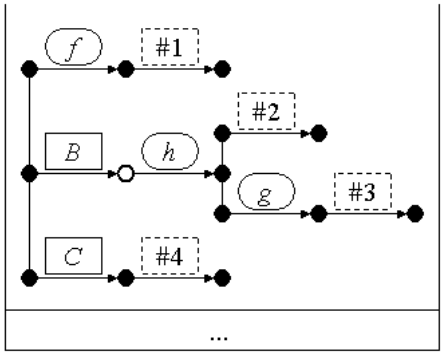
Obrázek 2.5: Pravidla jako cesty ve stromě pravidel

Obrázek 2.5 lze použít jako definici přívětivého analyzátoru, který bude přijímat jazyk gramatiky aritmetického výrazu. Přívětivý analyzátor může být implementován jako zásobníkový automat (s výhledem).

Symboly zásobníku a stavy automatu implementujeme jako ukazatele nebo reference na uzly (případně hrany) stromů pravidel; určují tedy nějakou pozici ve stromě. Reference na uzel bude graficky znázorněna černou tečkou s bílým středem (viz obrázek 2.6), grafická podoba zásobníku viz obrázek 2.7.



Obrázek 2.6: Pozice ve stromě pravidel v grafické notaci



Obrázek 2.7: Stav odložený na zásobníku v grafické notaci

Pozice ve stromě pravidel může být znázorněna také pomocí tzv. *tečkované notace*. V tečkované notaci je pravá strana pravidla rozdělena pomocí symbolu tečky (.) na dvě části – první část (před tečkou) odpovídá již zpracovaným symbolům pravidla, druhá část (za tečkou) těm dosud nezpracovaným. Pro každé pravidlo $A \rightarrow \alpha\beta \in P$ existuje posloupnost tečkovaných pravidel od „$A \rightarrow .\alpha\beta$“ do „$A \rightarrow \alpha\beta.$“.

V některých případech může jedné pozici ve stromě pravidel odpovídat více různých tečkovaných pravidel. Je to tehdy, když je tečka na začátku pravidel nebo když více pravidel ze stejné skupiny (tj. se stejným neterminálem na levé straně) má stejný prefix pravých stran. Taková tečkovaná pravidla označíme jako ekvivalentní (viz definice 2.22).

Definice 2.22 (Ekvivalence tečkovaných pravidel)**.**
Pro libovolnou bezkontextovou gramatiku $G = (N, T, S, P)$ definujeme ekvivalenci $=_G$ na tečkovaných pravidlech jako:

$$A \to \alpha.\beta =_G B \to \gamma.\delta \;\Leftrightarrow_{def}\; A = B \text{ a } \alpha = \gamma,$$

kde $A, B \in N$, $\alpha, \beta, \gamma, \delta \in (N \cup T)^*$, $A \to \alpha\beta, B \to \gamma\delta \in P$.

Třídu ekvivalence $=_G$, která obsahuje tečkované pravidlo $A \to \alpha.\beta$, značíme $[A \to \alpha.\beta]_{=G}$ (případně jen $[A \to \alpha.\beta]$).

Zmíněné případy tvoří třídy ekvivalence $=_G$, libovolné tečkované pravidlo může být reprezentant celé své třídy. Na začátku analýzy (tedy počáteční stav automatu) míří reference na kořen stromu počátečního symbolu, což je v tečkované notaci znázorněno jako: $[S \to .\alpha]$ pro libovolné pravidlo $S \to \alpha \in P$ a počáteční symbol S. Naopak koncové stavy automatu jsou $[S \to \alpha.]$. Pozici z obrázku 2.6 znázorňuje v tečkované notaci třída:

$$[A \to B.h] = \{A \to B.h, A \to B.hg\}.$$

To odpovídá grafické notaci: každý uzel je reprezentován jednou třídou ekvivalence, která je určena neterminálem na levé straně pravidla a pravou částí pravidla před tečkou. Část pravidla před tečkou dává cestu ve stromě k dané pozici; jsou-li ještě nějaké symboly za tečkou, znamená to pouze, že část před tečkou je prefixem nějakého existujícího pravidla. Neterminál (označme jej A) určuje dva stromy pravidel – první pro A-pravidla bez levé rekurze a druhý pro A-pravidla s levou rekurzí, jsou-li nějaká. Kořen stromu pravidel pro nějaké levě rekurzivní A-pravidlo může být značen $[A \to A.\alpha]$, kde $A \to A\alpha \in P$.

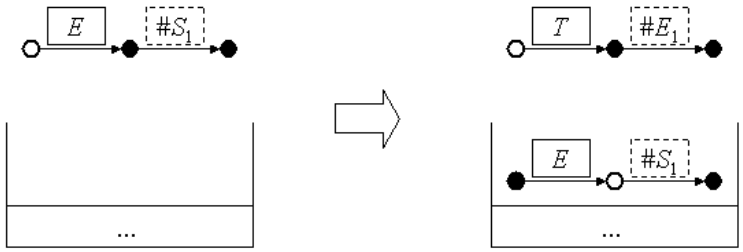
Při vlastní analýze se prochází stromy pravidel, počínaje kořenem stromu počátečního symbolu. Chování analyzátoru pak záleží na ohodnocení jednotlivých hran:

- analyzátor přechází přes hranu s terminálem (příklad viz obrázek 2.8),

– změní svůj stav (v řeči stromů pravidel se reference přesune na následující uzel),
– přečte terminál ze vstupu (musí se shodovat s terminálem z hrany, jinak výpočet končí zamítnutím);

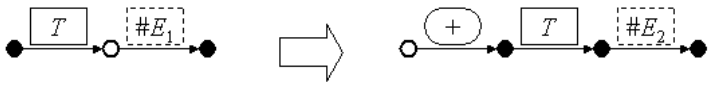
Obrázek 2.8: Analyzátor přechází přes hranu s terminálem (změna stavu analyzátoru, obsah zásobníku se nemění)

- analyzátor přechází přes hranu s neterminálem (příklad viz obrázek 2.9),
  – změní svůj stav (reference se přesune na kořen stromu nerekurzivního pravidla daného neterminálu),
  – stav reprezentující konec přecházené hrany vloží na zásobník,
  – vstup beze změny;



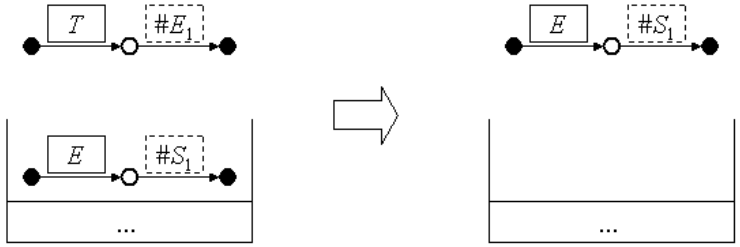
Obrázek 2.9: Analyzátor přechází přes hranu s neterminálem (změna stavu analyzátoru a obsahu zásobníku)

- analyzátor přechází přes hranu s pravidlem,
  – jestliže aktuální výhled vstupu odpovídá výhledu neterminálu tohoto pravidla do levé rekurze ($DLRF$), změní svůj stav (reference se přesune na kořen stromu levě rekurzivního pravidla daného neterminálu), příklad viz obrázek 2.10,



Obrázek 2.10: Analyzátor přechází přes hranu s pravidlem a výhled vstupu odpovídá *DLRF* (změna stavu analyzátoru, obsah zásobníku se nemění)

– jestliže aktuální výhled vstupu odpovídá výhledu neterminálu tohoto pravidla bez levé rekurze (*NLRF*), končí zpracování tohoto neterminálu a změní svůj stav na ten, který je uložen na vrcholu zásobníku (reference se přesune na uzel, kterým končí hrana ohodnocená neterminálem tohoto pravidla, kvůli které jsme přeskočili na kořen tohoto stromu), příklad viz obrázek 2.11,



Obrázek 2.11: Analyzátor přechází přes hranu s pravidlem a výhled vstupu odpovídá *NLRF* (změna stavu analyzátoru a obsahu zásobníku)

– jestliže aktuální výhled vstupu neodpovídá *DLRF* ani *NLRF* a zásobník je prázdný, výpočet končí a vstup je přijat,
– jinak výpočet končí a vstup je odmítnut.

Přívětivý analyzátor je tedy zásobníkový automat rozdělený na části, které odpovídají jednotlivým neterminálům – jako by byl tvořen rekurzivně volanými procedurami. Nejprve se analyzuje nerekurzivní část neterminálu podle pravidel bez levé rekurze a pak, pokud to jde, podle levě rekurzivních pravidel.

V příkladu 2.2 se podívejme, jak pracuje přívětivý analyzátor pro gramatiku aritmetického výrazu. Předpokládáme, že na vstupu je slovo „3+2“, které odpovídá dané gramatice.

Příklad 2.2 (Práce přívětivého analyzátoru)**.**

Vstup	**Stav**	**Zásobník**
Začátek výpočtu.		
$.3+2$	$[S \to .E]$	λ
Přechod k neterminálu E.		
$.3+2$	$[E \to .T]$	$[S \to E.]$
Přechod k neterminálu T.		
$.3+2$	$[T \to .F]$	$[S \to E.]$ $[E \to T.]$
Přechod k neterminálu F.		
$.3+2$	$[F \to .num]$	$[S \to E.]$ $[E \to T.]$ $[T \to F.]$
Přečtení „3“.		
$3.+2$	$[F \to num.]$	$[S \to E.]$ $[E \to T.]$ $[T \to F.]$
Konec rozvoje podle pravidla $F \to num$. Není žádné levě rekurzivní F-pravidlo, takže rozvoj neterminálu F končí.		
$3.+2$	$[T \to F.]$	$[S \to E.]$ $[E \to T.]$
Konec rozvoje podle pravidla $T \to F$. Nyní můžeme vyzkoušet, jestli je možné použít nějaké levě rekurzivní T-pravidlo. V tuto chvíli žádné takové není, takže rozvoj neterminálu T končí.		
$3.+2$	$[E \to T.]$	$[S \to E.]$
Konec rozvoje podle pravidla $E \to T$. Nyní můžeme vyzkoušet, jestli je možné použít nějaké levě rekurzivní E-pravidlo. Takové pravidlo je $E \to E+T$.		
$3.+2$	$[E \to E.+T]$	$[S \to E.]$
Přečtení „+“.		
$3+.2$	$[E \to E+.T]$	$[S \to E.]$
Přechod k neterminálu T.		
$3+.2$	$[T \to .F]$	$[S \to E.]$ $[E \to E+T.]$

Vstup	**Stav**	**Zásobník**
Přechod k neterminálu F.		
		$[S \rightarrow E.]$
		$[E \rightarrow E + T.]$
$3 + .2$	$[F \rightarrow .num]$	$[T \rightarrow F.]$
Přečtení „2“.		
		$[S \rightarrow E.]$
		$[E \rightarrow E + T.]$
$3 + 2.$	$[F \rightarrow num.]$	$[T \rightarrow F.]$
Konec rozvoje podle pravidla $F \rightarrow num$. Není žádné levě rekurzivní F-pravidlo, takže rozvoj neterminálu F končí.		
		$[S \rightarrow E.]$
$3 + 2.$	$[T \rightarrow F.]$	$[E \rightarrow E + T.]$
Konec rozvoje podle pravidla $T \rightarrow F$. Není možné použít žádné levě rekurzivní pravidlo $\Rightarrow$ dokončení T.		
$3 + 2.$	$[E \rightarrow E + T.]$	$[S \rightarrow E.]$
Konec rozvoje podle pravidla $E \rightarrow E+T$. Není možné použít žádné levě rekurzivní pravidlo $\Rightarrow$ dokončení E.		
$3 + 2.$	$[S \rightarrow E.]$	λ
Konec rozvoje podle pravidla $S \rightarrow E$, byla dosažena finální pozice $\Rightarrow$ analýza končí.		

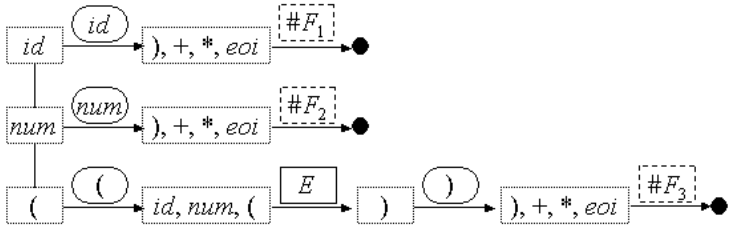
□

Při analýze je užitečné mít nějakou nápovědu, která cesta ve stromě pravidel je ta správná a vede k cíli, je-li jich více než jen jedna. Touto nápovědou jsou výhledy (*FIRST*, *FOLLOW*, *DLRF*, *NLRF*). Výhledy mohou být zapsány do uzlů stromu; při výpočtu pak tato informace dovolí vybrat správnou větev, která odpovídá vstupu (nebo má alespoň šanci odpovídat vstupu) a přes kterou bude výpočet pokračovat. Pro příklad gramatiky aritmetického výrazu stačí výhled délky 1. Navíc byl přidán speciální terminální symbol *eoi* (end of input), symbolizující konec vstupu. Vypočítané hodnoty funkcí *FIRST*, *FOLLOW*, *DLRF* a *NLRF* pro gramatiku aritmetického výrazu jsou k dispozici v tabulce 2.1.

Vypočítané výhledy v uzlech stromu ukazuje obrázek 2.12. Protože obrázek s výhledy pro celou gramatiku by byl příliš velký, byla vybrána jen část zobrazující neterminál F.

Neterminál	FIRST	FOLLOW	DLRF	NLRF
S	$num, id, ($	eoi		eoi
E	$num, id, ($	$+,), eoi$	$+$	$), eoi$
T	$num, id, ($	$+, *,), eoi$	$*$	$+,), eoi$
F	$num, id, ($	$+, *,), eoi$		$+, *,), eoi$

Tabulka 2.1: Vypočítané výhledy pro gramatiku aritmetického výrazu



Obrázek 2.12: Strom pravidel neterminálu F s vypočítanými výhledy

2.5 Rozšíření a redukce

Zpracování rozšiřitelných jazyků vyžaduje rozšiřitelné/modifikovatelné analyzátory. Změny ve struktuře přívětivého analyzátoru jsou prováděny prostřednictvím změn v lese stromů pravidel. V této podkapitole předvedeme možnosti rozšíření a redukce stromů pravidel a přívětivých analyzátorů na nich vystavěných.

Rozšíření jazyka a jeho gramatiky se provádí především přidáním nového pravidla; je také možné přidat nový terminál či neterminál. Přidání nového pravidla znamená přidání nového stromu pravidel do lesa stromů pravidel, resp. nové větve do již existujícího stromu pravidel a aktualizaci výhledů.

Tyto změny jsou již ze své podstaty aditivní – to znamená, že původní hrany zůstávají beze změny, pouze se k nim přidají nové části. Po provedení rozšíření jsou původní i nově přidané prvky rovnoprávné. Tato vlastnost dovoluje rozšiřovat analyzátor v libovolnou dobu, například za běhu, takže popis rozšíření může být součástí třeba právě analyzovaného textu.

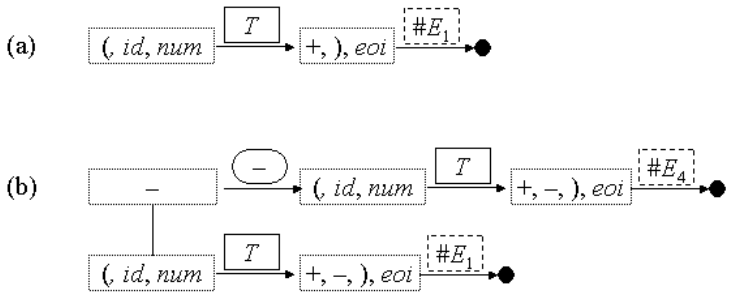
Pro praktické využití může mít smysl možnosti rozšíření nějak omezit – např. na místa, kde jsou definice podprogramů, datových typů a proměnných. Povolit rozšíření kdekoliv by mohlo být v konečném důsledku spíše

kontraproduktivní: způsobilo by nečitelnost a nejednoznačnost.

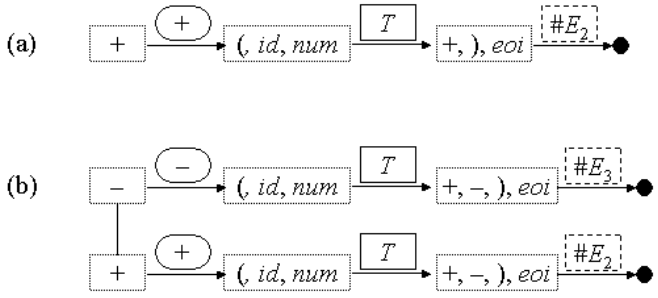
Z pohledu automatů je ještě třeba dodat, že není problém, ani když jsou při rozšíření uložené nějaké reference na zásobníku – tyto reference míří na původní uzly a ty se nemění. Rovněž stav a přechody budou korektní, pouze se doplní nové odpovídající přidaným uzlům a hranám.

Na obrázcích 2.13 a 2.14 je příklad rozšíření gramatiky aritmetického výrazu o možnost odčítání. Přidáme nový terminální symbol „−“ a dvě pravidla $E \to E - T$, $E \to -T$.

Schopnost rozšíření analyzátoru za běhu je jedna z hlavních výhod přívětivé analýzy.



Obrázek 2.13: Původní (a) a rozšířený (b) strom pravidel pro neterminál E gramatiky aritmetického výrazu – pravidla bez levé rekurze



Obrázek 2.14: Původní (a) a rozšířený (b) strom pravidel pro neterminál E gramatiky aritmetického výrazu – pravidla s levou rekurzí

Někdy může být užitečné rozpoznávaný jazyk naopak omezit. Ať už jde o odstranění předchozího rozšíření nebo původních konstruktů jazyka (třeba proto, aby nebyly v konfliktu s dalším plánovaným rozšířením).

Odstranění pravidla znamená vyjmutí odpovídající větve ze stromu pravidel a aktualizaci výhledů.

Redukce gramatiky je o něco nebezpečnější než její rozšíření. Může se stát, že odstraníme část, kterou analyzátor potřebuje pro dokončení výpočtu. Doslova: odřízneme větev, na které právě sedíme. Proto je třeba zavést pro redukci nějaká omezení nebo alespoň rozpoznat, že se analyzátor dostal do nekonzistentního stavu, a tento nějak řešit. Tato omezení zahrnují:

- cestu ve stromě pravidel je možné odstranit, jen když pro všechny dosud nevyřešené situace, které jsou na zásobníku, budou cesty zachovány,
- a také musí zůstat alespoň jedna cesta pro všechny neterminály, přes které bude třeba projít.

Zmíněná omezení mohou být oslabena, pokud je umožněno provádět redukci a rozšíření současně. Redukce tak může dočasně zablokovat cestu k dokončení výpočtu, avšak následující rozšíření ji opět obnoví.

Kapitola 3

Extensible Transducer

V této kapitole představíme implementovanou aplikaci – Extensible Transducer (ExTra) – z uživatelského pohledu; jak se spouští, jak se s ní pracuje, jaké má vlastnosti a jaký je formát konfiguračního souboru. Popíšeme sémantické instrukce, které je možné používat v programech pro sémantické akce. Zaměříme se na rozšíření a redukci gramatiky, uvedeme rovněž příklad konfiguračního souboru, kde se rozšíření a redukce využívají.

ExTra je implementace za běhu rozšiřitelného přívětivého analyzátoru – translátoru (transduceru)[1] – který dokáže gramatiku vstupního jazyka za běhu rozšířit i zredukovat, jak permanentně, tak i lokálně. Název ExTra, který je zkratkou ze sousloví Extensible Transducer, zdůrazňuje možnost rozšiřování.

Jedná se o aplikaci spouštěnou na příkazové řádce, jako vstup a výstup slouží textově orientované soubory. Vstupní soubory jsou dva a oba povinné: konfigurační soubor (langfile), který obsahuje veškerá nastavení, včetně definic vstupního a výstupního jazyka, a soubor se zdrojovým textem ve vstupním jazyce (sourcefile), který se bude analyzovat. Přesný popis formátu konfiguračního souboru a jeho příklad jsou v podkapitole 3.4. Výstupem programu je soubor s vygenerovaným textem ve výstupním jazyce (targetfile) a případně, pokud si to uživatel přeje, zpráva o načtení konfiguračního souboru (lstfile). Vše shrnuje obrázek 3.1.

[1] Pod pojmem translátor (transducer) rozumíme takový analyzátor, který generuje textový výstup.

Obrázek 3.1: Vstupní a výstupní soubory aplikace ExTra

3.1 Spuštění

ExTra se startuje na příkazové řádce spuštěním souboru `extra.exe` podle následující specifikace:

```
extra.exe [-d][-q][-w] <langfile> <sourcefile> <targetfile>
          [-l<lstfile>],
```

kde

- `-d` vypíše na standardní výstup jazyk a nastavení aplikace (dump),
- `-q` potlačí výpis hlavičky programu,
- `-w` potlačí všechna varování,
- `<langfile>` je název konfiguračního souboru (musí existovat),
- `<sourcefile>` je název souboru se vstupním (zdrojovým) textem (musí existovat),
- `<targetfile>` je název souboru s výstupním (vygenerovaným) textem (bude vytvořen, resp. přepsán),
- `-l <lstfile>` vytvoří záznam o načtení konfiguračního souboru, uloží se do souboru `lstfile` (bude vytvořen, resp. přepsán).

Zadání názvů konfiguračního souboru, souboru se vstupním textem a souboru s výstupním textem je povinné; ostatní položky jsou volitelné (volitelnost je označena pomocí hranatých závorek `[]`).

Příklad 3.1 (Ukázka spuštění)**.**
Máme konfigurační soubor `lang.lng` a soubor se vstupním textem `in.txt`; výstupní text chceme uložit do souboru `out.txt`, přejeme si záznam o načtení konfiguračního souboru a ten uložit do souboru `lst.txt`. ExTra spustíme takto:

```
extra.exe lang.lng in.txt out.txt -llst.txt
```

Za předpokladu, že nenastane žádná chyba, ExTra vypíše na standardní výstup následující text:

```
ExTra: Extensible Transducer                    (hlavička programu)
         2002, 2006 (c) Michal Zemlicka
         2007 (c) Pavel Sasek

Language file...'lang.lng'                      (konfigurační soubor)
Source file.....'in.txt'                        (soubor se vstupním textem)
Target file.....'out.txt'                       (soubor s výstupním textem)

LangLearn:                                      (načítání konfiguračního souboru)
 ... OK                                         (bez chyby)

Transduce:                                      (analýza vstupního textu)
 ... OK; 101 lines parsed.                      (bez chyby)
```

Nyní předpokládejme, že soubor `lang.lng` neexistuje; pak by nastala po spuštění chyba a ExTra by na standardní výstup vypsal:

```
ExTra: Extensible Transducer                    (hlavička programu)
         2002, 2006 (c) Michal Zemlicka
         2007 (c) Pavel Sasek

Language file...'lang.lng'                      (konfigurační soubor)
Source file.....'in.txt'                        (soubor se vstupním textem)
Target file.....'out.txt'                       (soubor s výstupním textem)

LangLearn:                                      (načítání konfiguračního souboru)
Filename: lang.lng (upřesnění chyby: název souboru, který nelze otevřít)
 ... Error E5: Cannot open file.                (chybové hlášení)
```

□

3.2 Definice jazyků

Vstupní jazyk se zadává v konfiguračním souboru pomocí své (přívětivé) gramatiky. Základem popisu gramatiky vstupního jazyka jsou definice terminálů (terminals), neterminálů (nonterminals) a pravidel (productions). Terminály a neterminály se nejprve definují v příslušných sekcích konfiguračního souboru (sekce *Terminals* a *Nonterminals*) – definice sestává pouze z názvu. Poté je lze použít jako součásti pravidel.

Seznam pravidel se uvádí v sekci *Productions*. Na levé straně pravidla musí být neterminál (přívětivá gramatika je bezkontextová), na pravé straně lze použít definované terminály, neterminály, sémantické akce a výstupní terminály.

Neterminály, které se vyskytují na pravých stranách pravidel, musí mít k sobě přidružen nějaký strom pravidel bez levé rekurze, tj. tyto neterminály musí být na levé straně nějakého pravidla bez levé rekurze. V některých případech je možné na pravých stranách pravidel použít i neterminál, který nemá strom pravidel bez levé rekurze (tj. tento neterminál není na levé straně žádného pravidla nebo je na levé straně pouze levě rekurzivních pravidel), a to tehdy, jsou-li splněny následující dvě podmínky:

1. tento neterminál není v místě žádného větvení příslušného stromu pravidel (tj. neúčastní se rozhodování o dalším postupu),
2. než bude při analýze vstupního textu zpracován, musí být pomocí rozšíření doplněno nějaké pravidlo bez levé rekurze s tímto neterminálem na levé straně.

Následující dva příklady ilustrují popsaný případ.

Příklad 3.2 (Neterminál bez pravidel – povolené použití)**.**
Neterminál `N` není na levé straně žádného pravidla (`N` je tedy neterminál bez pravidel). Pokud se však v pravidle „`PROLOG::=` ...“ objeví sémantická akce, která přidá pomocí rozšíření nové pravidlo tvaru „`N::=` ...“, bude tato gramatika v pořádku.

```
S::=[PROLOG][N][EPILOG]
PROLOG::= ...
EPILOG::= ...
```

□

Příklad 3.3 (Neterminál bez pravidel – nepovolené použití)**.**
Neterminál `N` opět není na levé straně žádného pravidla. Navíc se ovšem vyskytuje v místě větvení, kde dochází k rozhodování, které pravidlo se má při analýze použít – ExTra by tuto gramatiku označil za chybnou.

```
S::=[N]
S::=(terminal) ...
```

□

V konfiguračním souboru je dále třeba uvést název počátečního symbolu a názvy speciálních terminálů, jsou-li využívány (identifikátor, řetězec, znak, celé číslo, reálné číslo, konec souboru, konec řádku, mezera, tabulátor). To se provádí v sekci *Language*.

Sekce *Language* slouží pro základní nastavení aplikace. Téměř všechny položky mají svou implicitní hodnotu (kromě povinných položek, jako je např. počáteční symbol gramatiky), takže stačí uvádět jen ty položky, kde jejich implicitní hodnota nevyhovuje. Určuje se zde velikost struktur (např. velikost tabulky terminálů, neterminálů atd.), velikost výhledu gramatiky, příznak, je-li gramatika lokálně nebo permanentně rozšiřitelná, přeskakují-li se bílé znaky, má-li se provádět zotavení z chyb, má-li se použít locale a jaké, tvary oddělovačů řetězců, znaků a komentářů. Podrobný popis jednotlivých položek je v podkapitole 3.4.1.

Pro potřeby lexikálního analyzátoru je ještě třeba uvést tvary definovaných terminálů. Podle svého tvaru jsou terminály rozděleny do dvou skupin – klíčová slova (keywords) a symboly (symbols). Klíčová slova splňují požadavky kladené na identifikátory, tj. začínají písmenem, a pak obsahují písmena nebo číslice, případně je možné explicitně povolit znak podtržítko (pomocí příznaku *UnderlineChar* v sekci *Language*); symboly jsou řetězce nealfanumerických znaků. Jeden terminál může mít i více než jeden tvar. Definice tvaru terminálu se provádí v sekcích *Keywords* (pro klíčová slova) a *Symbols* (pro symboly).

Sémantická analýza je v ExTra reprezentována pomocí sémantických akcí (semantic actions). Sémantické akce – poté, co je jejich název definován v sekci *Actions* – se zapisují přímo do pravé části pravidel, a to na libovolnou pozici (kromě začátku levě rekurzivních pravidel), nemusí tedy být pouze na konci pravidla.

Co daná sémantická akce dělá, určuje její sémantický program (pokud

žádný nemá, nedělá nic). Sémantické programy se píší v jazyce, který se podobá asembleru, s využitím níže uvedených sémantických instrukcí (podkapitola 3.3). Programy využívají svůj vlastní sémantický zásobník, jehož velikost je definována v sekci *Language*.

Sémantické instrukce mohou mít žádný, jeden, dva nebo tři operandy. Rozlišují se čtyři adresovací módy (tj. kam se ukládá nebo odkud se čte hodnota), které lze pro operandy použít:

- registr
  - `r0`, `r1`, `r2`;
- zásobník
  - `s0` – vrchol zásobníku,
  - `s1` – místo pod vrcholem zásobníku,
  - `s2` – místo pod `s1`;
- generátor
  - `g0`, `g1`, `g2`, `g3`;
- přímá hodnota.

Přímá hodnota může být v libovolném počtu operandů. Generátory a generování čtveřic jsou zde připraveny pro případné rozšiřování aplikace a zachování kompatibility s připravovaným za běhu rozšiřitelným překladačem (ExTra lze tak využívat pro ladění jeho gramatik).

Operandy instrukcí jsou podle možností čtení a zápisu rozděleny do tří skupin. Každá instrukce má příznaky odpovídající jejím operandům, které určují, do jaké skupiny daný operand patří:

- *src* – operand pouze ke čtení (zdroj),
- *dst* – operand pouze pro zápis (cíl),
- *acc* – je povolen zápis i čtení operandu (akumulátor).

	Zásobník	Registr	Generátor	Přímá hodnota
src	$\surd$	$\surd$	$\times$	$\surd$
dst	$\times$	$\surd$	$\surd$	$\times$
acc	$\surd$	$\surd$	$\times$	$\times$

Tabulka 3.1: Kombinace adresovacích módů a příznaků sémantických instrukcí

Kombinace adresovacích módů a příznaků ukazuje tabulka 3.1 – povolené kombinace označuje $\surd$, nepovolené $\times$. ExTra kontroluje správnost použití operandů instrukcí a v případě nepovolené kombinace ohlásí chybu.

Hodnoty operandů sémantických instrukcí jsou následujících typů (v závorce jsou uvedeny odpovídající typy jazyka C):

- celé číslo (`int`),
- reálné číslo (`double`),
- znak (`char`),
- řetězec (`char*`),
- identifikátor (`char*`).

Uživatel si sám spravuje počet kopií jednotlivých hodnot – pro tento účel mají sémantické instrukce, které pracují s hodnotami, kopírovací (končí na „c“, od slova copy) a přesouvací verzi (končí na „m“, od slova move); navíc je k dispozici instrukce pro uvolnění hodnoty. Proč je důležité mít možnost spravovat počet kopií, souvisí s tím, jak jsou hodnoty reprezentovány, to je popsáno v podkapitole 4.4 Sémantická analýza.

Poznámka 3.1.
Hodnoty všech typů, pokud se objeví ve vstupním textu, mají omezenou délku (v počtu znaků). Pro typ znak je samozřejmě maximální délka 1 znak, pro ostatní typy je maximální délka v současné verzi programu nastavena na 200 znaků (`MaxValLen`). Dále platí omezení, která plynou z použitých datových typů jazyka C.

Poslední instrukce každého sémantického programu by měla být instrukce *return*, která ukončí provádění daného programu. Jinak bude provádění

pokračovat dalšími instrukcemi, dokud se nenarazí na *return* nebo konec proudu instrukcí (instrukce všech programů jsou uloženy za sebou). Toto způsobí neočekávané provádění instrukcí následujícího programu, což typicky vede k chybě; proto ExTra kontroluje, obsahují-li sémantické programy instrukci *return*, a pokud ne, vypíše varování. Na druhou stranu někdy může být chybějící instrukce *return* záměrem, a to tehdy, má-li několik sémantických akcí stejné programy – takový program stačí napsat pouze jednou, programy pro ostatní akce vložit nad něj a nechat je prázdné (především bez instrukce *return*, varování o její nepřítomnosti lze potlačit vložením speciální instrukce *pragma-noreturn*).

Příklad 3.4 (Sémantické akce se stejným programem)**.**

```
#akce1
pragma-noreturn
#akce2
pragma-noreturn
#akce3
```

instrukce programu

```
return
```

□

Výstupní jazyk se definuje pomocí výstupních terminálů (out terminals) a sémantických akcí. Každý výstupní terminál musí být definován v příslušné sekci konfiguračního souboru (sekce *OutTerms*) a mít přiřazen nějaký tvar (sekce *OutShapes*), kterým může být libovolný řetězec. Na rozdíl od klíčových slov a symbolů výstupní terminál smí mít pouze jeden tvar. Výstupní terminály pak mohou být součástí pravidel stejně jako „klasické“ terminály a neterminály gramatiky. Pokud analyzátor při zpracování gramatiky narazí na výstupní terminál, napíše do souboru s výstupním textem jeho tvar. Sémantické akce se použijí, je-li potřeba vypsat např. identifikátor, řetězec nebo číslo, které pocházejí ze souboru se vstupním textem.

3.3 Instrukční soubor

Následuje seznam sémantických instrukcí s popisem významu. Operandy jsou označeny příznaky *src*, *dst* nebo *acc* podle toho, jestli z nich lze číst, psát nebo obojí. Jestliže má instrukce více operandů, oddělují se od sebe čárkou (`,`). V případě přímého operandu se řetězce uzavírají do uvozovek (`""`), znaky do apostrofů (`''`), čísla se píší přímo. Pokud je potřeba zapsat znak uvozovka, prefixuje se zpětným lomítkem (`\"`), obdobně znak apostrof (`\'`); znak zpětné lomítko se píše zdvojeně (`\\`). Příklad 3.5 ilustruje způsob psaní přímých operandů na instrukci *put*, která slouží k zápisu do souboru s výstupním textem.

Příklad 3.5 (Zápis přímých operandů)**.**

```
put "retezec"
put "uvozovky \"\" v retezci"
put "zpetne lomitko \\ v retezci"
put 'c'
put '\''
put '\\'
put 1
put 1.2
```

□

3.3.1 Práce se zásobníkem

Odebírání hodnot ze zásobníku (*pop*) a vkládání hodnot na zásobník (*push*). Instrukce mají kopírovací (končí na „c“, od slova copy) a přesouvací verzi (končí na „m“, od slova move); to souvisí s reprezentací hodnot (viz podkapitola 4.4 Sémantická analýza). Kopírování nebo přesouvání se týká hodnot (jestli zůstane na původním místě zachována, nebo ne), ukazatel na vrchol zásobníku se vždy změní: instrukce *pop*/*popc*/*popm* jej o jedna sníží, instrukce *push*/*pushc*/*pushm* jej o jedna zvýší.

pop dst Zkopíruje hodnotu z vrcholu zásobníku do *dst*, hodnota na vrcholu zásobníku zůstane zachována (implicitně stejné chování jako *popc*).

popc dst Zkopíruje hodnotu z vrcholu zásobníku do *dst*, hodnota na vrcholu zásobníku zůstane zachována.

popm dst Přesune hodnotu z vrcholu zásobníku do *dst*, hodnota na vrcholu zásobníku nebude zachována.

push src Zkopíruje hodnotu ze *src* na vrchol zásobníku (implicitně stejné chování jako *pushc*).

pushc src Zkopíruje hodnotu ze *src* na vrchol zásobníku.

pushm src Přesune hodnotu ze *src* na vrchol zásobníku.

3.3.2 Správa kopií

Kopie, přesun a uvolnění hodnoty. Rozdíl mezi kopií a přesunem je vysvětlen v podkapitole 4.4 Sémantická analýza.

copy dst, src Zkopíruje hodnotu ze *src* do *dst*.

move dst, src Přesune hodnotu ze *src* do *dst*.

del dst Uvolní *dst*.

3.3.3 Vstup a výstup

Získání hodnoty ze souboru se vstupním textem, výpis hodnoty nebo bílých znaků do souboru s výstupní textem. Instrukce *put* má kopírovací (končí na „c“, od slova copy) a přesouvací verzi (končí na „m“, od slova move); to souvisí s reprezentací hodnot (viz podkapitola 4.4 Sémantická analýza). Instrukce *fetch* vždy přesouvá.

put src Vypíše do souboru s výstupním textem hodnotu ze *src*, hodnota v *src* zůstane zachována (implicitně stejné chování jako *putc*).

putc src Vypíše do souboru s výstupním textem hodnotu ze *src*, hodnota v *src* zůstane zachována.

putm src Vypíše do souboru s výstupním textem hodnotu ze *src*, hodnota v *src* nebude zachována.

fetch dst Vyzvedne hodnotu z fronty lexikálního analyzátoru a uloží ji do *dst*.

writeln Vloží do souboru s výstupním textem konec řádku.

spaces src Vloží do souboru s výstupním textem mezery, jejich počet je dán číselnou hodnotou v *src*.

3.3.4 Aritmetické operace

Základní aritmetické operace pro celá a reálná čísla. Název odpovídá dané aritmetické operaci, prefix „r“ označuje instrukce pracující s reálnými čísly.

add acc, src Celočíselné sčítání: $acc = acc + src$ (neboli $acc += src$).

sub acc, src Celočíselné odčítání: $acc = acc - src$ (neboli $acc -= src$).

mul acc, src Celočíselné násobení: $acc = acc \times src$ (neboli $acc *= src$).

div acc, src Celočíselné dělení: $acc = acc \div src$ (neboli $acc /= src$).

mod acc, src Zbytek po celočíselném dělení: $acc = acc\ mod\ src$ (neboli $acc\ \%= src$).

inc acc Přičtení jedničky: $acc = acc + 1$ (neboli $acc++$).

dec acc Odečtení jedničky: $acc = acc - 1$ (neboli $acc--$).

radd acc, src Reálné sčítání: $acc = acc + src$ (neboli $acc += src$).

rsub acc, src Reálné odčítání: $acc = acc - src$ (neboli $acc -= src$).

rmul acc, src Reálné násobení: $acc = acc \times src$ (neboli $acc *= src$).

rdiv acc, src Reálné dělení: $acc = acc \div src$ (neboli $acc /= src$).

3.3.5 Konverze

Konverze mezi hodnotou typu reálné číslo a celé číslo, identifikátor a řetězec. Prefix „conv“ znamená conversion, pak následuje výchozí typ („r“ pro reálná čísla, „i“ pro celá čísla, „id“ pro identifikátory nebo „str“ pro řetězce), dále předložka „to“ a nakonec cílový typ.

convrtoi dst, src Konverze reálné hodnoty na celočíselnou (desetinná část se usekne).

convitor dst, src Konverze celočíselné hodnoty na reálnou.

convidtostr dst, src Konverze identifikátoru na řetězec.

convstrtoid dst, src Konverze řetězce na identifikátor.

3.3.6 Bitové operace

Základní bitové operace, všechny pracují s celočíselnými operandy. Název odpovídá dané logické funkci, resp. „sh“ znamená shift, „l“ left a „r“ right.

and acc, src Bitové and: $acc = acc \wedge src$ (neboli $acc \,\&= src$).

or acc, src Bitové or: $acc = acc \vee src$ (neboli $acc \mid= src$).

xor acc, src Bitové xor: $acc = acc\ xor\ src$ (neboli $acc\ \hat{}= src$).

not acc Bitová negace: $acc = \neg acc$.

shl acc, src Bitový posun hodnoty v *acc* doleva o *src* bitů ($acc\ <<= src$).

shr acc, src Bitový posun hodnoty v *acc* doprava o *src* bitů ($acc\ >>= src$).

3.3.7 Řízení rozšíření a redukce

Zahájení a případně ukončení rozšíření nebo redukce jazyka (obecně změny jazyka). Prefix „perm“ znamená permanent a označuje instrukce pro permanentní změny jazyka, prefix „loc“ znamená local a označuje instrukce pro lokální změny jazyka.

Instrukce *permdef*-*permuse* a *locdef*-*locuse*-*locend* musí být správně uzávorkovány; navíc po prvním použití *locdef* již nelze používat *permdef* a *permuse* (jazyk se mění nejprve permanentně a pak již pouze lokálně).

Více informací viz podkapitola 3.5 Použití rozšíření a redukce.

permdef Začátek permanentní změny jazyka – nyní mohou následovat instrukce s prefixem *def*/*undef*/*set*. Tuto instrukci lze použít jen v případě permanentního rozšiřování (tj. příznak *PermExtensible* v sekci *Language* musí být nastaven na Yes).

permuse Změněný jazyk se stane aktuálním. Tuto instrukci lze použít jen v případě permanentního rozšiřování (tj. příznak *PermExtensible* v sekci *Language* musí být nastaven na Yes), a byla-li před tím volána instrukce *permdef*.

locdef Začátek lokální změny jazyka – nyní mohou následovat instrukce s prefixem *def/undef/set*. Tuto instrukci lze použít jen v případě lokálního rozšiřování (tj. příznak *LocExtensible* v sekci *Language* musí být nastaven na Yes).

locuse Změněný jazyk se stane aktuálním. Tuto instrukci lze použít jen v případě lokálního rozšiřování (tj. příznak *LocExtensible* v sekci *Language* musí být nastaven na Yes), a byla-li před tím volána instrukce *locdef*.

locend Konec změny jazyka, návrat k předchozímu jazyku. Opouštěný jazyk je zahozen (pokud to nebyl původní jazyk). Tuto instrukci lze použít jen v případě lokálního rozšiřování (tj. příznak *LocExtensible* v sekci *Language* musí být nastaven na Yes), a byla-li před tím volána instrukce *locuse*.

3.3.8 Rozšíření

Pro všechny zde uvedené instrukce platí, že je lze použít jen v případě permanentního nebo lokálního rozšiřování (tj. příznaky *LocExtensible* nebo *PermExtensible* v sekci *Language* musí být nastaveny na Yes), a byla-li před tím volána instrukce *permdef* nebo *locdef*. Název je tvořen prefixem „def“ a pak následuje zkratka sekce konfiguračního souboru, do které se bude přidávat nový prvek.

defterm src Definice nového terminálu (v *src* je jeho název jako řetězec).

defnont src Definice nového neterminálu (v *src* je jeho název jako řetězec).

defoutterm src Definice nového výstupního terminálu (v *src* je jeho název jako řetězec).

defact src Definice nové sémantické akce (v *src* je její název jako řetězec).

defprod src Definice nového pravidla (v *src* je jeho tvar jako řetězec).

defkwd src1, src2 Definice nového klíčového slova (v *src1* je název již definovaného terminálu, v *src2* jeho tvar, oba jako řetězec).

defsymb src1, src2 Definice nového symbolu (v *src1* je název již definovaného terminálu, v *src2* jeho tvar, oba jako řetězec).

defoutshp src1, src2 Definice tvaru výstupního terminálu (v *src1* je název již definovaného výstupního terminálu, v *src2* jeho tvar, oba jako řetězec).

defins src Definice nové sémantické instrukce, vloží se na konec proudu instrukcí (v *src* je její název včetně operandů jako řetězec).

deftermkwd src Definice nového terminálu a klíčového slova současně, klíčové slovo bude mít stejný tvar jako je název terminálu (v *src* je název terminálu jako řetězec). Tato instrukce je zkratka za instrukce *defterm* a *defkwd*, protože situace, kdy má klíčové slovo stejný tvar jako název terminálu, nastává často.

3.3.9 Redukce

Pro všechny zde uvedené instrukce platí, že je lze použít jen v případě permanentního nebo lokálního rozšiřování (tj. příznaky *LocExtensible* nebo *PermExtensible* v sekci *Language* musí být nastaveny na Yes), a byla-li před tím volána instrukce *permdef* nebo *locdef*. Název je tvořen prefixem „undef“ a pak následuje zkratka sekce konfiguračního souboru, ze které se bude odebírat.

undefterm src Oddefinování terminálu (v *src* je jeho název jako řetězec).

undefnont src Oddefinování neterminálu (v *src* je jeho název jako řetězec).

undefoutterm src Oddefinování výstupního terminálu (v *src* je jeho název jako řetězec).

undefact src Oddefinování sémantické akce (v *src* je její název jako řetězec).

undefprod src Oddefinování pravidla (v *src* je jeho tvar jako řetězec).

undefkwd src Oddefinování klíčového slova (v *src* je jeho tvar jako řetězec).

undefsymb src Oddefinování symbolu (v *src* je jeho tvar jako řetězec).

undefoutshp src Oddefinování tvaru výstupního terminálu (v *src* je název výstupního terminálu jako řetězec).

undefins src Oddefinování sémantické instrukce (v *src* je její pořadové číslo v proudu instrukcí jako celé číslo).

3.3.10 Změny nastavení

Instrukce *setlang* slouží ke změně nastavení položek sekce *Language* z konfiguračního souboru, jejich kompletní přehled viz podkapitola 3.4.1. Lze ji použít jen v případě permanentního nebo lokálního rozšiřování (tj. příznaky *LocExtensible* nebo *PermExtensible* v sekci *Language* musí být nastaveny na Yes), a byla-li před tím volána instrukce *permdef* nebo *locdef*. Všechny změny se projeví až v novém jazyce, tedy po volání instrukce *permuse* nebo *locuse*.

Rozhodli jsme se nepovolit změnu velikosti struktur (tj. položky *Terminals*, *Nonterminals*, *OutTerms*, *Actions*, *Productions*, *Keywords*, *SymbNodes*, *SemInsts*, *Edges*, *Views*, *Nodes*, *SemStackSize*, *LangStackSize*, *CallStackSize*), protože by to znamenalo nutnost realokací.

setlang src1, src2 Změna nastavení položky ze sekce *Language* (v *src1* je název proměnné, v *src2* její nová hodnota, oba jako řetězec).

Příklad 3.6 (Změna oddělovače komentářů za běhu)**.**
Ukázka možného použití instrukce *setlang*: změna oddělovače komentářů za běhu. Provede-li se sémantická akce *changecomment*, vše mezi znaky { a } se bude považovat za komentář. Předpokládá se, že příznak *PermExtensible* je nastaven na Yes.

```
#changecomment
permdef
setlang "CommentStart1", "{"
setlang "CommentEnd1", "}"
permuse
return
```

□

3.3.11 Skoky

Podmíněné a nepodmíněné skoky. Délka skoku je dána celým číslem (kladné znamená skok v proudu instrukcí vpřed, záporné skok vzad); skok +1 odpovídá instrukci, která následuje za danou instrukcí skoku.

Název instrukcí podmíněných skoků je tvořen porovnávací operací, která se použije v podmínce, a sufixem „t“ jako true (skok se provede, když podmínka platí), resp. „f“ jako false (skok se provede, když podmínka neplatí). Dvě skupiny instrukcí podmíněných skoků (se sufixem „f“, se sufixem „t“) jsou implementovány pro pohodlné formulování podmínek, ve skutečnosti si navzájem odpovídají: *eqt* = *nef*, *net* = *eqf*, *gtt* = *lef*, *get* = *ltf*, *ltt* = *gef*, *let* = *gtf*.

jmp src Nepodmíněný skok o *src* instrukcí (v *src* je celé číslo).

call src Provedení sémantické akce (v *src* je její název jako řetězec). Počet vnořených volání instrukce *call* je omezen velikostí zásobníku této instrukce, jeho velikost se nastavuje pomocí položky *CallStackSize* v sekci *Language*.

eqt src1, src2, src3 Podmíněný skok: jestliže *src1* = *src2*, skoč o *src3* instrukcí (všechny operandy jsou celá čísla).

net src1, src2, src3 Podmíněný skok: jestliže *src1* $\neq$ *src2*, skoč o *src3* instrukcí (všechny operandy jsou celá čísla).

gtt src1, src2, src3 Podmíněný skok: jestliže *src1* $>$ *src2*, skoč o *src3* instrukcí (všechny operandy jsou celá čísla).

get src1, src2, src3 Podmíněný skok: jestliže *src1* $\geq$ *src2*, skoč o *src3* instrukcí (všechny operandy jsou celá čísla).

ltt src1, src2, src3 Podmíněný skok: jestliže *src1* $<$ *src2*, skoč o *src3* instrukcí (všechny operandy jsou celá čísla).

let src1, src2, src3 Podmíněný skok: jestliže *src1* $\leq$ *src2*, skoč o *src3* instrukcí (všechny operandy jsou celá čísla).

eqf src1, src2, src3 Podmíněný skok: jestliže neplatí *src1* = *src2*, skoč o *src3* instrukcí (všechny operandy jsou celá čísla).

nef src1, src2, src3 Podmíněný skok: jestliže neplatí *src1* $\neq$ *src2*, skoč o *src3* instrukcí (všechny operandy jsou celá čísla).

gtf src1, src2, src3 Podmíněný skok: jestliže neplatí *src1* $>$ *src2*, skoč o *src3* instrukcí (všechny operandy jsou celá čísla).

gef src1, src2, src3 Podmíněný skok: jestliže neplatí *src1* $\geq$ *src2*, skoč o *src3* instrukcí (všechny operandy jsou celá čísla).

ltf src1, src2, src3 Podmíněný skok: jestliže neplatí *src1* $<$ *src2*, skoč o *src3* instrukcí (všechny operandy jsou celá čísla).

lef src1, src2, src3 Podmíněný skok: jestliže neplatí *src1* $\leq$ *src2*, skoč o *src3* instrukcí (všechny operandy jsou celá čísla).

3.3.12 Externí aplikace

Instrukce *exe* umožňuje spustit libovolnou externí aplikaci, tato se provede jako nový proces a ExTra čeká, dokud tento proces neskončí.

exe src Spuštění externí aplikace, v *src* je uvedena cesta jako řetězec.

Externí aplikace má přístup k hodnotám uloženým v registrech a na sémantickém zásobníku – může je číst i měnit. Hodnoty se před jejím zavoláním uloží do souboru a po jejím provedení se aktualizují. Tímto mechanismem byla vyřešena potřeba komunikace mezi ExTra a externí aplikací. Pro práci se zmíněným souborem byla vyvinuta knihovna *exelib*, následuje seznam jejích funkcí se stručným popisem:

- `int exelib_init(int _StackSize)` – zahájení práce s knihovnou, provede inicializace a nahraje hodnoty do paměti, tato funkce musí být zavolána na začátku práce s knihovnou,
  - `_StackSize` – maximální velikost sémantického zásobníku;
- `int exelib_close(void)` – ukončení práce s knihovnou, uloží hodnoty zpět do souboru;
- `int exelib_get_reg(int num, ExeLibValType *value)` – získání hodnoty z registru,
  - `num` – číslo registru,

  - `value` – výstupní parametr s hodnotou;

- `int exelib_set_reg(int num, ExeLibValType value)` – uložení hodnoty do registru,
  - `num` – číslo registru,
  - `value` – vstupní parametr s hodnotou;
- `int exelib_pop(ExeLibValType *value)` – získání hodnoty z vrcholu zásobníku,
  - `value` – výstupní parametr s hodnotou;
- `int exelib_push(ExeLibValType value)` – uložení hodnoty na vrchol zásobníku,
  - `value` – vstupní parametr s hodnotou;
- `int exelib_isempty(int *result)` – test prázdnosti zásobníku,
  - `result` – výstupní parametr s výsledkem (0 znamená neprázdný, jinak prázdný);
- `int exelib_check_reg_type(int num, ExeLibDataType *type)` – získání typu hodnoty z registru,
  - `num` – číslo registru,
  - `type` – výstupní parametr s typem hodnoty;
- `int exelib_check_stack_type(ExeLibDataType *type)` – získání typu hodnoty z vrcholu zásobníku,
  - `type` – výstupní parametr s typem hodnoty;
- `char *exelib_get_error_text(void)` – vrací textový popis poslední chyby.

Téměř všechny funkce (kromě `exelib_get_error_text`) vracejí nulu, je-li vše v pořádku; nenulovou hodnotu, pokud nastane chyba. Textový popis poslední chyby lze získat pomocí zmíněné `exelib_get_error_text`.

ExTra kontroluje návratovou hodnotu externí aplikace: jestliže doběhne bez chyby, musí předat nulu, nenulová návratová hodnota signalizuje chybu

(hodnota -1 je rezervována pro chybu při spouštění externí aplikace, proto by ji neměla vracet).

Instrukce *exe* je jednoduchý prostředek, jak doplňovat nové sémantické instrukce bez zásahu do zdrojových kódů ExTra. Ukázku použití čtenář nalezne na přiloženém CD, příklad *exelib*.

Poznámka 3.2.
Instrukce *exe* je implementována pomocí funkce knihovny jazyka C `_spawnl`, která je platformově závislá. Implementace byla překládána na platformě Win32 pomocí překladače Microsoft Visual Studio 2005 Professional Edition.

3.3.13 Ostatní

return Ukončení provádění programu sémantické akce.

nop „No operation“ – nedělá nic kromě zvýšení čítače aktuální instrukce (program counter).

3.3.14 Speciální instrukce

Zde uvedené instrukce slouží k úpravě chování aplikace a pro ladící účely. Všechny názvy začínají prefixem „pragma-“.

pragma-noreturn Potlačí varování o nepřítomnosti instrukce *return* v daném sémantickém programu.

pragma-dump src Vypíše na standardní výstup aktuální jazyk a nastavení aplikace (dump). V *src* je řetězec s informačním textem, který slouží k identifikaci daného výpisu, pokud je tato instrukce použita vícekrát.

pragma-breakpoint src Pozastaví provádění sémantického programu a celého analyzátoru (breakpoint). V *src* je řetězec s informačním textem, který slouží k identifikaci daného pozastavení, pokud je tato instrukce použita vícekrát.

pragma-reg src Vypíše na standardní výstup aktuální obsah registrů. V *src* je řetězec s informačním textem, který slouží k identifikaci tohoto výpisu, pokud je tato instrukce použita vícekrát.

pragma-stack src Vypíše na standardní výstup aktuální obsah sémantického zásobníku. V *src* je řetězec s informačním textem, který slouží k identifikaci tohoto výpisu, pokud je tato instrukce použita vícekrát.

3.4 Konfigurační soubor

Konfigurační soubor je spolu se zdrojovým textem vstupem programu. Obsahuje definice vstupního i výstupního jazyka, včetně všech nastavení aplikace, popisu sémantických akcí a instrukcí.

Jedná se o jednoduchý textový soubor, který je rozdělen do 10 sekcí. Každá sekce začíná řádkem s názvem sekce uvedeným v hranatých závorkách (např. `[Language]`). Některé sekce jsou povinné, jiné lze případně vynechat. Pořadí sekcí je však pevně dané a musí být zachováno. Soubor může libovolně obsahovat prázdné řádky nebo mezery (přeskočí se), řádek začínající středníkem (`;`) se chápe jako komentář.

Přehled sekcí je uveden v tabulce 3.2, jejich podrobný popis následuje v dalších podkapitolách.

Pořadí	**Název sekce**	**Povinná/nepovinná**
1	Language	povinná
2	Terminals	povinná
3	Nonterminals	povinná
4	OutTerms	nepovinná
5	Actions	nepovinná
6	Productions	povinná
7	Keywords	nepovinná
8	Symbols	nepovinná
9	OutShapes	nepovinná
10	Instructions	nepovinná

Tabulka 3.2: Přehled sekcí konfiguračního souboru

Poznámka 3.3.
Délka řádku konfiguračního souboru je v současné verzi programu omezena na 300 znaků (`MaxCfgLineLen`); názvy terminálů, neterminálů, výstupních terminálů a sémantických akcí na 25 znaků (`MaxGramNameLen`); tvary klíčových slov na 25 znaků (`MaxKwdShapeLen`); tvary výstupních terminálů na

50 znaků (`MaxOutShapeLen`); délka pravidla na 250 znaků (`MaxProdLen`); záznamy, které se dělí na levou a pravou část (tj. záznamy sekce *Language*, pravidla, tvary výstupních terminálů, klíčová slova a symboly), mají navíc levou část omezenou na 50 znaků (`MaxCfgLeftPartLen`).

Poznámka 3.4.
Členění konfiguračního souboru vychází z původní implementace [Žem02b] a bylo ponecháno mimo jiné kvůli zachování kompatibility.

Začněme ukázkou: příklad 3.7 zobrazuje konfigurační soubor pro gramatiku aritmetického výrazu, jejíž definice byla uvedena v kapitole o přívětivé analýze (příklad 2.1). Pro jednoduchost jsme z gramatiky odstranili podporu identifikátorů. Aplikace ExTra, které bude předložen takový konfigurační soubor, určí, patří-li vstup do jazyka generovaného gramatikou aritmetického výrazu, a navíc spočítá výsledek výrazu.

Příklad 3.7 (Konfigurační soubor pro gramatiku aritmetického výrazu)**.**

```
[Language]
  Name = ArithmeticExpression
  Lookahead = 1
  StartingSymbol = S
  IntName = num
  EofName = eof
  Terminals = 10
  Nonterminals = 10
  Actions = 10
  Productions = 10
  Edges = 100
  Views = 100
  Nodes = 100
  SymbNodes = 10
  SemInsts = 100
  SemStackSize = 100

[Terminals]
  num
  eof
  left_parenthesis
  right_parenthesis
```

```
  plus
  star

[Nonterminals]
  S
  E
  T
  F

[Actions]
  print
  getnum
  add
  mul

[Productions]
  S::=[E]{print}
  E::=[E](plus)[T]{add}
  E::=[T]
  T::=[T](star)[F]{mul}
  T::=[F]
  F::=(num){getnum}
  F::=(left_parenthesis)[E](right_parenthesis)

[Symbols]
  left_parenthesis = (
  right_parenthesis = )
  plus = +
  star = *

[Instructions]
  #print
  pop r0
  put r0
  return

  #getnum
  fetch r0
```

```
push r0
return

#add
pop r0
add s0, r0
return

#mul
pop r0
mul s0, r0
return
```

□

3.4.1 Sekce Language

Sekce *Language* obsahuje nastavení aplikace, definice speciálních symbolů a velikostí struktur.

Jednotlivé položky jsou vždy na samostatném řádku a mají tvar:

proměnná = hodnota.

Hodnotou může být v závislosti na konkrétní proměnné:

- Yes/No (např. `LocExtensible = Yes`),
- číslo (např. `Lookahead = 1`),
- znak (např. `StrDelim = "`) nebo
- řetězec (např. `StrName = str`).

Jak je vidět z příkladů v závorkách, hodnoty typu znak ani řetězec nejsou uzavřeny do žádných oddělovačů. U názvů proměnných nezáleží na velikosti písmen, u hodnot ano.

Sekce je povinná, ne však všechny její položky; vždy záleží na konkrétní gramatice, kterou konfigurační soubor popisuje, jaké využívá prostředky a jak je potřebuje inicializovat. Vždy je nutné uvést počáteční symbol a velikosti struktur pro prvky, které jsou v dané gramatice obsaženy (terminály,

neterminály, výstupní terminály, sémantické akce, pravidla a s tím související hrany a uzly stromů pravidel a výhledy, klíčová slova, symboly, sémantické instrukce a s tím související velikost sémantického zásobníku). Gramatika může, ale nemusí využívat speciální terminály (s tím související oddělovače), rozšíření, komentáře apod.

Takže zatímco počáteční symbol gramatiky (*StartingSymbol*) je nutno vždy uvést, např. název terminálu pro identifikátory (*IdName*) není potřeba uvádět, pokud gramatika s identifikátory vůbec nepracuje.

Každá položka by měla být v sekci *Language* uvedena nejvýše jednou. V případě, že je některá položka uvedena vícekrát, použije se ta, která je napsána jako poslední.

Následuje popis všech položek, které se mohou v sekci *Language* objevit.

Velikosti struktur

Terminals Maximální počet terminálů. Hodnota typu číslo, implicitní nastavení je 10.

Nonterminals Maximální počet neterminálů. Hodnota typu číslo, implicitní nastavení je 10.

OutTerms Maximální počet výstupních terminálů. Hodnota typu číslo, implicitní nastavení je 0.

Actions Maximální počet sémantických akcí. Hodnota typu číslo, implicitní nastavení je 0.

Productions Maximální počet pravidel gramatiky. Hodnota typu číslo, implicitní nastavení je 10.

Keywords Maximální počet klíčových slov. Hodnota typu číslo, implicitní nastavení je 0.

SymbNodes Maximální počet uzlů ve stromě symbolů. Hodnota typu číslo, implicitní nastavení je 0.

SemInsts Maximální počet sémantických instrukcí. Hodnota typu číslo, implicitní nastavení je 0.

Edges Maximální počet hran stromů pravidel. Hodnota typu číslo, implicitní nastavení je 100.

Views Maximální počet prvků stromů výhledů. Hodnota typu číslo, implicitní nastavení je 100.

Nodes Maximální počet prvků stromů uzlů. Hodnota typu číslo, implicitní nastavení je 100.

SemStackSize Velikost sémantického zásobníku. Hodnota typu číslo, implicitní nastavení je 0.

LangStackSize Velikost zásobníku jazyků. Ovlivňuje počet zanoření lokálních rozšíření a redukcí. Hodnota typu číslo, implicitní nastavení je 0.

CallStackSize Velikost zásobníku pro instrukci *call*. Ovlivňuje počet zanoření volání sémantických akcí pomocí instrukce *call*. Hodnota typu číslo, implicitní nastavení je 0.

Speciální terminály a počáteční symbol

StartingSymbol Název počátečního symbolu gramatiky vstupního jazyka. Tento symbol (neterminál) musí být definován v sekci *Nonterminals*. Hodnota typu řetězec a je povinná.

IntName Název terminálu pro celá čísla. Tento terminál musí být definován v sekci *Terminals*. Specifikace tohoto názvu implikuje nastavení Yes pro příznak *UseInts*. Hodnota typu řetězec.

RealName Název terminálu pro reálná čísla. Tento terminál musí být definován v sekci *Terminals*. Specifikace tohoto názvu implikuje nastavení Yes pro příznak *UseReals* a je nutno zadat také *DecimalPoint*. Hodnota typu řetězec.

IdName Název terminálu pro identifikátory. Tento terminál musí být definován v sekci *Terminals*. Specifikace tohoto názvu implikuje nastavení Yes pro příznak *UseIds*. Hodnota typu řetězec.

StrName Název terminálu pro řetězce. Tento terminál musí být definován v sekci *Terminals*. Specifikace tohoto názvu implikuje nastavení Yes pro příznak *UseStrings* a je nutno zadat také *StrDelim*. Hodnota typu řetězec.

CharName Název terminálu pro znaky. Tento terminál musí být definován v sekci *Terminals*. Specifikace tohoto názvu implikuje nastavení Yes pro příznak *UseChars* a je nutno zadat také *CharDelim*. Hodnota typu řetězec.

EofName Název terminálu pro znak konec souboru (konec vstupu). Tento terminál musí být definován v sekci *Terminals*. Hodnota typu řetězec.

NewLineName Název terminálu pro znak konec řádku. Tento terminál musí být definován v sekci *Terminals*. Hodnota typu řetězec.

SpaceName Název terminálu pro znak mezera. Tento terminál musí být definován v sekci *Terminals*. Hodnota typu řetězec.

TabName Název terminálu pro znak tabulátor. Tento terminál musí být definován v sekci *Terminals*. Hodnota typu řetězec.

Příznaky pro speciální terminály

UseIds Určuje, budou-li se rozeznávat identifikátory. Hodnota typu Yes/No, implicitní nastavení je No. Nastaví se automaticky na Yes, když je zadán název terminálu pro identifikátory (*IdName*).

UseInts Určuje, budou-li se rozeznávat celá čísla. Hodnota typu Yes/No, implicitní nastavení je No. Nastaví se automaticky na Yes, když je zadán název terminálu pro celá čísla (*IntName*).

UseReals Určuje, budou-li se rozeznávat reálná čísla. Hodnota typu Yes/No, implicitní nastavení je No. Nastaví se automaticky na Yes, když je zadán název terminálu pro reálná čísla (*RealName*).

UseChars Určuje, budou-li se rozeznávat znaky. Hodnota typu Yes/No, implicitní nastavení je No. Nastaví se automaticky na Yes, když je zadán název terminálu pro znaky (*CharName*).

UseStrings Určuje, budou-li se rozeznávat řetězce. Hodnota typu Yes/No, implicitní nastavení je No. Nastaví se automaticky na Yes, když je zadán název terminálu pro řetězce (*StrName*).

SkipWhiteChars Určuje, budou-li se v souboru se vstupním textem přeskakovat bílé znaky (konec řádku, mezera, tabulátor). Pokud se nepřeskakují, hlásí se jako příslušné terminály (*NewLineName*, *SpaceName*,

TabName). Hodnota typu Yes/No, implicitní nastavení je Yes, tj. bílé znaky se přeskakují.

Oddělovače

DecimalPoint Tvar oddělovače desetinné části reálných čísel v souboru se vstupním textem (jeho tvar v souboru s výstupním textem se řídí podle locale). Musí být zadáno při používání reálných čísel (*RealName*, *UseReals*). Hodnota typu znak, implicitní nastavení je „.“.

StrDelim Tvar oddělovače řetězců v souboru se vstupním textem. Musí být zadáno při používání řetězců (*StrName*, *UseStrings*). Pokud je potřeba zapsat do řetězce znak *StrDelim*, prefixuje se zpětným lomítkem (*StrDelim*); znak zpětné lomítko se píše zdvojeně (\\). Hodnota typu znak, implicitní nastavení je „"“.

CharDelim Tvar oddělovače znaků v souboru se vstupním textem. Musí být zadáno při používání znaků (*CharName*, *UseChars*). Pokud je potřeba zapsat znak *CharDelim* jako znak, prefixuje se zpětným lomítkem (*CharDelim*); znak zpětné lomítko se píše zdvojeně (\\). Hodnota typu znak, implicitní nastavení je „'“.

OutStrDelim Tvar oddělovače řetězců v souboru s výstupním textem. Hodnota typu znak, implicitní nastavení je „"“.

OutCharDelim Tvar oddělovače znaků v souboru s výstupním textem. Hodnota typu znak, implicitní nastavení je „'“.

Rozšíření a redukce

PermExtensible Určuje, je-li jazyk permanentně rozšiřitelný (tj. lze používat rozšíření a redukci s permanentní platností). Hodnota typu Yes/No, implicitní nastavení je No.

LocExtensible Určuje, je-li jazyk lokálně rozšiřitelný (tj. lze používat rozšíření a redukci s lokální platností). Hodnota typu Yes/No, implicitní nastavení je No.

Extensible Určuje, je-li jazyk permanentně a lokálně rozšiřitelný (tj. lze používat rozšíření a redukci s permanentní a lokální platností). Jedná

se o zkratku za příznaky *PermExtensible* a *LocExtensible*. Hodnota typu Yes/No, implicitní nastavení je No.

Výhled

Lookahead Velikost výhledu analyzátoru. Hodnota typu číslo, implicitní nastavení je 1.

Komentáře

CommentStart1 Tvar začátku blokového komentáře v souboru se vstupním textem (první typ blokového komentáře). Hodnota typu řetězec.

CommentEnd1 Tvar konce blokového komentáře v souboru se vstupním textem (první typ blokového komentáře). Hodnota typu řetězec.

CommentStart2 Tvar začátku blokového komentáře v souboru se vstupním textem (druhý typ blokového komentáře). Hodnota typu řetězec.

CommentEnd2 Tvar konce blokového komentáře v souboru se vstupním textem (druhý typ blokového komentáře). Hodnota typu řetězec.

CommentLine Tvar začátku řádkového komentáře v souboru se vstupním textem. Hodnota typu řetězec.

Ostatní

Name Název jazyka, který je definován tímto konfiguračním souborem. Název má pouze informační charakter, používá se např. ve výpise jazyka a nastavení aplikace (dump). Hodnota typu řetězec, implicitní nastavení je „DefaultName“.

ErrorRecovery Určuje, má-li se provádět zotavení z chyb. Hodnota typu Yes/No, implicitní nastavení je No.

PutBadChar Určuje, má-li se při zotavení z chyby „neznámý znak ve vstupním textu“ tento znak napsat do souboru s výstupním textem. Hodnota typu Yes/No, implicitní nastavení je Yes.

UseLocale Určuje, má-li se použít locale při čtení souboru se vstupním textem. Hodnota typu Yes/No, implicitní nastavení je No.

Locale Název locale, které se má použít při čtení souboru se vstupním textem. Hodnota typu řetězec, který odpovídá parametru locale funkce jazyka C `setlocale` (např. „english“, „czech“), implicitní nastavení je „C“ (standardní locale, které se rovná situaci bez použití locale).

AllAheads Určuje, mají-li se vytvářet uzly ve stromech pravidel na všech místech (i když tam není větvení). Hodnota typu Yes/No, implicitní nastavení je No.

Extended Určuje, má-li se používat rozšířená přívětivá gramatika, to znamená oslabení kontroly konfliktů. Je-li nějaké pravidlo prefixem jiného pravidla a je-li možné pokračovat oběma způsoby, pak dostane přednost to delší pravidlo. Pokud by v takovém případě nebyla oslabena kontrola konfliktů, gramatika by byla odmítnuta již při počítání výhledů. Typickým příkladem použití rozšířené přívětivé gramatiky je příkaz if-then-else. Hodnota typu Yes/No, implicitní nastavení je No.

CaseSensitive Určuje, záleží-li v souboru se vstupním textem na velikosti písmen (u klíčových slov). Hodnota typu Yes/No, implicitní nastavení je No.

UnderlineChar Určuje, je-li podtržítko (_) považováno za znak (a může tedy být součástí identifikátoru). Hodnota typu Yes/No, implicitní nastavení je No.

Následující tři položky jsou užitečné zejména při tvoření gramatiky a jejím ladění, uživatel píšící gramatiku se tak může soustředit na jednotlivé části a ostatní ignorovat.

IgnoreValues Určuje, mají-li se ignorovat hodnoty speciálních terminálů (identifikátory, celá čísla, reálná čísla, znaky a řetězce). V případě nastavení na Yes nebude lexikální analyzátor hodnoty speciálních terminálů předávat. Hodnota typu Yes/No, implicitní nastavení je No.

IgnoreActions Určuje, mají-li se ignorovat sémantické akce. V případě nastavení na Yes se nebudou sémantické akce provádět (jako by byl jejich program prázdný). Hodnota typu Yes/No, implicitní nastavení je No.

IgnoreOutTerms Určuje, mají-li se ignorovat výstupní terminály. V případě nastavení na Yes se nebudou vypisovat tvary výstupních terminálů do souboru s výstupním textem (jako by byl jejich tvar prázdné slovo). Hodnota typu Yes/No, implicitní nastavení je No.

Poznámka 3.5.
Maximální velikost výhledu je omezena na 3 (`MaxAhead`), jde o čistě technické omezení a lze jej samozřejmě změnit předefinováním konstanty `MaxAhead` a následnou kompilací. Maximální délka oddělovače komentářů je omezena na 5 znaků (`MaxComDelimLen`) a maximální velikost číselných hodnot na 10 milionů (`MaxInt`).

Poznámka 3.6.
V konfiguračním souboru je povoleno pro pojmenovávání objektů používat libovolné znaky (kromě `[`, `]`, `(`, `)`, {, }, `<`, `>`, `=`, `:`, `#` a `;`), případně s použitím locale, avšak může tím být ztracena možnost využít tento konfigurační soubor pro konstruktor přívětivých analyzátorů KindCons [Žem02a] (viz kapitola 5 Příbuzné práce).

3.4.2 Sekce Terminals

Sekce *Terminals* obsahuje definice terminálů gramatiky vstupního jazyka.

Musí zde být definovány speciální terminály (jsou-li použity), na které se odkazuje sekce *Language* (*IntName*, *RealName*, *IdName*, *StrName*, *CharName*, *EofName*, *NewLineName*, *SpaceName*, *TabName*).

Definice terminálu sestává pouze z názvu definovaného terminálu, na každém řádku je uvedena jedna definice. Na velikosti písmen v názvech terminálů záleží.

Sekce *Terminals* je povinná.

3.4.3 Sekce Nonterminals

Sekce *Nonterminals* obsahuje definice neterminálů gramatiky vstupního jazyka.

Musí zde být definován počáteční symbol, na který se odkazuje sekce *Language* (*StartingSymbol*).

Definice neterminálu sestává pouze z názvu definovaného neterminálu, na každém řádku je uvedena jedna definice. Na velikosti písmen v názvech neterminálů záleží.

Sekce *Nonterminals* je povinná.

3.4.4 Sekce OutTerms

Sekce *OutTerms* obsahuje definice výstupních terminálů.

Definice výstupního terminálu sestává pouze z názvu definovaného výstupního terminálu, na každém řádku je uvedena jedna definice. Na velikosti písmen v názvech výstupních terminálů záleží.

Sekce *OutTerms* je nepovinná, gramatika nemusí používat výstupní terminály.

3.4.5 Sekce Actions

Sekce *Actions* obsahuje definice sémantických akcí.

Definice sémantické akce sestává pouze z názvu definované sémantické akce, na každém řádku je uvedena jedna definice. Na velikosti písmen v názvech sémantických akcí záleží.

Sekce *Actions* je nepovinná, gramatika nemusí používat sémantické akce.

3.4.6 Sekce Productions

Sekce *Productions* obsahuje definice pravidel gramatiky vstupního jazyka.

Pravidlo se skládá z levé a pravé části, které jsou odděleny řetězcem dvě dvojtečky a rovnítko (`::=`). V levé části je název neterminálu (není uzavřen do žádných závorek); v pravé pak kombinace:

- názvů neterminálů (uzavřeny do hranatých závorek: `[]`),
- názvů terminálů (uzavřeny do kulatých závorek: `()`),
- názvů sémantických akcí (uzavřeny do složených závorek: {}) a
- názvů výstupních terminálů (uzavřeny do špičatých závorek: `<>`).

neterminál `::=` ... `[`*neterminál*`]` ... `(`*terminál*`)` ... {*akce*}
... `<`*výstupní_terminál*`>` ...

Obrázek 3.2: Tvar definice pravidla

Pravá část může být také prázdná, což symbolizuje přepis na prázdné slovo (λ). Pravidla s levou rekurzí musí mít na začátku pravé strany příslušný neterminál, nesmí zde být sémantická akce nebo výstupní terminál.

Každá definice pravidla je uvedena na samostatném řádku. Na velikosti písmen záleží.

Sekce *Productions* je povinná.

3.4.7 Sekce Keywords

Sekce *Keywords* obsahuje definice klíčových slov.

Definice klíčového slova má tvar:

název_terminálu = *klíčové_slovo*.

Definice jsou uvedeny vždy každá na samostatném řádku. Na velikosti písmen u názvů terminálů záleží, u klíčového slova se rozlišování velkých a malých písmen řídí nastavením příznaku *CaseSensitive* ze sekce *Language*.

Sekce *Keywords* je nepovinná, gramatika nemusí definovat klíčová slova.

3.4.8 Sekce Symbols

Sekce *Symbols* obsahuje definice symbolů.

Definice symbolu má tvar:

název_terminálu = *symbol*.

Symbol může obsahovat jeden i více znaků, tyto znaky lze uvést přímo nebo pomocí ASCII kódů. ASCII kódy se prefixují písmenem „A“, např. A43A43 definuje symbol „++“.

Definice jsou uvedeny vždy každá na samostatném řádku. Na velikosti písmen u názvů terminálů záleží.

Sekce *Symbols* je nepovinná, gramatika nemusí definovat symboly.

3.4.9 Sekce OutShapes

Sekce *OutShapes* obsahuje definice tvarů výstupních terminálů.

Definice tvaru výstupního terminálu vypadá následovně:

název_výstupního_terminálu = *tvar*.

Podobně jako u symbolů i zde lze pro popis tvaru používat ASCII kódy.

Definice jsou uvedeny vždy každá na samostatném řádku. Na velikosti písmen záleží.

Sekce *OutShapes* je nepovinná, gramatika nemusí definovat tvary výstupních terminálů.

3.4.10 Sekce Instructions

Sekce *Instructions* obsahuje programy pro sémantické akce; programy se skládají ze sémantických instrukcí.

Každý program je uveden hlavičkou, což je název sémantické akce, který má navíc na začátku znak mříž (#). Nezáleží na pořadí, v jakém byly definovány sémantické akce v sekci *Actions* a v jakém jsou teď pro ně definovány programy. Po hlavičce programu následují jednotlivé sémantické instrukce s případnými operandy.

#*název_akce*
instrukce
instrukce
⋮

Obrázek 3.3: Program pro sémantickou akci

Každý program by měl končit instrukcí *return*, neboť jeho vykonávání skončí právě při zpracování této instrukce. Pokud pro sémantickou akci neexistuje žádný program, nedělá tato nic (její program se považuje za prázdný).

Každá instrukce nebo název akce jsou uvedeny na samostatném řádku. Na velikosti písmen záleží.

Sekce *Instructions* je nepovinná, gramatika nemusí definovat programy pro sémantické akce.

3.5 Použití rozšíření a redukce

Mluvíme-li o rozšíření a redukci jazyka, znamená to také rozšíření a redukci jeho definice, která je uvedena v daném konfiguračním souboru. V této souvislosti jsou tedy oba pojmy záměnné. Vzhledem k tomu, že konfigurační soubor definuje jak vstupní, tak výstupní jazyk, zasahují rozšíření a redukce oba tyto jazyky.

Rozšíření a redukce jazyka se provádí pomocí specializovaných sémantických instrukcí. Kdekoliv na pravé straně libovolného pravidla (kromě začátku levě rekurzivního pravidla) může být sémantická akce, která tyto instrukce používá. Navíc musí být rozšíření a redukce povoleny nastavením příznaků *PermExtensible* (permanentní, tj. od místa použití až do konce

analýzy) nebo *LocExtensible* (lokální, tj. od místa použití do explicitně řečeného ukončení, viz dále) v sekci *Language* na Yes.

Rozšíření a redukce jsou chápány jako speciální případy obecné změny jazyka – buď se do jazyka přidávají nové vlastností, nebo se ty již existující odebírají. A tak se na redukci i rozšiřování používají stejné příznaky (*PermExtensible*, *LocExtensible*) i řídící instrukce. Rozdíl je tedy až ve vlastních změnách jazyka – jestli se vlastnosti přidávají (pak je to rozšíření) nebo odebírají (pak je to redukce).

Základem změny jazyka jsou řídící instrukce *permdef* a *permuse* v případě permanentní změny, *locdef*, *locuse* a *locend* v případě lokální změny.

Instrukcí *permdef*, resp. *locdef*, se analyzátoru oznámí, že od této chvíle se bude aktuální jazyk měnit (a nezáleží na tom, jestli rozšiřovat, redukovat nebo třeba obojí). K rozšiřování slouží série instrukcí začínajících prefixem *def*, pro redukci jsou instrukce začínající prefixem *undef*. Je možné přidávat a odebírat terminály, neterminály, výstupní terminály, akce, pravidla, klíčová slova, symboly, tvary výstupních terminálů a instrukce. Pomocí instrukce *setlang* lze měnit některá nastavení ze sekce *Language*.

Všechny změny se provádí pouze na kopii jazyka a analýza stále probíhá podle aktuálního jazyka, dokud se nepoužije instrukce *permuse*, resp. *locuse*. Touto instrukcí dojde k přepnutí analyzátoru na změněný jazyk. Nyní samozřejmě mohou následovat další vnořená volání instrukce *permdef*, resp. *locdef*. V případě lokálních vnořených změn se jejich pořadí udržuje na zásobníku jazyků, jehož velikost se nastavuje pomocí položky *LangStackSize* v sekci *Language*.

Poznámka 3.7.
Celkový počet jazyků je omezen na 100 (`MaxLang`) – do toho je započítán i originální jazyk definovaný v konfiguračním souboru a pomocné jazyky, které se využívají při změně (viz podkapitola 4.5 Implementace rozšíření a redukce). To omezuje počet vnoření lokálních změn jazyka, neboť je potřeba pamatovat si předchozí jazyky pro případ návratu.

Pokud se jedná o lokální změnu, může po instrukci *locuse* následovat instrukce *locend*. Tato zahodí aktuální jazyk a vrátí se k předchozímu, tedy k tomu, který byl aktuální před posledním použitím *locuse*.

Každá z instrukcí *permdef*, *permuse* a *locdef*, *locuse*, *locend* může být v programu jiné sémantické akce; důležité je pouze pořadí, ve kterém jsou prováděny. Nejprve musí být změna zahájena (*permdef* nebo *locdef*), poté lze aktuální jazyk vyměnit (*permuse* nebo *locuse*), dále mohou následovat

případné vnořené změny, nakonec v případě lokální změny návrat k předchozímu jazyku (*locend*). Permanentní a lokální změny nelze míchat – nejprve probíhají permanentní změny (žádná, jedna nebo více), pak lokální. Když proběhne nějaká lokální změna jazyka, nelze již provádět permanentní změny.

```
permdef
⋮
instrukce s prefixem def/undef/set
⋮
permuse
```

Obrázek 3.4: Řídící instrukce pro permanentní změnu jazyka

```
locdef
⋮
instrukce s prefixem def/undef/set
⋮
locuse
⋮
locend
```

Obrázek 3.5: Řídící instrukce pro lokální změnu jazyka

Zatímco přidávání vlastností je omezeno pouze paměťovými nároky a nutností zachovat třídu gramatiky (musí zůstat přívětivá), na redukci jsou kladeny ještě další požadavky: není možné odebrat pravidlo, které zrovna analyzátor analyzuje (a tedy ve kterém byla sémantická akce, jež provedla redukci). Stejně tak není možné odebrat pravidla, která jsou rozpracovaná a kam se tedy analyzátor ještě vrátí (to jsou ta pravidla, ze kterých se analyzátor postupně do toho aktuálního dostal a nyní jsou uložena na jeho zásobníku). Je zakázáno oddefinovat prvky (terminály, neterminály, výstupní terminály, akce), na které se odkazuje nějaké existující pravidlo. Z toho plyne, že není možné oddefinovat mimo jiné počáteční symbol, protože na ten se odkazuje pravidlo, kterým začala analýza (a to je buď aktuální, nebo je na

zásobníku). Pokud byly použity nějaké speciální terminály, musí být zachovány. Klíčová slova a symboly se nesmí odkazovat na oddefinované terminály. Výše uvedené podmínky ExTra kontroluje a pokud nejsou splněny, ohlásí chybu.

Nyní uvedeme jednoduchý příklad rozšíření a redukce; použijeme konfigurační soubor pro gramatiku aritmetického výrazu uvedený v příkladu 3.7. Za běhu přidáme operaci odčítání a odebereme operaci sčítání. Kromě toho byla původní gramatika mírně pozměněna, a to tak, aby soubor se vstupním textem mohl obsahovat seznam více aritmetických výrazů. Kdekoliv v tomto seznamu lze uvést klíčová slova `begin` (změna jazyka) a `end` (návrat k původnímu jazyku), mezi kterými půjde odčítat, ale nepůjde sčítat.

Příklad 3.8 (Konfigurační soubor s rozšířením a redukcí)**.**

```
[Language]
  Name = ArithmeticExpression2
  Lookahead = 1
  LocExtensible = Yes
  StartingSymbol = S
  IntName = num
  EofName = eof
  Terminals = 10
  Nonterminals = 10
  Actions = 10
  Productions = 20
  Edges = 100
  Views = 200
  Nodes = 100
  SymbNodes = 10
  Keywords = 10
  SemInsts = 100
  SemStackSize = 100


[Terminals]
  num
  eof
  left_parenthesis
  right_parenthesis
```

```
  plus
  star
  begin
  end

[Nonterminals]
  S
  E
  T
  F
  EXP_LIST
  EXP

[Actions]
  print
  getnum
  add
  mul
  new_exp
  orig_exp

[Productions]
  S::=[EXP_LIST]
  EXP_LIST::=[EXP_LIST][EXP]
  EXP_LIST::=[EXP]
  EXP::=(begin){new_exp}[EXP_LIST](end){orig_exp}
  EXP::=[E]{print}
  E::=[E](plus)[T]{add}
  E::=[T]
  T::=[T](star)[F]{mul}
  T::=[F]
  F::=(num){getnum}
  F::=(left_parenthesis)[E](right_parenthesis)


[Keywords]
  begin = begin
  end = end
```

```
[Symbols]
  left_parenthesis = (
  right_parenthesis = )
  plus = +
  star = *

[Instructions]
  #print
  pop r0
  put r0
  spaces 1
  return

  #getnum
  fetch r0
  push r0
  return

  #add
  pop r0
  add s0, r0
  return

  #mul
  pop r0
  mul s0, r0
  return

  #new_exp
  locdef
  defterm "minus"
  defsymb "minus", "-"
  defact "sub"
  defins "#sub"
  defins "pop r0"
  defins "sub s0, r0"
  defins "return"
```

```
defprod "E::=[E](minus)[T]{sub}"
undefprod "E::=[E](plus)[T]{add}"
locuse
return

#orig_exp
locend
return
```

□

3.6 Další vlastnosti

3.6.1 Zotavení z chyb

Aplikace ExTra obsahuje základní zotavení z vybraných chyb, které mohou nastat jednak při čtení konfiguračního souboru, jednak při analýze vstupního textu.

Implicitně je vypnuté – nastane-li nějaká chyba, zpracování konfiguračního souboru či analýza jsou přerušeny, vypíše se chybové hlášení a program skončí s příslušným chybovým kódem.

Zotavení z chyb se zapne nastavením příznaku *ErrorRecovery* v sekci *Language* na Yes. Nastane-li chyba, pro kterou není známé zotavení, je chování obdobné jako v minulém případě. Pokud však existuje pro chybu zotavení, provede se, vypíše se chybové hlášení a program se pokusí pokračovat dál. Nakonec program podá hlášení o celkovém počtu chyb.

Konfigurační soubor je řádkově orientovaný, každý řádek se zpracovává víceméně nezávisle na ostatních, a tak chyby v konfiguračním souboru se řeší přeskočením chybného řádku. Dojde-li k chybě v lexikálním analyzátoru, chybné části vstupního textu se přeskočí. Chyby syntaktického analyzátoru se řeší vkládáním očekávaných terminálů nebo přeskočením částí pravidel, které neodpovídají vstupu a podle kterých nelze analyzovat dál.

Při zotavení z lexikální chyby „neznámý znak ve vstupním textu“ může být výhodné napsat tento znak do souboru s výstupním textem. ExTra to tak implicitně dělá; pokud by se však toto chování pro danou úlohu nehodilo, lze jej potlačit nastavením příznaku *PutBadChar* v sekci *Language* na No.

Toto zotavení z chyb bylo motivováno zejména použitím aplikace na texty psané v přirozeném jazyce, kde se může vyskytnout pro gramatiku neznámý

znak, ale jeho přeskočení nemusí negativně ovlivnit analýzu zbytku vstupního textu.

3.6.2 Bílé znaky

Implicitně ExTra při analýze vstupního textu automaticky přeskakuje bílé znaky (mezera, tabulátor, konec řádku). Někdy však může být informace o bílých znacích důležitá – ať už se analyzuje nějaký řádkově orientovaný jazyk nebo třeba text v přirozeném jazyce.

Nastavením příznaku *SkipWhiteChars* v sekci *Language* na No se zruší implicitní přeskakování bílých znaků a lexikální analyzátor, narazí-li na bílý znak, bude vracet odpovídající terminál. Tyto terminály musí být samozřejmě definovány v sekci *Terminals* a je rovněž nutné uvést jejich názvy v sekci *Language*, jedná se o položky:

- *NewLineName* (terminál pro konec řádku),
- *SpaceName* (terminál pro mezeru) a
- *TabName* (terminál pro tabulátor).

3.6.3 Locale

Změna locale umožňuje jednoduše používat ve vstupním textu znaky z vyšší části ASCII tabulky.

Pro práci s locale slouží v sekci *Language* příznak *UseLocale*, který určuje, má-li se locale používat, a položka *Locale*, kde se uvádí konkrétní locale. Hodnota položky *Locale* je řetězec, který odpovídá parametru locale funkce jazyka C `setlocale`, tedy např. „english“ nebo „czech“ (viz manuál k funkci standardní knihovny jazyka C: `char *setlocale(int category, const char *locale)`).

Neřekne-li se jinak (tj. locale se nepoužije), implicitně se považují za znaky, které tvoří identifikátory a klíčová slova, pouze písmena anglické abecedy a číslice, ostatní znaky se berou jako symboly. Takže např. slovo „jméno“ lexikální analyzátor rozebere jako terminál identifikátor s hodnotou „jm“, terminál symbol s tvarem „é“ (je-li takový symbol definovaný, jinak nastane chyba) a terminál identifikátor s hodnotou „no“. S nastaveným locale na „czech“ již lexikální analyzátor vrátí očekávaný terminál identifikátor s hodnotou „jméno“.

Poznámka 3.8.
Použití locale je platformově závislé, zejména tvar jazykového řetězce (položka *Locale*). Implementace byla překládána na platformě Win32 pomocí překladače Microsoft Visual Studio 2005 Professional Edition.

Kapitola 4

Popis implementace

V této kapitole popíšeme vnitřní strukturu aplikace a postupy implementace. Na závěr se zaměříme na implementaci rozšíření a redukce.

Tato práce je součástí většího projektu, proto některé části byly převzaty z původní implementace [Žem02b] a v mnoha případech jsme brali ohledy na možnosti dalšího rozvoje. Některá rozhodnutí tedy byla ovlivněna tak, aby bylo později možné snadno implementovat další plánované funkce.

Aplikace ExTra byla naprogramována v jazyce C s pomocí vývojového prostředí Microsoft Visual Studio 2005 Professional Edition.

Vstupním bodem je funkce `main`. Zde se zpracují parametry z příkazové řádky – přepínače `d`, `q`, `w` a `l`, vstupní a výstupní soubory. Poté se provedou požadované akce, nakonec se uvolní datové struktury a aplikace skončí. Takto vypadá ideální scénář, v případě chyby může nastat konec dříve.

Běh aplikace lze rozdělit na dvě samostatné fáze:

1. zpracování konfiguračního souboru (funkce `LangLearn`),
2. analýza vstupního textu (funkce `CompileC`).

Ve první fázi se přečte celý konfigurační soubor a získané informace se uloží do datové struktury jazyka (`LangRec`), která je zde pro tento účel vytvořena a inicializována. Další informace, které se v této fázi dopočítají, budou rovněž uloženy do struktury jazyka. A tak jsou v ní veškeré informace o jazyce a nastavení aplikace – terminály, neterminály, pravidla, sémantické akce a instrukce, klíčová slova a symboly, výstupní terminály, výhledy, stromy pravidel, informace o speciálních terminálech, nastavení příznaků jazyka, tvary oddělovačů, velikosti struktur a jiné.

Jazykem se zde rozumí veškeré informace pocházející z konfiguračního souboru a další z nich odvozené – tedy ty informace, které se použijí k analýze vstupního textu a vygenerování výstupního textu. Jedná se tedy o zobecnění pojmů definice vstupního a výstupního jazyka.

Ve druhé fázi se nejprve vytvoří a inicializuje další důležitá datová struktura – kompilační struktura (`CompRec`). V té jsou všechny informace potřebné za běhu – právě používaný jazyk, další jazyky související s rozšiřováním a redukcí, sémantický zásobník a zásobník jazyků (souvisí s lokálním rozšiřováním a redukcí), registry, generátory, struktury lexikálního analyzátoru a další. Zejména se ale v této fázi spustí přívětivý analyzátor (funkce `Parse`), který provede analýzu vstupního textu a vygeneruje výstupní text.

Za běžných okolností probíhá nejprve první fáze a pak druhá fáze; jsou tedy odděleny. Pokud se však použije rozšíření nebo redukce, dojde k jejich prolínání – ve druhé fázi se tak opět využijí některé části z první fáze, které slouží ke zpracování informací o jazyce.

Obě fáze jsou podrobněji popsány v následujících podkapitolách.

Nyní se ještě jednou zmíníme o struktuře jazyka. Záznamy o jednotlivých objektech jsou ukládány do tabulek (polí), indexy v těchto tabulkách – kódy (handly) – dané objekty jednoznačně identifikují. A tak se v záznamech při odkazování na jiné záznamy používají právě tyto kódy. Díky tomu lze datové struktury mimo jiné snadno kopírovat, což je výhodné především proto, že případná rozšíření se provádí na kopii struktury jazyka (viz podkapitola 4.5 Implementace rozšíření a redukce). Některé objekty jsou ve své tabulce uloženy pouze jako lineární seznam záznamů (to platí pro terminály, neterminály, sémantické akce, klíčová slova, výstupní terminály, pravidla, sémantické instrukce); jiné tvoří stromy (strom pravidel) nebo trie (symboly, výhledy, uzly). Stromy a trie jsou ukládány binarizovaně (odkazy typu „další“ a „alternativa“).

4.1 Zpracování konfiguračního souboru

Jak již bylo řečeno v předchozí kapitole, konfigurační soubor je jednoduchý textový soubor, rozdělený do 10 sekcí, každá sekce začíná svým jménem v hranatých závorkách. Celkově je řádkově orientovaný – každá hodnota leží na samostatném řádku. Při jeho zpracování se samozřejmě těchto vlastností využívá.

Zpracování konfiguračního souboru provádí funkce `LangLearn`, která postupně čte jednotlivé řádky. Funkce pracuje stavově – pamatuje si aktu-

ální sekci; v rámci jedné sekce jsou řádky zpracovávány nezávisle na sobě. Obsahuje-li řádek název sekce, zavolá ukončovací funkci předchozí sekce (post-funkce), poté inicializační funkci této sekce (pre-funkce) a prohlásí ji za aktuální. Kromě různých speciálních případů (komentář, prázdný řádek) jinak zavolá funkci, která má na starost zpracování řádku v dané sekci (line-funkce).

Line-funkce rozebere zadaný řádek a získané informace uloží do struktury jazyka. Podrobněji pro sekci *Language* poznamená hodnoty položek; pro sekce *Terminals*, *Nonterminals*, *OutTerms* a *Actions* vytvoří záznamy o definovaných objektech; pro sekci *Productions* vytvoří záznam o pravidle a současně jej vloží do příslušného stromu pravidel neterminálu z levé strany (ke každému neterminálu patří dva stromy pravidel: pro pravidla bez levé rekurze a pro pravidla s levou rekurzí, jak to bylo popsáno v kapitole 2); pro sekce *Keywords*, *Symbols* a *OutShapes* uloží zadané tvary a pro sekci *Instructions* vloží zadanou instrukci do proudu instrukcí.

Úkolem pre-funkcí, které jsou volány vždy před zpracováním své sekce, je inicializace a nastavení implicitních hodnot. Ve skutečnosti pre-funkci v současné verzi programu využívá pouze sekce *Language*. Celý tento mechanismus byl navržen obecně a může být plně využit při dalším rozšiřování aplikace. Za zmínku stojí, že existuje ještě čtvrtý druh funkce – missing-funkce – která se volá, pokud příslušná sekce chybí (využívá se pro kontrolu přítomnosti povinných sekcí).

Post-funkce jsou volány po zpracování své sekce, a tak provádí kontroly získaných údajů a dopočítávají další. Důležitá je zejména post-funkce sekce *Productions*, neboť zde se vypočítají výhledy gramatiky a uzly, které jsou součástí stromů pravidel.

Výhledy se počítají postupně pro všechny neterminály gramatiky (funkce `ViewMakeAll`), využívají se k tomu stromy pravidel pro levě rekurzivní i nerekurzivní pravidla, navíc také pomocné stromy hran, které následují za neterminály ve stromech pravidel. Poté se projdou stromy pravidel všech neterminálů a doplní se do nich uzly (`NodeRebuild`), k jejich vytvoření jsou potřebné výhledy.

4.2 Lexikální analýza

Syntaktický analyzátor potřebuje mít možnost podívat se, jaké terminály se nacházejí v souboru se vstupním textem za aktuální pozicí (kolik terminálů dopředu vidí, udává velikost jeho výhledu). Aktuální terminál a něko-

lik následujících se udržují ve frontě v kompilační struktuře. Pokud je tato fronta prázdná nebo neobsahuje dostatečný počet terminálů, je volána hlavní funkce lexikálního analyzátoru `LexGetTerm`, která čte vstup a zařazuje do fronty další terminál; to se opakuje, dokud není výhled dostatečný.

Funkce `LexGetTerm` pracuje takto – přečte z aktuální pozice v souboru se vstupním textem jeden znak (ve skutečnosti jsou znaky přednačítány do další fronty v kompilační struktuře) a podle tohoto znaku se rozhoduje, jak postupovat dál:

1. Mají-li se přeskakovat bílé znaky a přečtený znak je bílý znak, přeskočí jej – to opakuje tak dlouho, dokud nenarazí na nějaký jiný než bílý znak.

2. Je-li přečtený znak konec souboru a má-li gramatika definovaný terminál pro konec souboru, vrátí tento terminál, jinak chyba.

3. Odpovídá-li přečtený znak (případně spolu s následujícími) oddělovači komentáře, přeskočí všechen další obsah, dokud nenarazí na koncový oddělovač komentáře (resp. konec řádku v případě řádkového komentáře).

4. Je-li přečtený znak písmeno (nebo podtržítko – záleží na nastavení příznaku *UnderlineChar* v sekci *Language*) a pak případně následují alfanumerické znaky (nebo podtržítko), porovná získaný řetězec s tvary klíčových slov. V případě shody vrátí terminál příslušného klíčového slova, jinak terminál pro identifikátor (jeho hodnotu uloží do fronty hodnot v kompilační struktuře). Pokud nejsou identifikátory v gramatice povoleny nebo byl řetězec příliš dlouhý, nastane chyba.

5. Je-li přečtený znak číslice a pak případně následují další číslice, vrátí terminál pro celé číslo, pokud je definován (hodnotu uloží do fronty hodnot v kompilační struktuře). Jsou-li povolena reálná čísla a mezi číslicemi se vyskytuje desetinný oddělovač, vrátí terminál pro reálná čísla (hodnotu uloží do fronty hodnot v kompilační struktuře). Opět mohlo dojít k chybě z důvodu nedefinovaného terminálu nebo překročení maximální délky.

6. Jsou-li povoleny řetězce a přečtený znak je oddělovač řetězců, čte další znaky, dokud nenarazí znovu na oddělovač řetězců. Vrátí terminál pro řetězce; v případě, že se oddělovač řetězců a znaků shodují, znaky

jsou v gramatice povoleny a slovo mělo délku jedna (bez případného zpětného lomítka), vrátí terminál pro znaky. Hodnotu uloží do fronty hodnot. Zde způsobí chybu výskyt konce souboru (konec souboru v řetězci) nebo překročení maximální délky.

7. Jsou-li povoleny znaky a přečtený znak je oddělovač znaků, přečte následující znak (nutno ošetřit speciální případy: oddělovač znaků, zpětné lomítko) a dále očekává znovu oddělovač znaků (není-li tu – chyba). Vrátí terminál pro znaky a hodnotu uloží do fronty hodnot.

8. Je-li přečtený znak bílý znak, vrátí příslušný terminál (musí být definován, jinak nastane chyba). Pokud se mají bílé znaky přeskakovat, přeskočily se již v bodě 1, takže sem se dostanou pouze v případě, že se přeskakovat neměly.

9. Nenastala-li žádná z předchozích možností (resp. chyba), prohledá se strom symbolů. V případě shody vrátí terminál příslušného symbolu.

10. Jinak chyba (neznámý znak ve vstupním textu) – znaku ze souboru se vstupním textem neodpovídá žádný terminál.

Další službou lexikálního analyzátoru je porovnání daného terminálu s aktuálním terminálem ze souboru se vstupním textem (`MatchTerm`). Tuto službu využívá syntaktický analyzátor v případě, že při procházení stromu pravidel narazí na hranu s terminálem – tento musí být shodný s aktuálním terminálem v souboru se vstupním textem. V opačném případě dojde k chybě a vstupní slovo je odmítnuto.

Kromě toho je samozřejmě třeba, aby lexikální analyzátor poskytoval hodnoty spojené se speciálními terminály. Jak bylo popsáno výše, ukládá je do fronty v kompilační struktuře (lze tomu zabránit nastavením příznaku *IgnoreValues* v sekci *Language* na Yes). Hodnoty z této fronty vyzvedává funkce `FetchVal`, ta je volána při zpracování instrukce *fetch*.

4.3 Syntaktická analýza

Syntaktický analyzátor, reprezentovaný funkcí `Parse`, tvoří základní část druhé fáze aplikace a celou analýzu vstupního textu řídí. Jedná se o implementaci přívětivého analyzátoru, podporujícího permanentní i lokální rozšíření a redukce, který byl představen v kapitole 2.

Syntaktický analyzátor pracuje tak, že prochází stromy pravidel jednotlivých neterminálů. Při svém rozhodování potřebuje vědět, jaké terminály následují v souboru se vstupním textem (výhled) – k tomu využívá lexikální analyzátor. Celá analýza začíná průchodem stromu pravidel počátečního symbolu, nejprve strom pro pravidla bez levé rekurze.

Strom pravidel je složen z hran a uzlů. Hrany jsou několika typů: hrana s terminálem, neterminálem, pravidlem, sémantickou akcí a výstupním terminálem. Průchod stromem znamená průchod jeho hran, uzly slouží k orientaci. Zpracování hrany stromu pravidel závisí na jejím typu:

Hrana s terminálem Přečte terminál ze vstupu a porovná jej s terminálem z hrany (využije služby lexikálního analyzátoru). Pokud se shodují, přejde ve stromě pravidel na následující hranu. Pokud se liší, nastane chyba, vstupní slovo je odmítnuto.

Hrana s neterminálem Rekurzivně přejde na kořen stromu pravidel daného neterminálu (strom pro pravidla bez levé rekurze). Vzhledem k tomu, že v tomto stromě (resp. dalších stromech, obsahoval-li neterminály) mohlo dojít k přepnutí jazyka, je nutné po dokončení rekurzivního zpracování aktualizovat proměnnou s jazykem, podle kterého se analyzuje. Současně zkontroluje, jestli nebyly neterminál, jehož strom pravidel se právě zpracovává, a aktuální hrana oddefinovány. Je-li vše v pořádku, pokračuje následující hranou.

Hrana s pravidlem Podle výhledu do levé rekurze ($DLRF$) zjistí, zda má pokračovat stromem pro pravidla s levou rekurzí stejného neterminálu, jehož strom pravidel se právě zpracovává. Pokud ne, končí analýza tohoto neterminálu – vrátí se z rekurze nebo, byl-li to počáteční symbol, končí celá analýza a vstup je přijat.

Hrana se sémantickou akcí Mají-li se zpracovávat sémantické akce, vykoná ji. Sémantická akce mohla přepnout jazyk, takže stejně jako u neterminálu je třeba aktualizovat proměnnou s jazykem, podle kterého se analyzuje, a provést kontroly. Pokračuje následující hranou.

Hrana s výstupním terminálem Mají-li se zpracovávat výstupní terminály, napíše do souboru s výstupním textem jeho tvar. Pokračuje následující hranou.

Chování syntaktického analyzátoru odpovídá popsanému přívětivému analyzátoru; navíc byly přidány sémantické akce a výstupní terminály.

Ve stromech pravidel může být samozřejmě větvení – stane se tak, když má více pravidel stejnou levou stranu, pak případně stejný prefix pravých stran. V místě větvení má hrana kromě odkazu na následující hranu také odkaz na alternativní hranu. Která hrana z alternativ je ta správná, pozná analyzátor podle výhledu vstupu a uzlů. Uzly jsou ve skutečnosti trie, ve kterých jsou uloženy výhledy pravidel a odkazy na příslušné hrany – takže porovná výhled vstupu s výhledy v uzlech. V případě shody pokračuje příslušnou hranou; jestliže žádný výhled v uzlu neodpovídá, nastane chyba.

Poznámka 4.1.
Syntaktický analyzátor je implementován pomocí rekurzivního volání funkce `Parse`. Hrozí tedy nebezpečí, že při rekurzi přeteče zásobník aplikace – to nenastane při použití levé rekurze, mohlo by to však nastat při použití např. pravé rekurze. Předpokládáme, že pro současné účely je tato implementace dostatečná, ale v produkční verzi by bylo vhodné ji předělat.

4.4 Sémantická analýza

Sémantickou analýzu tvoří sémantické akce a instrukce. Sémantické instrukce jsou uloženy pro všechny akce dohromady v proudu instrukcí. Každá sémantická akce má u sebe informaci o tom, kde v proudu instrukcí začíná její program. Konec programu je dán instrukcí *return*.

Když analyzátor přejde na hranu se sémantickou akcí, zjistí, kde v proudu instrukcí program akce začíná, a provádí jednotlivé instrukce, dokud nenarazí na instrukci *return* (resp. konec proudu).

Vykonání jedné instrukce je úkolem funkce `PerformInstr`. Na začátku zjistí aktuální hodnoty operandů (v případě přímého operandu je hodnota uložena u instrukcí, jinak má informaci o tom, jestli se má hledat v registru nebo na zásobníku; aktuální stav registrů a zásobníku je v kompilační struktuře). Nakonec provede výkonný kód dané instrukce.

Instrukce většinou představují primitivní operace, takže jejich výkonný kód v jazyce C bývá poměrně jednoduchý: zkontrolují se operandy a provede se daná akce (někdy odpovídá přímo nějakému operátoru jazyka C nebo se zavolá funkce, která ji provede). Složitější situace nastává u rozšíření a redukce, implementace těchto instrukcí je podrobněji rozebrána dále.

Další příklad obtížnější instrukce je *exe*. Kromě spuštění externí aplikace má za úkol uložit hodnoty všech registrů a celý sémantický zásobník do souboru, aby k těmto hodnotám mohla externí aplikace přistupovat, a poté

všechny hodnoty načíst ze souboru zpět do registrů a na zásobník.

Instrukce mají podle druhu svých operandů příznaky, za každý operand jeden. Příznak instrukce říká, jestli je možné z operandu číst (*src*), do operandu psát (*dst*) nebo z něj číst i do něj psát (*acc*). Program provádí kontrolu kombinací adresovacích módů (registr, zásobník, generátor, přímý operand) a příznaků ještě ve fázi načítání z konfiguračního souboru (ve funkci `SemAsmLine`). Kontroluje, je-li možné daný adresovací mód v operandu, kterému přísluší daný příznak, použít; nekontroluje, je-li operand ve výkonném kódu instrukce skutečně použit podle příznaku – to ovšem nijak nevadí, neboť výkonný kód píše programátor této aplikace (nehlídá sám sebe), použití konkrétního adresovacího módu volí uživatel v konfiguračním souboru (to hlídáno je).

Příznak *acc* byl zaveden zejména kvůli dvouoperandovým aritmetickým instrukcím, které si ze svého prvního operandu přečtou jednu ze vstupních hodnot a nakonec tam uloží výsledek. Vzhledem k tomu, že tímto prvním operandem bývá místo na zásobníku, byla povolena kombinace *acc* a zásobník, i když zásobník jinak nemůže být cílem. Jinou možností by bylo řešit tyto případy výjimkami z testů (pak by příznak neodpovídal skutečnému použití operandu) nebo použití přímo se zásobníkem zakázat a oba operandy vždy ukládat do registrů, po provedení operace výsledek explicitně ukládat na zásobník (stejný problém jako u předchozího, navíc uživatel by byl nucen explicitně předávat výsledek na zásobník, tj. celkově by použil více instrukcí), také by šlo použít třetí operand pro výsledek (neřešilo by použití se zásobníkem).

Jak již bylo řečeno v kapitole 3, správa kopií hodnot v operandech je ponechána na uživateli. Ten pomocí kopírovacích nebo přesouvacích instrukcí určuje počet kopií. Tato správa je důležitá zejména pro hodnoty typu řetězec nebo identifikátor, protože tyto jsou reprezentovány jako ukazatel na dynamicky alokovaná pole. Při kopírování takových hodnot se tato pole duplikují, tj. pro kopii se vytvoří nové pole se stejným obsahem (v kopii bude jiný ukazatel mířící na nové pole); při přesouvání se přesune pouze ukazatel (v kopii bude stejný ukazatel). Program sám neprovádí implicitní uvolňování; když už nejsou tyto alokace potřeba, uživatel je může uklidit – pro tento účel má k dispozici instrukci *del*. To se týká hodnot, které přišly z lexikálního analyzátoru; ne přímých operandů, protože ty jsou uloženy spolu s instrukcí a zůstanou po celou dobu běhu programu (pokud ovšem nebude daná instrukce oddefinována při nějaké redukci). Ostatní typy, tj. celé číslo, reálné číslo a znak, jsou reprezentovány přímo svou hodnotou, takže se přesouvá

nebo kopíruje tato hodnota, nic se nealokuje. V případě instrukce přesunu zůstane ve zdroji neinicializovaná hodnota.

Vzhledem k tomu, že přidávání nových sémantických instrukcí je jednou z možností, jak tuto aplikaci na úrovni zdrojových kódů snadno rozšiřovat, popíšeme zde, které části kódu je třeba změnit nebo doplnit:

- definovat nový identifikátor instrukce ve výčtu `InstOp`,
- aktualizovat číslo poslední instrukce `LastMnemo`,
- vytvořit nový uživatelský název instrukce v poli `Mnemos`,
- přidat novou `case` větev pro načítání operandů instrukce ve funkci `SemAsmLine`,
- definovat příznaky instrukce v poli `InstrFlags`,
- přidat `case` větev s výkonným kódem instrukce ve funkci `PerformInstr` (hodnoty operandů jsou připraveny v proměnných `src`, `src2`, `src3`, `dst` nebo `acc` podle zadaných příznaků instrukce).

Po provedení těchto šesti jednoduchých kroků a kompilaci se instrukce stane součástí aplikace a lze ji používat v konfiguračních souborech.

4.5 Implementace rozšíření a redukce

Rozšíření a redukce (změny jazyka) jsou implementovány prostřednictvím sémantických akcí. Využívá se vlastností přívětivých gramatik, které lze snadno rozšiřovat, aniž by se zásadně ovlivnila již existující část struktury. Důležitý je také fakt, že se pracuje s kódy objektů, což umožňuje snadné přechody mezi jednotlivými strukturami jazyků.

Změny jazyka lze samozřejmě vnořovat – zákaz vnořování by bylo přílišné a zbytečné omezení. Pro permanentní změnu by takový zákaz znamenal, že ji lze provést pouze jednou, protože každá následující změna jazyka navazuje na předchozí. Z implementačního pohledu to nepřináší žádnou režii navíc. V případě lokální verze je kvůli vnořování potřeba všechny opuštěné jazyky zachovávat a udržovat jejich pořadí na zásobníku (zásobník jazyků).

ExTra spravuje tabulku jazyků (resp. struktur jazyků), takže stejně tak jako jiné objekty (terminály, neterminály, ...) i jazyky mají své kódy.

Velikost tabulky jazyků je dána konstantou `MaxLang`, `MaxLang` je tedy maximální počet jazyků, které mohou v aplikaci existovat současně. Jazyky je možné kopírovat (funkce `LangCopy`), vytvářet nové (funkce `LangNew`) nebo existující mazat a tím uvolnit příslušný záznam v tabulce (funkce `LangFree`).

Podle toho, jak se s jazykem v dané chvíli pracuje nebo jak byl definován, mohou být některé jazyky označeny jako:

- aktuální – jazyk, podle kterého právě probíhá analýza, analyzátor si z něj bere všechny informace, které potřebuje v průběhu analýzy, zejména strom pravidel a výhledy;
- originální – jazyk, který byl definován v konfiguračním souboru, takto označený jazyk bude vždy zachován (při přechodu k novému jazyku se neuvolňuje[1]);
- pracovní – jazyk, ve kterém právě probíhají změny pomocí instrukcí s prefixem *def*/*undef*/*set*, z hlediska změn je následníkem aktuálního jazyka.

Kromě toho mohou být v tabulce jazyků ještě další jazyky, které nejsou aktuální, originální ani pracovní; to nastane při vnořených lokálních změnách, kdy je třeba udržovat opuštěné jazyky, jak bylo zmíněno výše.

V kompilační struktuře se udržují odkazy do tabulky jazyků na struktury aktuálního, originálního a pracovního jazyka – právě tyto odkazy přiřazují jazykům dané označení. Na začátku analýzy aktuální a originální jazyk splývají (analýza probíhá podle originálního jazyka) a odkaz na pracovní jazyk je neplatný (žádné rozšiřování ještě neprobíhá, a tak žádný jazyk v tabulce není označen jako pracovní).

Řídící instrukce rozšíření a redukce spravují pracovní jazyk a řeší vlastní přepínání mezi jednotlivými jazyky. Instrukce *permdef*, resp. *locdef*, ověří, je-li odkaz na pracovní jazyk neplatný (null). Pokud ne, znamená to, že je nějaký jazyk za pracovní označen, protože se již provádí změna jazyka (tzn. tato instrukce byla volána vícekrát za sebou, a to je chyba). Pokud je neplatný, zkopíruje aktuální jazyk a kopii označí jako pracovní jazyk. Analýza stále probíhá podle původního jazyka, ale nyní mohou následovat instrukce

[1]Do budoucna se počítá s tím, že by aplikace umožňovala překlad více souborů se vstupním textem podle stejného jazyka. Každý soubor může obsahovat změny jazyka, proto je výhodné zachovat originální jazyk, aby se nemusel načítat pokaždé znovu z konfiguračního souboru.

s prefixem *def*/*undef*/*set*, které veškeré své akce provádějí v pracovním jazyce.

Instrukce s prefixem *def* slouží k rozšiřování, tedy přidávání nových prvků do jazyka. V jejich implementaci lze s výhodou využít stejné mechanismy jako při zpracování konfiguračního souboru. Rozšíření o nové terminály, neterminály, výstupní terminály a jejich tvar, sémantické akce a klíčová slova znamená pouze vložení nového záznamu do příslušné tabulky ve struktuře jazyka. Nový symbol se vloží na příslušné místo do stromu symbolů (resp. vloží se tam odkaz na daný terminál), nová sémantická instrukce se zařadí na konec proudu instrukcí. Nejtěžší situace nastává v případě přidání nového pravidla – kromě záznamu do tabulky pravidel se musí ještě vložit nová větev do stromu pravidel. Původní hrany stromů pravidel se nemění, mohou být ovlivněny uzly. Ukázka rozšíření stromu pravidel je uvedena v podkapitole 2.5.

Instrukce s prefixem *undef* slouží k redukci, tedy odebírání existujících prvků z jazyka. Tady bylo nutné vytvořit ke zpracování konfiguračního souboru inverzní operace. Tyto operace vždy vyhledají daný objekt a označí jej za oddefinovaný (tj. název nebo tvar se nahradí prázdným řetězcem, u symbolů se odebere odkaz na terminál, oddefinovaná sémantická instrukce se nahradí instrukcí *nop*) – pro pozdější kontroly je třeba zachovat záznamy o oddefinovaných objektech. Nejobtížnější je opět odebírání pravidla, které vyžaduje vyjmutí příslušné větve ze stromu pravidel.

Konečně instrukce s prefixem *set* pouze přepíše příslušná nastavení ve struktuře jazyka.

Instrukcí *permuse*, resp. *locuse*, se ukončí změny dělané v pracovním jazyce. Pokud se prováděly nějaké redukce (tj. byla volána některá z instrukcí s prefixem *undef*), zkontroluje se konzistence jazyka (`LangStructCheck`):

- jsou-li použity speciální terminály, musí zůstat definované;
- nesmí být odstraněn počáteční symbol;
- neterminály na levých stranách pravidel musí zůstat definované (neterminál A z levé strany pravidla lze oddefinovat poté, co byla oddefinována všechna A-pravidla);
- všechny hrany ve stromech pravidel musí odkazovat na definované objekty (objekt, tj. terminál, neterminál, výstupní terminál nebo sémantická akce, je definovaný, je-li jeho název uveden v příslušné sekci, tj. *Terminals*, *Nonterminals*, *OutTerms* nebo *Actions*);

- klíčová slova musí odkazovat na definované terminály;
- symboly musí odkazovat na definované terminály.

Mohlo se ovšem stát, že byl nějaký objekt odstraněn a poté znovu definován – v takovém případě se provede aktualizace odkazu na tento objekt. Zobecnění kontroly konzistence, kterou provádí funkce `LangStructCheck`, viz obrázek 4.1.

```
if ( objekt je používán/požadován )
     if ( objekt byl oddefinován )
          if ( objekt byl definován znovu )
               oprav odkaz na objekt
          else
               chyba
```

Obrázek 4.1: Pseudokód kontroly konzistence redukovaného jazyka

Další kontroly gramatiky se provádějí za běhu analyzátoru (např. máli zpracovávaný neterminál v aktuálním pravidle nějaká pravidla bez levé rekurze). Tyto kontroly by bylo možné dělat pro redukované gramatiky již zde, avšak může být výhodné takové anomálie povolit s tím, že budou pomocí dalších rozšíření opraveny. Ukázka gramatiky, která využívá opravy během rozšíření je v [Žem06, příklad 9.10].

V tuto chvíli se pracovní jazyk nachází víceméně v takovém stavu, jako kdyby byl právě načten z konfiguračního souboru, a tak je možné využít stejné prostředky pro vytvoření dopočítávaných struktur jako u originálního jazyka. Tímto je pracovní jazyk připraven, aby podle něho pokračovala analýza.

Nyní instrukce *permuse*, resp. *locuse*, aktualizuje záznamy v kompilační struktuře – pracovní jazyk prohlásí za aktuální a odkaz na pracovní jazyk zneplatní (null), čímž je připraven na další vnořené změny. Ke skutečně „fyzickému“ přepnutí jazyků, tzn. použití nového jazyka při analýze, dojde ve chvíli, kdy si analyzátor aktualizuje svoji proměnnou s aktuálním jazykem (z kompilační struktury), podle kterého analyzuje. To se děje ihned po návratu ze zpracování sémantické akce. Rovněž je třeba, aby se tak stalo také po návratu ze zpracování neterminálu, neboť i tam mohlo dojít ke změně jazyka.

Závěrečný postup se liší podle toho, jedná-li se o permanentní nebo lokální změnu. V případě lokální změny se kód aktuálního jazyka, není-li to originální jazyk, vloží na zásobník jazyků, aby bylo možné se k němu eventuelně vrátit. V případě permanentní změny se aktuální jazyk, není-li to originální jazyk, uvolní, protože už se k němu nikdy nikdo nevrátí. Opouštěné jazyky se uvolňují při permanentní i lokální změně; rozdíl je v tom, že u permanentní se tak děje při postupu dopředu k dalším jazykům (jiný postup tu ani není), u lokální se uvolnění provádí při postupu zpět k předchozím jazykům (tedy v instrukci *locend*).

Je důležité, aby prvky z původního jazyka měly v novém jazyce stejné kódy. To pak analyzátoru umožňuje, aby po změně aktuálního jazyka pokračoval dál, jako kdyby k žádné změně jazyka nedošlo. Analyzátor pracuje s kódy objektů: pamatuje si kód aktuální hrany ve stromě pravidel, tento kód použije v novém jazyce pro získání záznamu o hraně – při přepnutí jazyků se vyměnily mimo jiné stromy pravidel (záznamy o hranách) – a tam si přečte kód následující hrany. Tím pádem již pracuje podle nového jazyka a nachází se ve stromě pravidel na stejném místě, kde původní jazyk opustil. To samozřejmě znamená, že toto místo musí i v novém jazyce existovat – takový požadavek je triviálně splněn při rozšiřování, redukce je právě proto omezena (nesmí se odstranit aktuální pravidlo nebo rozpracovaná pravidla, která jsou na zásobníku). Odstranění aktuálního nebo nějakého rozpracovaného pravidla pozná analyzátor snadno podle toho, že záznam, který požaduje (např. záznam následující hrany), je označen jako oddefinovaný.

Při lokální změně se lze vrátit k předchozímu jazyku (nebo jazykům, byla-li změna provedena vícekrát) a tím změnu zrušit. K tomu slouží instrukce *locend*. Nejprve aktualizuje záznamy v kompilační struktuře – nový aktuální jazyk je ten z vrcholu zásobníku jazyků, resp. originální jazyk (originální jazyk se na zásobník neukládá), pokud je zásobník jazyků prázdný (tato situace nastane v jednoduchém případě bez vnořených změn). Nakonec uvolní původně aktuální jazyk.

Instrukce *locdef*, *locuse* a *locend* (stejně tak jako *permdef* a *permuse*) musí být správně uzávorkovány, nicméně je možné vynechat *locend* – pak se lokální změna navenek chová jako permanentní, platí až do konce (vnitřně je tu však rozdíl – zachovává se předchozí jazyk).

4.6 Chyby a zotavení

Téměř každá funkce v programu vrací chybový kód a tento je při každém volání funkce testován. Dojde-li tedy někde k chybě, daná funkce vrátí příslušný kód chyby. Volající funkce to zjistí zmíněným testem, sama ukončí svou práci a vrátí také tento kód chyby, případně kód jiné chyby. Další funkce, která ji volala, taktéž testuje návratovou hodnotu atd. – kód chyby se tak propaguje až na nejvyšší úroveň, do funkce `main`.

Ke každé chybě je přiřazeno textové chybové hlášení, které se ve funkci `main` vypíše na standardní výstup. Toto hlášení má však často obecný charakter a nemusí tak být na první pohled jasné, kde přesně k chybě došlo a co ji způsobilo. Proto se během propagace chyby mezi jednotlivými voláními funkcí vypisují další informační hlášení, která upřesňují, proč a kde k chybě došlo. V místě vzniku chyby, resp. na nižších úrovních jejího šíření, je o ní typicky k dispozici více informací než na nejvyšší úrovni (ve funkci `main`).

Pro chyby vzniklé při načítání konfiguračního souboru takovou upřesňující informací může být řádek, který chybu způsobil (viz příklad 4.1).

Příklad 4.1 (Chyba při načítání konfiguračního souboru)**.**

```
LangLearn:                                  (načítání konfiguračního souboru)
jazyk.lng (113): locdef 1                   (chybný řádek)
 ... Error E75: Too many instruction operands.   (chybové hlášení)
```

□

Příklad 4.2 ukazuje chybu zjištěnou za běhu – pokus o oddefinování neterminálu, který neexistuje. Na tuto chybu se přijde při zpracování instrukce *undefnont* a právě informace o tom, že k chybě došlo v této instrukci, ji může pomoci odhalit. Samozřejmostí je oznámení, o který neterminál se jedná.

Příklad 4.2 (Chyba zjištěná za běhu)**.**

```
LangLearn:                                  (načítání konfiguračního souboru)
 ... OK                                     (bez chyby)

Transduce:                                  (analýza vstupního textu)
Nonterminal: A                              (neterminál, který způsobil chybu)
undefnont: A                    (instrukce, kde došlo k chybě, a její operand)
 ... Error E44: Unknown nonterminal name.   (chybové hlášení)
```

□

Zotavení z chyb je implementováno v podobě větvení ve výše popsaných testech volání funkcí nebo přímo v místě vzniku chyby.

V prvním případě se jedná o schéma: zavolej funkci, jestliže došlo k chybě a má-li se dělat zotavení, zotav se (proveď potřebné akce, započítej chybu pro výsledné hlášení a vypiš chybové hlášení pro tuto chybu, případně zavolej funkci znovu). To odpovídá např. zotavení z chyby „neznámý znak ve vstupním textu“, kde onou akcí je přeskočení problémového znaku.

Některé chyby je výhodnější řešit přímo v místě jejich vzniku, protože lze využít k zotavení informace, které na vyšších úrovních nejsou k dispozici. Příkladem může být chyba „dlouhý identifikátor“, kde se použije dosud přečtená část identifikátoru.

4.7 Struktura projektu

Na závěr uvedeme přehled souborů se zdrojovým kódem a jejich stručný popis, který může být užitečný pro toho, kdo by chtěl aplikaci rozšiřovat.

- `actions.c`, `actions.h`: Zpracování sémantických akcí.
- `addstr.c`, `addstr.h`: Pomocné funkce pro práci s řetězci.
- `codestac.c`, `codestac.h`: Pomocné funkce pro práci se zásobníkem (použito při výpočtu výhledů).
- `compil.c`, `compil.h`: Kompilace, spuštění druhé fáze programu.
- `confproc.c`, `confproc.h`: Pomocné funkce pro zpracování konfiguračního souboru.
- `debug.h`: Nastavení ladících tisků a testů.
- `dump.c`, `dump.h`: Funkce pro výpis struktury jazyka.
- `edges.c`, `edges.h`: Práce se stromem hran a s pravidly.
- `errors.c`, `errors.h`: Zpracování chyb (chybové kódy, hlášení, ...).
- `exelib.c`, `exelib.h`: Knihovna funkcí pro externí aplikaci, kterou volá instrukce *exe*.
- `global.h`: Globální definice.

- `extra.c`: Hlavní soubor s funkcí `main`.
- `language.c`, `language.h`: Práce se strukturou jazyka a zpracování konfiguračního souboru.
- `lexer.c`, `lexer.h`: Lexikální analyzátor.
- `nodes.c`, `nodes.h`: Práce s uzly stromu pravidel.
- `nonterm.c`, `nonterm.h`: Zpracování neterminálů.
- `outterm.c`, `outterm.h`: Zpracování výstupních terminálů.
- `parser.c`, `parser.h`: Syntaktický analyzátor.
- `semasm.c`, `semasm.h`: Zpracování sémantických instrukcí.
- `semprg.c`, `semprg.h`: Provádění sémantických instrukcí.
- `terminal.c`, `terminal.h`: Zpracování terminálů, klíčových slov a symbolů.
- `views.c`, `views.h`: Zpracování výhledů.
- `writer.c`, `writer.h`: Pomocné funkce pro zápis do souboru s výstupním textem.

Poznámka 4.2.
Zdrojové kódy byly překládány v operačním systému Microsoft Windows XP pomocí vývojového prostředí Microsoft Visual Studio 2005 Professional Edition. S drobnými úpravami je možné použít i jiné překladače a platformy: testovali jsme Microsoft Windows XP a překladač Open Watcom C32 Optimizing Compiler Version 1.5, Linux a překladač GCC 4.1.1.

Kapitola 5

Příbuzné práce

Tato kapitola přináší přehled dalších prací, které se zabývají podobnou problematikou, tedy zejména rozšiřitelností syntaxe nebo rozšiřitelností za běhu. Její ambicí není být vyčerpávající srovnávací studií, spíše se jedná o příklady zajímavých projektů na dané téma.

Jedním z úkolů plynoucích přímo ze zadání bylo porovnat tuto práci s aplikací KindTran [Žem02b] a diplomovou prací Překladač rozšiřitelného jazyka [Žem94].

KindTran je přímý předchůdce této práce, byl použit jako její základ. Jedná se o translátor (transducer) založený na přívětivých gramatikách. Přestože je připraven na rozšiřitelnost, sám ji ještě plně nepodporuje – to bylo úkolem právě této práce.

S aplikací KindTran úzce souvisí konstruktor přívětivých analyzátorů KindCons [Žem02a] – oba projekty dokonce sdílejí část svých zdrojových kódů, což se projevuje mimo jiné tím, že používají stejný konfigurační soubor. A tak, pokud se v konfiguračním souboru pro KindTran (resp. ExTra) používají konstrukty, které podporuje i KindCons, lze s jeho pomocí vygenerovat zdrojové a hlavičkové soubory v jazyce C, které implementují analyzátor pro danou gramatiku. Jeho velkou výhodou je, že tyto zdrojové kódy vypadají jako ručně napsané, jsou tedy dobře čitelné a přehledné.

Diplomová práce [Žem94] implementuje za běhu rozšiřitelný překladač pro LL(1) jazyky, jeho síla odpovídá 1-přívětivému analyzátoru. Co se týče rozšiřitelnosti, nabízí možnost permanentního rozšiřování syntaxe jazyka za běhu obdobně jako tato práce, ne však lokální rozšíření nebo redukci. Také podporuje sémantické akce, ale tyto mohou být pouze na konci pravidel. Dalším omezením je velikost výhledu, a to na 1.

XML

V současné době jsou nejčastěji používanými za běhu rozšiřitelnými analyzátory ty, které pracují s jazykem XML [W3C]. Dokument v jazyce XML se skládá ze značek (tagů) a musí splňovat daná syntaktická pravidla (např. počáteční a koncové tagy musí být správně uzávorkovány). Uživatel si může stanovit další pravidla, která má daný dokument splňovat, k tomu slouží tzv. XML schéma – to může být přímo součástí dokumentu. Pomocí transformačního jazyka XSLT je možné XML dokumenty převádět na dokumenty v jiném formátu (např. HTML, prostý text, ...).

Lua

Svou sémantiku dovoluje rozšiřovat skriptovací jazyk Lua [IdFC06], který našel své uplatnění mimo jiné při programování her. Využívá mechanismus metatabulek (metatables) a metametod (metamethods). Metatabulka je asociativní pole, které váže operace s metametodami, a tyto mohou být měněny, čímž lze ovlivnit chování operací.

Delegating Compiler Object

Rozdělit kompilátor na spolupracující objekty (Delegating Compiler Object, DCO) navrhuje [Bos97]. Každý DCO obsahuje jeden nebo více lexikálních analyzátorů a syntaktických analyzátorů, jednotlivé DCO spolupracují při analýze vstupního textu. Konstrukty přijímaného jazyka jsou rozděleny do skupin, každá skupina je pak překládána odpovídajícím DCO. Překladač může být rozšířen o nové jazykové konstrukty přidáním nových DCO, které tyto konstrukty implementují.

Earley

[Kol02] využívá pro implementaci rozšiřitelného analyzátoru algoritmus Earley. Stavy analyzátoru reprezentují množinu rozpoznaných pravidel. Na rozdíl od analyzátoru, který je vystavěn pomocí nějakého generátoru a má předpočítané možné množiny a přechody mezi nimi pro všechny platné vstupy, tato technika vytváří takovou množinu za běhu – této vlastnosti je využito pro rozšiřitelnost.

Seed

Jiným příkladem rozšiřitelného jazyka je Seed7 [Mer]. Jádro jazyka podporuje jen několik primitivních akcí, na jejichž základě lze definovat typy, syntaxi operátorů a příkazů, jejich sémantiku. Standardní část jazyka je pak definována pomocí těchto akcí, každý zdrojový text je vkládá jako standardní knihovnu. Stejným způsobem pak může uživatel definovat své vlastní příkazy a operátory. Teoreticky je možné vložením vlastní knihovny definovat libovolný jazyk, i když autor přiznává, že primitivní akce jsou šity na míru jazyku Seed7.

Kapitola 6

Závěr

Cílem této práce bylo zabývat se možností rozšíření syntaxe jazyka a vytvořit za běhu rozšiřitelný analyzátor, který podporuje permanentní i lokální rozšíření přijímaného jazyka (gramatiky), přičemž pokyny pro tato rozšíření mohou být součástí analyzovaného textu. Analyzátor měl být založen na přívětivých gramatikách. Tohoto cíle bylo dosaženo vytvořením aplikace ExTra, která tyto požadavky splňuje.

Podařilo se vyvinout aplikaci, která je schopna za běhu změnit syntaxi přijímaného jazyka – přidat nové konstrukty nebo naopak odebrat existující konstrukty – pomocí přidávání, resp. odebírání, pravidel gramatiky. Díky tomu, že pravidla jsou reprezentována pomocí stromů, není třeba vytvářet celou strukturu znovu – stačí pouze přidat další větev, která odpovídá přidanému pravidlu, nebo selektivně „vyříznout“ větev, která odpovídá odebranému pravidlu. Analýza může okamžitě po změně pokračovat bez nutnosti restartu.

Součástí pravidel jsou také sémantické akce, které představují sémantickou analýzu, takže spolu se změnou syntaxe je možné za běhu ovlivňovat také sémantiku. Vzhledem k tomu, že je dovoleno měnit tvary klíčových slov a symbolů, zasahuje rozšíření a redukce rovněž lexikální analyzátor. Pomocí změn výstupních terminálů je možné ovládat výstupní jazyk. To vše lze samozřejmě udělat buď s permanentní platností, nebo s lokální platností.

Podpora lokální platnosti rozšíření nebo redukce jazyka přináší nové možnosti využití rozšiřitelnosti – dočasné přidání nebo potlačení některých konstruktů, střídání různých dialektů a v konečném důsledku i celých jazyků. Nabízí nový způsob řešení situací, které nejsou bezkontextové a mají lokální charakter.

Díky tomu, že ExTra je založen na přívětivých gramatikách, lze v definici jazyka používat pravidla s přímou levou rekurzí (taková obsahuje např. známá gramatika aritmetického výrazu, viz příklad 2.1) nebo pravidla se stejným prefixem pravých stran. Sémantické akce je možné umisťovat na téměř libovolná místa v pravidlech.

Aplikace byla nad rámec zadání obohacena o další vlastnosti, které mohou být užitečné při jejím používání – např. možnost spravovat bílé znaky ve vstupním textu, možnost použít ve vstupním textu znaky s diakritikou, rozšíření souboru sémantických instrukcí, rozšíření možností operandů sémantických instrukcí, základní zotavení z chyb a další.

6.1 Možnosti dalšího vývoje

Další vývoj by mohl směřovat k dopracování rozšiřitelného translátoru na rozšiřitelný překladač. To by znamenalo rozvinout zejména sémantickou analýzu a doplnit generátor mezikódu, případně přidat back-end.

Některé části aplikace byly udělány na míru současné verzi, při dalším rozšiřování by bylo třeba je doplnit, resp. nahradit jiným mechanismem. Např. zotavení z chyb by mohlo být řešeno pomocí skeletálních množin ([Plá98]) nebo polohlaviček ([Plá92]).

Dále by bylo možné implementovat dokonalejší rozšiřitelný lexikální analyzátor podle myšlenek, které byly představeny v práci [Jan03]. Tento lexikální analyzátor využívá k definování tvarů terminálů regulární výrazy, ke každému terminálu tak může náležet celý regulární jazyk. Současně nabízí možnost za běhu přidávat nebo odebírat terminály a jejich tvar.

Literatura

[AU72] Alfred V. Aho a Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling, Volume I: Parsing.* Prentice-Hall, Englewood Cliffs, N. J., 1972.

[Bos97] Jan Bosch. Delegating compiler objects: Modularity and reusability in language engineering. *Nordic Journal of Computing*, 4(1):66–92, 1997.

[Chy84] Michal Chytil. *Automaty a gramatiky.* SNTL, Praha, 1984.

[IdFC06] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, a Waldemar Celes. *Lua 5.1 Reference Manual.* Lua.org, 2006.

[Jan03] Jan Janeček. Rozšiřitelný lexikální analyzátor. Diplomová práce, Matematicko-fyzikální fakulta, Univerzita Karlova, Praha, 2003.

[Kol02] Donovan Michael Kolbly. *Extensible Language Implementation.* Dizertační práce, The University of Texas at Austin, 2002.

[KŽ05] Jaroslav Král a Michal Žemlička. Architecture, specification, and design of service-oriented systems. In Zoran Stojanovic a Ajantha Dahanayake, editoři, *Service-Oriented Software System Engineering: Challenges and Practices*, s. 182–200, Hershey, PA, USA, 2005. Idea Group Publishing.

[Mer] Thomas Mertes. Seed7. http://seed7.sourceforge.net.

[Plá92] Martin Plátek. Syntactic error recovery with formal guarantees I. Technická zpráva, Matematicko-fyzikální fakulta, Univerzita Karlova, Praha, duben 1992.

[Plá98] Martin Plátek. Testovatelné podmínky pro bezpečnou skeletální množinu a jejich zobecnění. In *Sborník krátkých referátů SOFSEM'88*, s. 43–46, Bratislava, 1998. ÚVT UJEP a JČMF.

[W3C] W3C. Extensible Markup Language. http://www.w3.org/XML/.

[Žem94] Michal Žemlička. Překladač rozšiřitelného jazyka. Diplomová práce, Matematicko-fyzikální fakulta, Univerzita Karlova, Praha, 1994.

[Žem96] Michal Žemlička. Syntaktická analýza rozšiřitelných jazyků. Technická zpráva, Katedra softwarového inženýrství, Matematicko-fyzikální fakulta, Univerzita Karlova, Praha, 1996.

[Žem02a] Michal Žemlička. KindCons – Kind Constructor. http://www.ms.mff.cuni.cz/~zemlicka/KindCons/, 2002.

[Žem02b] Michal Žemlička. KindTran – Kind Transducer. http://www.ms.mff.cuni.cz/~zemlicka/KindTran/, 2002.

[Žem02c] Michal Žemlička. Principles of Kind Parsing – An Introduction. Technická zpráva, Katedra softwarového inženýrství, Matematicko-fyzikální fakulta, Univerzita Karlova, Praha, 2002.

[Žem06] Michal Žemlička. *Principles of Kind Parsing.* Dizertační práce, Matematicko-fyzikální fakulta, Univerzita Karlova, Praha, 2006.

Seznam obrázků

Seznam tabulek

Dodatek A

Rozsah práce

Tato práce přímo navazovala na projekt KindTran [Žem02b], a tak zde přesněji uvedeme, co bylo doplněno, přidáno a změněno. Detailnější popis, až na úrovni jednotlivých funkcí, lze nalézt přímo ve zdrojových kódech ExTra. Celkem bylo doplněno více jak osm tisíc řádků kódu.

- Permanentní a lokální rozšíření a redukce, s tím související změny mnoha dalších částí;
- rozšíření souboru sémantických instrukcí:
  - doplnění celočíselných aritmetických operací,
  - reálné aritmetické operace,
  - konverze,
  - bitové operace,
  - řízení rozšíření a redukce,
  - rozšíření,
  - redukce,
  - změny nastavení,
  - skoky,
  - externí aplikace,
  - speciální instrukce;
- rozšíření počtu operandů sémantických instrukcí na tři;

- rozšíření možností použití přímého operandu;
- nový příznak sémantických instrukcí: *acc*;
- správa bílých znaků;
- možnost používat znaky s diakritikou (locale);
- správa nového typu: reálné číslo;
- základní zotavení z chyb;
- zdokonalení správy chyb;
- zdokonalení správy varování;
- možnost zadávání tvaru symbolů a výstupních terminálů pomocí ASCII kódů;
- nové příznaky sekce *Language*;
- nové speciální terminály;
- doplnění výpisu struktury jazyka (dump);
- testy:
  - splnění přívětivé podmínky o disjunkci výhledů,
  - správné použití operandů sémantických instrukcí,
  - příliš dlouhé hodnoty ve vstupním textu,
  - příliš dlouhé hodnoty v konfiguračním souboru,
  - neterminál bez pravidel v místě větvení,
  - dva sémantické programy pro stejnou akci,
  - žádný sémantický program pro akci,
  - přítomnost instrukce *return* v sémantickém programu,
  - a jiné;
- zdokonalení načítání konfiguračního souboru;
- označení mrtvého kódu;
- okomentování kódu;
- ...

Dodatek B

Obsah CD

K práci je přiloženo CD, které obsahuje implementaci ExTra a příklady konfiguračních souborů. Následuje podrobný obsah:

- Adresář `src` – zdrojové kódy ExTra v jazyce C.
- Adresář `dist` – spustitelná verze ExTra pro platformu Win32 (debug a release verze, testována byla zejména debug verze).
- Adresář `text` – elektronická verze textu této práce.
- Adresář `examples` – příklady konfiguračních souborů (včetně souborů se vstupním textem a případně očekávaných souborů s výstupním textem):
  - adresář `ariexp1` – jednoduchý aritmetický výraz, v souboru s výstupním textem je jeho výsledek, elektronická verze příkladu 3.7;
  - adresář `ariexp2` – seznam aritmetických výrazů s ukázkou použití rozšíření a redukce, mezi klíčovými slovy `begin` a `end` lze odčítat, ale nelze sčítat, elektronická verze příkladu 3.8;
  - adresář `extpas` – podmnožina jazyka Pascal s rozšiřitelnou syntaxí v době překladu, elektronická verze dodatku C;
  - adresář `extpaspp` – přidává k příkladu *extpas* výstupní jazyk, v souboru s výstupním textem je vstupní zdrojový kód ve formátu html (pretty printer);

– adresář `2lang` – kombinace dvou programovacích jazyků v jednom souboru (podmnožina jazyka C a Pascal), pro přepínání mezi nimi slouží klíčová slova `clike`, `paslike` a `finish`, pravidla definují jen jádro sloužící k přepínání a vlastní jazyky jsou implementovány pomocí rozšíření a redukce;
– adresář `wiki2tex` – převod textu ve formátu wiki (podmnožina) do formátu LaTeX;
– adresář `wiki2xml` – převod textu ve formátu wiki (podmnožina) do formátu XML;
– adresář `text2xml` – lokálně rozšiřitelný převod neformátovaného textu (tím se rozumí posloupnost odstavců, které jsou odděleny prázdnými řádky) do formátu XML, každý odstavec je implicitně uzavřen mezi `<tag></tag>`, jedná se o rozšiřitelnost výstupního jazyka pomocí definic (definice má tvar `<znacka>`, nová značka platí až do výskytu `<>`), které umožňují implicitní značku změnit;
– adresář `exelib` – ukázka použití instrukce *exe* a knihovny *exelib*, v souboru se vstupním textem se očekává řetězec, v souboru s výstupním textem je pak jeho délka, na její výpočet byl volán program `strlen.exe`.

Dodatek C

Příklad rozšiřitelného jazyka

Příklad C.1 ukazuje již složitější konfigurační soubor definující gramatiku jazyka, který obsahuje příkazy pro rozšíření své syntaxe za běhu. Jedná se o podmnožinu jazyka Pascal (byla použita gramatika jazyka Pascal ve formátu KindTran, viz [Žem02b]). Na všech místech, kde lze použít běžné příkazy, se mohou navíc objevit `langdef` (*locdef*), `languse` (*locuse*), `langend` (*locend*), `terminal` (*defterm*), `nonterminal` (*defnont*), `outterminal` (*defoutterm*), `action` (*defact*), `production` nebo `rule` (*defprod*), `keyword` (*defkwd*), `symbol` (*defsymb*), `outshape` (*defoutshp*), `instruction` (*defins*) a `termkwd` (*deftermkwd*), které vyvolají v závorkách uvedené sémantické instrukce.

Příklad C.1 (Konfigurační soubor rozšiřitelného jazyka)**.**
Příklad ilustruje čistě možnost rozšíření analyzátoru za běhu; pokyny pro tato rozšíření, která mají lokální charakter, jsou součástí analyzovaného textu. Nedefinuje žádný výstupní jazyk (pomocí výstupních terminálů), výstupem je tedy pouze informace, zda soubor se vstupním textem odpovídá gramatice vstupního jazyka.

```
[Language]
  Name = Extensible language
  LocExtensible = Yes
  Extended = Yes
  Lookahead = 1
  StartingSymbol = S
  IdName = id
  IntName = number
```

```
  StrName = string
  EofName = eof
  StrDelim = "
  CommentStart1 = {
  CommentEnd1 = }
  LangStackSize = 10
  SemStackSize = 100
  Terminals = 100
  Nonterminals = 100
  Actions = 100
  Productions = 100
  Keywords = 100
  SymbNodes = 100
  SemInsts = 100
  Edges = 1000
  Nodes = 1000
  Views = 1000


[Terminals]
  eof
  id
  number
  string
  program
  var
  forward
  function
  procedure
  begin
  end
  if
  then
  else
  while
  do
  and
  or
```

```
xor
not
div
mod
leftbrace
rightbrace
leftparenthesis
rightparenthesis
comma
semicolon
equal
lt
gt
le
ge
ne
plus
minus
slash
star
dot
colon
assign
locdef
locuse
locend
defterm
defnont
defoutterm
defact
defprod
defkwd
defsymb
defoutshp
defins
deftermkwd
```

```
[Nonterminals]
  S
  ID
  ID_LIST
  STR
  NUM
  DECL_LIST
  DECL
  BLOCK
  VARDECL
  FUNCDECL
  PROCDECL
  VARDEC_LIST
  VARDEC
  TYPESPEC
  FUNCHEADER
  PROCHEADER
  FORMPARS
  FORMPAR_LIST
  FORMPAR
  STATEMENT_LIST
  STATEMENT
  L_VAL
  TRUEPARAMS
  TRUEPAR_LIST
  TRUEPAR
  COND
  EXPR
  EXPR_LIST
  TERM
  FACTOR
  R_VAL


[Actions]
  getval
  delval
  locdef
```

```
  locuse
  locend
  defterm
  defnont
  defoutterm
  defact
  defprod
  defkwd
  defsymb
  defoutshp
  defins
  deftermkwd


[Productions]
  S::=(program)[ID](semicolon)[BLOCK](dot)
  BLOCK::=(begin)[STATEMENT_LIST](end)
  BLOCK::=[DECL_LIST](begin)[STATEMENT_LIST](end)
  DECL_LIST::=[DECL_LIST][DECL]
  DECL_LIST::=[DECL]
  DECL::=[FUNCDECL]
  DECL::=[PROCDECL]
  DECL::=[VARDECL]
  VARDECL::=(var)[VARDEC_LIST]
  VARDEC_LIST::=[VARDEC]
  VARDEC_LIST::=[VARDEC_LIST][VARDEC]
  VARDEC::=[ID_LIST](colon)[TYPESPEC](semicolon)
  TYPESPEC::=[ID]
  ID_LIST::=[ID]
  ID_LIST::=[ID_LIST](comma)[ID]
  FUNCDECL::=[FUNCHEADER](forward)(semicolon)
  FUNCDECL::=[FUNCHEADER][BLOCK](semicolon)
  FUNCHEADER::=(function)[ID](colon)[TYPESPEC](semicolon)
  FUNCHEADER::=(function)[ID][FORMPARS](colon)[TYPESPEC](semicolon)
  PROCDECL::=[PROCHEADER](forward)(semicolon)
  PROCDECL::=[PROCHEADER][BLOCK](semicolon)
  PROCHEADER::=(procedure)[ID](semicolon)
  PROCHEADER::=(procedure)[ID][FORMPARS](semicolon)
```

```
FORMPARS::=(leftparenthesis)[FORMPAR_LIST](rightparenthesis)
FORMPAR_LIST::=[FORMPAR]
FORMPAR_LIST::=[FORMPAR_LIST](semicolon)[FORMPAR]
FORMPAR::=[ID_LIST](colon)[TYPESPEC]
FORMPAR::=(var)[ID_LIST](colon)[TYPESPEC]
STATEMENT_LIST::=[STATEMENT]
STATEMENT_LIST::=[STATEMENT_LIST](semicolon)[STATEMENT]
STATEMENT::=
STATEMENT::=(begin)[STATEMENT_LIST](end)
STATEMENT::=[L_VAL]
STATEMENT::=[L_VAL][TRUEPARAMS]
STATEMENT::=[L_VAL](assign)[COND]
STATEMENT::=(while)[COND](do)[STATEMENT]
STATEMENT::=(if)[COND](then)[STATEMENT](else)[STATEMENT]
STATEMENT::=(if)[COND](then)[STATEMENT]
STATEMENT::=(locdef){locdef}
STATEMENT::=(locuse){locuse}
STATEMENT::=(locend){locend}
STATEMENT::=(defterm)(string){getval}{defterm}
STATEMENT::=(defnont)(string){getval}{defnont}
STATEMENT::=(defoutterm)(string){getval}{defoutterm}
STATEMENT::=(defact)(string){getval}{defact}
STATEMENT::=(defprod)(string){getval}{defprod}
STATEMENT::=(defkwd)(string){getval}(string){getval}{defkwd}
STATEMENT::=(defsymb)(string){getval}(string){getval}{defsymb}
STATEMENT::=(defoutshp)(string){getval}(string){getval}{defoutshp}
STATEMENT::=(defins)(string){getval}{defins}
STATEMENT::=(deftermkwd)(string){getval}{deftermkwd}
L_VAL::=[ID]
EXPR_LIST::=[EXPR]
EXPR_LIST::=[EXPR_LIST](comma)[EXPR]
TRUEPARAMS::=(leftparenthesis)[TRUEPAR_LIST](rightparenthesis)
TRUEPAR_LIST::=[TRUEPAR]
TRUEPAR_LIST::=[TRUEPAR_LIST](comma)[TRUEPAR]
TRUEPAR::=[COND]
COND::=[EXPR]
COND::=[EXPR](equal)[EXPR]
COND::=[EXPR](lt)[EXPR]
```

```
  COND::=[EXPR](gt)[EXPR]
  COND::=[EXPR](le)[EXPR]
  COND::=[EXPR](ge)[EXPR]
  COND::=[EXPR](ne)[EXPR]
  EXPR::=[TERM]
  EXPR::=(minus)[TERM]
  EXPR::=(plus)[TERM]
  EXPR::=[EXPR](plus)[TERM]
  EXPR::=[EXPR](minus)[TERM]
  EXPR::=[EXPR](or)[TERM]
  EXPR::=[EXPR](xor)[TERM]
  TERM::=[FACTOR]
  TERM::=[TERM](star)[FACTOR]
  TERM::=[TERM](slash)[FACTOR]
  TERM::=[TERM](div)[FACTOR]
  TERM::=[TERM](mod)[FACTOR]
  TERM::=[TERM](and)[FACTOR]
  FACTOR::=(not)[FACTOR]
  FACTOR::=[R_VAL]
  FACTOR::=(leftparenthesis)[COND](rightparenthesis)
  R_VAL::=[ID]
  R_VAL::=[ID][TRUEPARAMS]
  R_VAL::=[NUM]
  R_VAL::=[STR]
  ID::=(id){delval}
  NUM::=(number){delval}
  STR::=(string){delval}


[Keywords]
  program = program
  var = var
  function = function
  procedure = procedure
  forward = forward
  begin = begin
  end = end
  if = if
```

```
  then = then
  else = else
  while = while
  do = do
  and = and
  or = or
  xor = xor
  not = not
  div = div
  mod = mod
  locdef = langdef
  locuse = languse
  locend = langend
  defterm = terminal
  defnont = nonterminal
  defoutterm = outterminal
  defact = action
  defprod = production
  defprod = rule
  defkwd = keyword
  defsymb = symbol
  defoutshp = outshape
  defins = instruction
  deftermkwd = termkwd


[Symbols]
  leftbrace = {
  rightbrace = }
  leftparenthesis = (
  rightparenthesis = )
  comma = ,
  semicolon = ;
  equal = =
  lt = <
  gt = >
  le = <=
  ge = >=
```

```
  ne = <>
  plus = +
  minus = -
  slash = /
  star = *
  dot = .
  colon = :
  assign = :=


[Instructions]
  #getval
  fetch r0
  pushm r0
  return

  #delval
  fetch r0
  del r0
  return

  #locdef
  locdef
  return

  #locuse
  locuse
  return

  #locend
  locend
  return

  #defterm
  popm r0
  defterm r0
  del r0
  return
```

```
#defnont
popm r0
defnont r0
del r0
return

#defoutterm
popm r0
defoutterm r0
del r0
return

#defact
popm r0
defact r0
del r0
return

#defprod
popm r0
defprod r0
del r0
return

#defkwd
popm r0
popm r1
defkwd r1, r0
del r0
del r1
return

#defsymb
popm r0
popm r1
defsymb r1, r0
del r0
```

```
  del r1
  return

  #defoutshp
  popm r0
  popm r1
  defoutshp r1, r0
  del r0
  del r1
  return

  #defins
  popm r0
  defins r0
  del r0
  return

  #deftermkwd
  popm r0
  deftermkwd r0
  del r0
  return
```

□

V příkladu C.2 si ukážeme, jak lze přímo v souboru se vstupním textem definovat nový syntaktický konstrukt: cyklus s podmínkou uprostřed.

Příklad C.2 (Soubor se vstupním textem)**.**
Cyklus s podmínkou uprostřed definovaný v souboru se vstupním textem.

```
{Vypocet prumeru rady celych cisel, ktera konci 0}
program Average;

var
  sum, num, count : integer;
  avg : real;

begin
```

```
  sum:=0;
  count:=0;

  {rozsireni jazyka o novy cyklus}
  langdef;
  termkwd "loop";
  termkwd "on";
  termkwd "exit";
  production "STATEMENT::=(loop)[STATEMENT_LIST](on)[COND](exit)
              [STATEMENT_LIST](end)";
  languse;

    loop
      read(num);

      on num=0 exit

      sum:=sum+num;
      count:=count+1;
    end;

  langend;

  if count<>0 then
    avg:=sum/count
  else
    avg:=0;

  writeln(avg);

end.
```

□