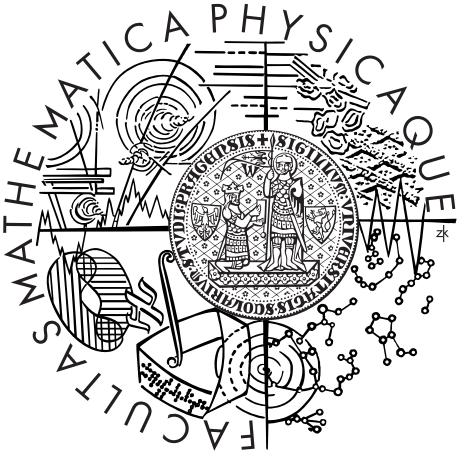
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Karel Fišer

Moderní implementace LALR(1) konstruktoru

Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. David Bednárek, Ph.D.

Studijní program: Informatika
Studijní obor: Softwarové systémy

Praha 2013

Na tomto místě bych rád poděkoval všem, kteří mě podporovali při práci na projektu, především svému vedoucímu RNDr. Davidu Bednárkovi, Ph.D., který mi dal velmi cenné rady a odborně mě vedl celou prací. Dále bych rád poděkoval panu RNDr. Michalu Žemličkovi, Ph.D., který mi ochotně půjčil knihu Parsing techniques [9] k prostudování.

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 24. července 2013 Karel Fišer

Název práce: Moderní implementace LALR(1) konstruktoru

Autor: Karel Fišer

Katedra: Katedra softwarového inženýrství

Vedoucí práce: RNDr. David Bednárek, Ph.D.

Abstrakt: Cílem této práce je navrhnout moderní design konstruktoru parserů a návrh poté realizovat. Výsledkem práce je programátorské dílo sestávající z programu, který ze vstupního souboru čte popis bezkontextové LALR(1) gramatiky a sémantických akcí. Do výstupních souborů generuje zdrojový kód syntaktického analyzátoru, který při parsování jazyka odpovídajícího dané gramatice vykonává dané sémantické akce. Součástí jsou šablony zdrojových kódů pro implementaci výsledného analyzátoru, a to pro několik cílových moderních objektových programovacích jazyků.

Klíčová slova: syntaktická analýza, LALR(1), konstruktor parserů, bison

Title: A modern implementation of LALR(1) parser generator

Author: Karel Fišer

Department: Department of Software Engineering

Supervisor: RNDr. David Bednárek, Ph.D.

Abstract: The goal of this thesis is to design and implement a modern parser generator. The result is a program that reads description of some context-free LALR(1) grammar and semantic actions from an input file. To output files the program generates source codes in several target modern object-oriented programming languages for implementation of the syntax analyzer which, when parsing the language corresponding to the given grammar, executes the given semantic actions.

Keywords: syntax analysis, LALR(1), parser generator, bison

Obsah

1. Úvod

Formální gramatiky a možnost formálně specifikovat jazyk, tedy exaktně definovat jeho strukturu, přináší také možnost daný jazyk efektivně automaticky zpracovávat. Je známo z teorie automatů, že regulární jazyky lze rozpoznávat konečnými automaty a jazyk popsaný bezkontextovou gramatikou lze rozpoznávat nedeterministickým zásobníkovým automatem. Na bezkontextové jazyky je tato práce zaměřena, protože jsou dostatečně jednoduché na specifikaci i lidské porozumění a dostatečně silné pro vyjádření např. programovacího jazyka.

Bezkontextová gramatika je formálně definována jako čtveřice (N, T, S, P), kde N je množina neterminálních symbolů, T množina terminálních symbolů, $S \in N$ je startovní neterminál a P množina pravidel gramatiky. Pravidla jsou pro bezkontextovou gramatiku ve tvaru $A \to w$, kde $A \in N$ a $w \in (N \cup T)^*$.

Důležitou podtřídou bezkontextových gramatik jsou gramatiky, které lze parsovat deterministickým zásobníkovým automatem. Donald Knuth tyto gramatiky pojmenoval jako `překladatelné zleva doprava` a označil je LR [10]. DeRemer tyto gramatiky dále studoval a zavedl jejich podtřídy SLR a LALR [6]. Všechny tyto LR gramatiky se vyznačují tím, že je lze parsovat zleva doprava metodou zdola nahoru (více v následující kapitole). Třída gramatik, kterou lze parsovat deterministickým zásobníkovým automatem metodou shora dolů se nazývá LL.

Konstruktor parserů je program, který ze zadané specifikace jazyka vygeneruje jiný program, který dokáže specifikovaný jazyk (přesněji data odpovídající svou strukturou zadanému jazyku) rozpoznávat a zpracovávat.

Analýza vstupních dat se typicky neprovádí přímo po jednotlivých bytech či znacích, ale po skupinách znaků, které tvoří tzv. tokeny. Část parseru, která se věnuje rozpoznávání tokenů ve vstupních datech, se nazývá lexer, lexikální analyzátor nebo tokenizer. Lexerem se tato práce detailněji nezabývá, součástí vygenerovaného parseru je pouze rozhraní, přes které je externí lexer volán.

Výstupem lexikální analýzy je proud tokenů, které tvoří elementární jednotky z pohledu syntaxe. Tokeny odpovídají terminálům bezkontextové gramatiky. Syntaktický analyzátor, tedy část parseru, která je pro tuto práci nejzajímavější, zpracovává tokeny produkované lexerem a kontroluje, zda je jejich pořadí přípustné, zda odpovídají specifikaci daného jazyka.

Syntaktický analyzátor tedy dokáže rozhodnout, zda daná posloupnost tokenů odpovídá slovu z daného jazyka, zda do jazyka patří, či nikoliv. Je ale třeba také

části vstupu dále zpracovávat, případně překládat do jiného jazyka. K tomu slouží sémantická analýza, která pracuje s jednotlivými konstrukcemi v daném jazyce, a tyto zpracovává podle logiky výsledné úlohy. Sémantická analýza může být součástí kódu výsledného parseru, ale je tvořena především uživatelským kódem. Musí totiž implementovat zpracování jazyka podle potřeb výsledné aplikace.

Vstupem konstruktoru je tedy bezkontextová gramatika, tvořená mimo jiné množinou pravidel. Každé pravidlo může mít zadanou sémantickou akci, což je uživatelský kód, který se provede vždy, když je dané pravidlo použito. Z pravidel je vygenerována implementace zásobníkového automatu, který popsaný jazyk zpracovává a spouští sémantické akce.

Cílem práce je vyvinout takový nástroj, aby bylo jeho použití podobné velmi rozšířenému LALR(1) konstruktoru Bison (více o tomto konstruktoru v kapitole 2.2). Gramatika bude tedy uložena v souboru, jehož název dostane konstruktor jako parametr příkazové řádky, bude moci obsahovat různá nastavení konstruktoru, gramatická pravidla včetně sémantických akcí a případně i další uživatelský kód. Bude však generovat více objektově orientovaný parser s přehlednějším a stabilním rozhraním. V projektu nemusí být implementováno zotavení ze syntaktických chyb, ale měl by řešit případné výjimky, ke kterým může dojít v rámci sémantických akcí. Uživatel by si tak mohl vybrat, jestli chce nechat výjimky parserem korektně projít, nebo jestli je má parser odchytávat a pokračovat dál.

Tento nástroj by neměl být zaměřen na žádný konkrétní cílový programovací jazyk, měl by být v tomto smyslu dobře rozšiřitelný a v rámci této práce již podporovat několik moderních programovacích jazyků.

2. Související práce

Generováním syntaktických parserů se zabývá mnoho projektů. Používají se k sestavení přední části překladače, implementaci kalkulátorů, nebo při interpretaci nekompilovaných programů. Různé projekty implementují generátor pro různé podmnožiny bezkontextových jazyků, v závislosti na tom, jaké metody se pro parsování použijí.

2.1 Techniky parsování

Podle [9, s. 65], existují pouze dvě různé techniky syntaktické analýzy, veškeré ostatní odlišnosti jsou jen technické detaily.

Jedna metoda se snaží napodobit původní proces generování gramatikou a znovu odvodit slovo z počátečního symbolu. Tato metoda je nazývána shora dolů (top-down), protože syntaktický strom je rekonstruován od kořene dolů.[1]

Druhá metoda pracuje obráceně zespoda nahoru (bottom-up) a snaží se zredukovat terminály přes neterminály až k počátečnímu symbolu. Syntaktický strom je konstruován od listů ke kořeni.

Většina parserů využívá výhled (lookahead) velikosti `k`, což je znalost následujících `k` tokenů od místa, které je právě analyzováno. Má-li parser pracovat deterministicky, je nutné, aby se v každém místě vstupu správně rozhodl, které pravidlo má být použito. To není někdy jednoznačné a parser musí zjistit, jaké tokeny následují, aby se mohl rozhodnout. Pokud není schopen pro danou gramatiku rozhodnout ani se znalostí následujících `k` tokenů, řekneme, že gramatika je pro daný typ parseru nejednoznačná nebo že obsahuje konflikty. Je známo, že žádný omezený výhled nepostačuje všem bezkontextovým gramatikám [9, s. 254]. Pokud parser s omezeným výhledem nestačí, může být použit zobecněný deterministický, který může najednou prověřovat více možností, jak vstup zpracovat.

2.1.1 LL parsery

LL parser provádí syntaktickou analýzu shora dolů. Analyzuje vstup zleva doprava a konstruuje nejlevější derivaci.

Základní algoritmus LL(1), tedy s výhledem na jeden následující token, je velmi silný a intuitivní. Může být implementován ručně jako sada vzájemně rekurzivních funkcí, pro každý neterminál jedna. Pokud gramatika neobsahuje prázdná pravidla, je

[1] Stromy v informatice rostou od svých kořenů směrem shora dolů.

to velmi jednoduché. Prázdná pravidla situaci komplikují a je třeba zjistit o gramatice více. Síla deterministického LL parsování může být ještě zvětšena rozšířením výhledu. Pokud je výhled omezený `k` tokeny, mluvíme o LL(k) parseru [9, s. 260].

Generátory LL parserů se musí vypořádat s levou rekurzí, která dělá LL parseru problémy, protože vytváří u levě rekurzivních pravidel konflikty.

2.1.2 LR parsery

LR parser provádí syntaktickou analýzu zdola nahoru. Analyzuje vstup zleva doprava a konstruuje nejpravější derivaci [10].

LR(0)

LR(0) parser se snaží bez jakéhokoli výhledu procházet vstup. Kterým pravidlům by mohl doposud přečtený vstup odpovídat, si pamatuje ve stavech, které si ukládá na zásobník. Pokud je na vrcholu zásobníku stav odpovídající přečtené celé pravé straně některého pravidla, vršek zásobníku je zredukován a nahrazen stavem, který reprezentuje rozečtený vstup zakončený neterminálem z levé strany právě zredukovaného pravidla. Když je vstup dočtený do konce a na zásobníku je pouze počáteční symbol gramatiky, vstup odpovídá dané gramatice.

Pro popis konstrukce automatu je jednodušší, když se startovní neterminál objevuje pouze na levé straně jediného pravidla. Z tohoto důvodu se vždy použije rozšířená gramatika, která má navíc nový startovní neterminál a pravidlo, kde je tento nový neterminál na levé straně a na pravé straně pouze původní startovní neterminál. Taková gramatika generuje stejný jazyk jako původní, obsahuje ale pouze jedno pravidlo se startovním neterminálem, které nazveme startovní pravidlo.

Pro výrobu stavů LR(0) automatu je použita metoda otečkovaných pravidel. Nový symbol tečky, který není ani terminál ani neterminál, je vkládán na pravou stranu pravidla a symbolizuje místo, kde je čtecí hlava automatu. Dvě otečkovaná pravidla jsou shodná, pokud reprezentují stejné (neotečkované) pravidlo gramatiky a mají tečku na stejném místě.

Uzávěr množiny otečkovaných pravidel je operace, která tranzitivně přidá do množiny všechna otečkovaná pravidla s tečkou na začátku a neterminálem `A` na levé straně, pokud se v množině vyskytuje otečkované pravidlo s tečkou před neterminálem `A`.

Přechod ze stavu `S` podle symbolu `X` je uzávěr množiny obsahující pravidla $A : \alpha X \cdot \beta$ taková, že množina `S` obsahovala pravidla $A : \alpha \cdot X\beta$.

Algoritmus konstrukce LR(0) automatu začíná s počátečním stavem, který je tranzitivním uzávěrem množiny, ve které je startovní pravidlo. Další stavy jsou vytvořeny jako přechody z již existujících stavů.

Ve skutečnosti je LR(0) algoritmus použitelný jen pro velmi jednoduché jazyky. Složitější jazyky vytváří bez výhledu konflikty.

LR(1)

Algoritmus LR(1) je velmi podobný. Má navíc výhled na jeden následující token, což řeší nejednoznačnosti u podstatné skupiny gramatik.

Třída LR(1) jazyků je navíc vlastní nadmnožinou LL(1) jazyků, LR(1) algoritmus je tedy použitelný nejen pro všechny LL(1) jazyky. To je jeho jasná výhoda. Nevýhoda ale je, že LR(1) automat obsahuje příliš mnoho stavů, protože pro každý stav LR(0) automatu má tolik stavů, kolik je pro reprezentovaná pravidla různých dosažitelných výhledů. Tyto stavy ale často není potřeba rozlišovat, protože jejich pravidla by konflikt nevytvářela. Další dvě popsané LR metody se budou snažit nevýhodu velkého množství stavů odstranit, i za cenu ztráty podpory některých LR(1) jazyků.

SLR(1)

Jednoduchý (Simple) LR(1) algoritmus používá také výhled na jeden token, pouze ale v případě redukce. Pokud může token ve výhledu následovat za neterminálem z levé strany redukovaného pravidla, redukce je provedena, v opačném případě se provede posun.

Počet stavů je výrazně menší než u LR(1) parseru, ale stejný jako u LALR(1) parseru, jehož třída podporovaných jazyků je vlastní nadmnožinou SLR(1) jazyků. V případě obecného konstruktoru se tedy dává přednost algoritmu LALR(1) [9, s. 315].

LALR(1)

Protože základní LR(1) parser štěpí stavy na základě různých množin výhledů, může mít mnohem více stavů než odpovídající SLR(1) nebo LR(0) parser. Tyto parsery ale nejsou, jak už bylo popsáno výše, příliš silné. S LALR (Lookahead Augmented LR) parsováním se snažíme redukovat počet stavů LR(1) automatu spojováním podobných stavů. To snižuje počet stavů na stejný, jako má SLR(1), ale zůstává zachována větší síla LR(1) výhledů [6].

Pokud má více stavů LR(1) analyzátoru stejná otečkovaná pravidla a liší se pouze ve výhledu, v LALR(1) konstruktoru se vyskytuje pouze jeden stav s těmito pravidly a sjednocenými množinami výhledů shodných otečkovaných pravidel. Pokud spojením stavů vznikne konflikt, jedná se o LR(1) gramatiku, která není LALR(1). Třída LALR(1) gramatik typicky stačí pro běžné programovací jazyky, je tedy tato metoda nejpoužívanější z metod pracujících zdola nahoru.

RRP LR

Gramatiky s regulární pravou částí (Regular Right Part) se liší od bezkontextových gramatik tím, že pravá strana pravidla může odpovídat nedeterministickému konečnému automatu. Podle [11] dokonce lze sestrojit LR parser, který takovou gramatiku zpracovává v lineárním čase.

Zadání pravidel gramatiky s regulární pravou částí by také nemusel být až takový problém, ten potom nastává při psaní sémantické akce pro takové pravidlo. Ta by byla příliš složitá a nepřehledná. Navíc se dá takové pravidlo vždy přepsat na několik běžných pravidel bezkontextové gramatiky, u kterých se zadají sémantické akce mnohem přehledněji.

GLR

Již víme, že můžeme konstruovat velmi silné a efektivní parsery pro LR gramatiky. Bohužel, některé gramatiky, které bychom chtěli prakticky používat, nepatří do třídy LR gramatik, a pokud se pokusíme je na LR gramatiku upravit, zjistíme, že výsledná gramatika je velmi složitá nebo neposkytuje správnou strukturu pro sémantickou analýzu, obvykle se situace týká obou těchto problémů. To omezuje praktické použití LR parserů. Na druhou stranu, prakticky používané problematické gramatiky jsou typicky téměř LR. Parser pro ně zkonstruovaný obsahuje pouze několik konfliktních stavů. V těchto situacích lze dobře použít zobecněný (generalized) LR parser, který většinou odpovídá běžnému LR parseru, ale v konfliktních situacích používá procházení do šířky s neomezeným výhledem, aby rozhodl, jak správně derivační strom postavit [9, s. 382].

Zobecněný LR (Generalized LR) parser lze použít nad libovolným deterministickým LR parserem a pouze pro ty rozhodnutí, která nejsou vyřešena původním algoritmem, využívá vyhledávání do šířky. V konfliktní situaci rozštěpí nebo okopíruje redukční zásobník automatu a simuluje obě možnosti. Po potřebném počtu kroků jeden ze zásobníků v závislosti na vstupu zanikne, nebo jsou oba zásobníky pro

vstup použitelné. V tom případě se musí GLR parser pro jeden z nich rozhodnout. Poté pokračuje dále původní algoritmus.

GLR algoritmus rozšiřuje použitelnost původních efektivních algoritmů na všechny bezkontextové gramatiky a pro gramatiky, které jsou téměř bez konfliktů vzhledem k původně použitému algoritmu, je také GLR algoritmus velmi efektivní.

2.2 Bison

GNU Bison je obecně použitelný konstruktor parserů, který z bezkontextové gramatiky vytváří deterministický LR nebo zobecněný GLR parser používající LALR(1) tabulky. Je částečně kompatibilní s konstruktorem Yacc (standardně používaný nástroj v Unixových systémech): všechny gramatiky napsané pro Yacc by měly fungovat s Bisonem bez jakékoli změny [2].

Původně unixový nástroj je naprogramován v jazyce C a pro šablony využívá makro procesor M4. Primárně generuje parser rovněž v jazyce C, ale podporuje i C++ a experimentálně Javu. Bison je na jazyk C primárně orientován, některé jeho vlastnosti tedy někdy není jednoduché přenést i do ostatních výstupních jazyků.

Bison je svým zaměřením na konstrukci LALR(1) parserů nejblíže zadání této práce, je zde tedy věnován větší prostor pro rozbor jeho vlastností a funkcionalit. Generované parsery jsou optimalizovány na rychlost, předpokládá se tedy, že daní za lepší objektové rozhraní bude právě zpomalení, které by ale nemělo přesáhnout hranici řádů.

2.2.1 Parametry konstruktoru

Konstruktor Bison je program spustitelný z příkazové řádky, nástroje pro sestavení programu nebo vývojového prostředí. Gramatika jazyku, pro který chceme zkonstruovat parser je uložena v souboru, který je předáván konstruktoru jako parametr. Další volitelné parametry mohou ovlivňovat běh konstruktoru i samotný výstup. Některé vlastnosti se dají ovlivnit jak na příkazové řádce, tak uvnitř souboru s gramatikou, což může být občas matoucí. Například jsou-li sémantická pravidla uvnitř souboru s gramatikou psána v nějakém programovacím jazyce, nemá smysl výstupní jazyk měnit z příkazové řádky.

Ačkoliv Bison nabízí několik parametrů pro změnu názvu vygenerovaného souboru, při verzi pro C++ generuje další hlavičkové soubory vždy do pracovního ad-

resáře, vždy stejně pojmenované. Není tedy možné zvolit jiný adresář pro umístění vygenerovaných souborů.

Některé vlastnosti se dají mimo příkazové řádky a parametrů uvnitř souboru s gramatikou dokonce změnit i přímo v kódu hojným používáním maker preprocesoru. To je velmi dobře použitelné pro výstupní jazyky, které preprocesor mají. Bohužel pro jazyky, které preprocesor nemají, tuto vlastnost Bison nijak nepodporuje a je tedy často velký rozdíl v jejich použití i rozhraní.

Jinak jsou parametry příkazové řádky velmi bohaté a zejména možnost ovlivnit úroveň chybových, ladících a informačních hlášení může být užitečná. Taktéž parametry uváděné uvnitř souboru s gramatikou pokrývají mnoho vlastností a je tedy možné vygenerovaný parser dobře přizpůsobit.

2.2.2 Priorita operátorů

Častým úkazem v běžně používaných formálních gramatikách jsou matematické výrazy nebo obecně výrazy, které jsou tvořeny infixovými binárními operátory. Jednoduše bychom chtěli definovat, že výraz může být číslo, ale také součet nebo součin dvou podvýrazů:

```
expr: number
expr: expr + expr
expr: expr * expr
```

To ale bez dalších informací vytváří `shift/reduce` konflikty, protože při zřetězení stejných nebo rozdílných operátorů ve výrazu vzniká nejednoznačnost. Bison tento problém řeší zavedením priority a asociativity terminálů, které jsou používány jako operátory.

Při zřetězení `+` a `*` bychom chtěli, aby se přednostně vyhodnotily součiny a až poté se sčítalo. Musíme tedy nastavit, aby mělo `*` větší prioritu. Při zřetězení stejných operátorů chceme, aby se výraz vyhodnocoval zleva doprava, tedy asociativita obou těchto operátorů bude levá. Pravá asociativita se využije například u operátoru přiřazení.

Pro definici terminálů má Bison deklaraci `token`, která v souboru s gramatikou definuje jména terminálů bez priorit. Další deklarace `left, right, nonassoc` se používají místo původní deklarace `token` pro definici terminálů, které mají navíc definovanou asociativitu, podle použité deklarace. Priorita operátorů je určena automaticky podle pozice deklarace. Dřívěji definované terminály mají nižší prioritu.

Dva terminály, které jsou definovány najednou, tedy jednou deklarací, mají stejnou prioritu a při jejich zřetězení se uplatní asociativita.

Ve skutečnosti není konflikt mezi dvěma terminály, ale mezi terminálem (shift) a mezi pravidlem (reduce). Bison přiřadí prioritu pravidlu automaticky podle posledního terminálu na pravé straně pravidla. Aby mohl uživatel toto chování změnit, zavádí Bison deklaraci `prec`, která se může použít za pravidlem. Jako parametr tato deklarace přijímá terminál, od kterého má být převzata priorita.

2.2.3 Sémantické hodnoty

Symboly gramatiky, ať už terminály či neterminály, samy o sobě nesou pro syntaktickou analýzu zásadní informaci. Pro sémantickou analýzu bývá ale užitečná ještě přídavná hodnota. Na tuto hodnotu je odkazováno ze sémantických akcí. Bison tuto vlastnost podporuje, uživatel si dokonce může vybrat, zda chce pro všechny symboly používat stejný typ sémantické hodnoty, nebo může specifikovat různé typy pro různé symboly gramatiky.

Ani jeden případ ale není úplně vhodně implementován pro moderní objektové jazyky. Konstruktor sice podporuje nastavení typů u jednotlivých symbolů gramatiky, nijak ale nekontroluje, zda uživatel následně nepoužije jiný typ. Uživatel tak může použít pro jeden symbol gramatiky na různých místech různý typ, což může vést ke špatně odhalitelným chybám ve vygenerovaném parseru. Implementace uvnitř parseru je také v obou případech nevhodná, v případě různých typů dokonce často nepoužitelná.

V případě jednoho typu pro všechny symboly je vždy sémantická hodnota neterminálu na levé straně konstruována okopírováním hodnoty prvního symbolu z pravé strany, nebo je použit výchozí konstruktor bez parametrů pokud je pravá strana prázdná.

V případě různých typů je pro C++ použita deklarace `union`, což je spíše zastaralý konstrukt pozůstalý z jazyka C. Nemůže obsahovat složitější objekty a nepodporuje správné vyvolání destruktorů. Bison sice přidává parametr, pomocí kterého se dá definovat destruktor jednotlivých typů, toto řešení ale není příliš čisté.

Pro Javu není různorodost sémantických hodnot vůbec podporována, namísto toho je doporučen předchozí model s jedním typem pro všechny symboly. S využitím polymorfismu a případného ručního přetypování. Vzhledem k typové struktuře v Javě to může být často přijatelné řešení, chybí zde ale opět typová kontrola od konstruktoru. Navíc toto řešení není vůbec použitelné, chce-li uživatel použít přímo nějaký primitivní typ v kombinaci s objekty.

2.2.4 Sledování pozice

V některých aplikacích je vhodné, abychom věděli, jaké místo ve zdrojových datech je reprezentováno jednotlivými symboly gramatiky. Například v chybovém hlášení je typicky vhodné uvést číslo řádku, na kterém se vyskytuje chyba. Bison zavádí funkcionalitu `Locations`, která umožňuje u každého symbolu vedle sémantické hodnoty držet také pozici.

Výchozí struktura pro implementaci v C reprezentuje rozsah řádků a sloupců, lze ji ale pomocí makra změnit. Taktéž výchozí inicializaci hodnoty pozice, která se provádí vždy při redukci pravidla lze změnit makrem.

Implementace v C++ je objektová. V podstatě zastává stejnou funkcionalitu jako v C, jen je rozdělena do dvou tříd. Třída `position` reprezentuje jednu konkrétní pozici, tedy řádek a sloupec, v souboru. Třída `location` pak reprezentuje rozsah dvou pozic. Inicializace `location` lze také změnit předefinováním makra.

Pro implementaci v Javě je rozhraní podobné jako v C++, třídu `position` ale musí implementovat sám uživatel. Protože Java nemá preprocesor, nelze nejspíš ze souboru s gramatikou změnit výchozí inicializaci `location`.

Pro některé aplikace je struktura pozice příliš složitá, pro jiné zase nemusí dostačovat. Je dobré, že Bison podporuje dosazení vlastního typu. Oddělení definice typu struktur a inicializace jejich hodnoty ale není příliš šťastné řešení.

2.2.5 Výjimky

Kromě povinných deklarací `throws` v Javě Bison uživatelské výjimky vůbec neřeší, přičemž ani není jasné, zda je proti vyvolaným výjimkám imunní. V C++, zejména s použitím `union`, může docházet k únikům dynamicky alokované paměti.

2.2.6 Zotavení ze syntaktických chyb

Bison disponuje mechanismem, který umožní zotavení parseru při syntaktické chybě. Pokud uživatel nespecifikuje žádné pravidlo se speciálním terminálem `error`, při syntaktické chybě parser generovaný Bisonem vyvolá funkci pro ohlášení chyb a skončí syntaktickou analýzu. Ta může být následně nastartována znovu, ale je ztracen kontext.

Pokud uživatel použije terminál `error` v nějakém pravidle, parser při výskytu syntaktické chyby toto pravidlo použije pro zotavení, aby mohl v syntaktické analýze pokračovat. Najde v takové situaci nejbližší použitelné pravidlo s terminálem `error` a pokud jsou na zásobníku přebytečné stavy, odstraní je, aby se dostal do stavu, kde

je možné aplikovat terminál `error`. Pak čte parser vstup, dokud nenarazí na symbol, který může podle pravidel gramatiky po terminálu error následovat. Tím je parser opět ve známém stavu a může pokračovat dále ve své činnosti, jako kdyby k chybě v syntaxi nedošlo [8, s. 109].

Zotavení z chyb je vždy jen odhad a řeší pouze typické chyby. Naopak, při chybách, které vývojář gramatiky neočekává, může chybné zotavení způsobit následující syntaktické chyby, které ve skutečnosti neexistují. Vždy ale platí, že syntakticky korektní vstup není zotavením poškozen a že první odhalená chyba je fakt. Co se stane po zotavení, nemusí být kvůli rušení části kontextu a části vstupu úplně korektní.

2.2.7 Data parseru

Data, která parser potřebuje jsou uložena v několika tabulkách. Některé tyto tabulky jsou typicky velmi řídké, tzn. mají mnoho stejných triviálních prvků. Konstruktor Bison provádí na některých tabulkách perfektní hashování, aby zmenšil paměťovou náročnost vygenerovaného parseru. Ten má zároveň zaručen konstantní přístup do tabulek.

2.3 Antlr

ANTLR (ANother Tool for Language Recognition) je výkonný konstruktor parserů, který lze použít na čtení, zpracovávání, vykonávání nebo překládání strukturovaného textu nebo binárních souborů. Je široce používaným nástrojem na sestavování všech druhů jazyků, nástrojů a systémů, v akademické sféře i průmyslu.

Z formálního popisu jazyka, zvaného gramatika, generuje LL(k) parser, který může automaticky sestavit syntaktický strom, což je struktura reprezentující, jak vstup odpovídá dané gramatice. Generuje syntaktické i lexikální parsery, a dokonce umí i automaticky sestavit syntaktický strom a nechat ho následně procházet a v určitých uzlech spouštět uživatelský kód. To vše v mnoha cílových programovacích jazycích obsahující Javu, C, C++, C#, Python a další [1].

ANTLR konstruuje parsery pracující shora dolů s uživatelsky nastaveným výhledem. Může tedy podporovat všechny `LL(k)` gramatiky, kde `k` je uživatelsky zadáno. Nestačí-li pro zadanou gramatiku výhled 1, může jej uživatel zvýšit a nemusí gramatiku přepracovat. Konstruktor umožňuje zadat bezkontextovou gramatiku s regulárními pravými stranami.

Součástí pravidel gramatiky jsou také sémantické akce, ty se mohou vyskytovat kdekoli uvnitř pravé strany pravidla. Je to nutné z toho důvodu, že ANTLR podporuje regulární pravé strany pravidel a u těch se musí sémantické akce provádět uvnitř případných opakování. Často to vede k nepřehlednostem ve vstupních souborech. Přímo u pravidla uživatel také zadává, jaký je typ sémantické hodnoty výsledného neterminálu, a sémantické hodnoty pro jednotlivé symboly pravidla si může lokálně pojmenovat.

Konstruktor ANTLR je naprogramován v Javě a je také na Javu primárně zaměřen. Nejnovější verze 4 dokonce zatím nepodporuje ani jiné cílové jazyky. Předchozí verze 3 podporuje množství cílových jazyků a tak se dá předpokládat, že i nová verze bude časem obohacena. ANTLR je stále aktivně vyvíjen, nicméně opět platí, že všechny funkcionality jsou obsaženy pouze pro cílový jazyk Java. Ostatním cílovým jazykům mohou méně podstatné části chybět. Například ANTLR verze 3 nepodporuje automatické sestavení syntaktického stromu pro cílový jazyk C++. Na šablonování a generování kódu je použit šablonovací nástroj `StringTemplate` programovaný taktéž v Javě.

Parsery generované ANTLR se nedají spustit samostatně, ale pouze s ANTLR runtime library, která je portována pro různé cílové jazyky.

2.4 Lime

PHP je moderní objektový interpretovaný jazyk používaný nejen pro generování webových stránek. Neexistuje příliš mnoho možností, jak pro PHP zkonstruovat efektivní parser a programátoři se často pouštějí do ruční implementace parseru. Existuje několik rozšíření běžně používaných generátorů, tyto rozšíření ale většinou nepodporují všechny vlastnosti obecných generátorů zaměřených na jiné cílové jazyky a jsou často zastaralé.

Lime je konstruktor LALR(1) parserů zaměřený na PHP. Je naprogramován v PHP a vygenerovaný parser je taktéž v jazyce PHP. Implementuje ale pouze základní funkčnost a nemá řádnou dokumentaci[13].

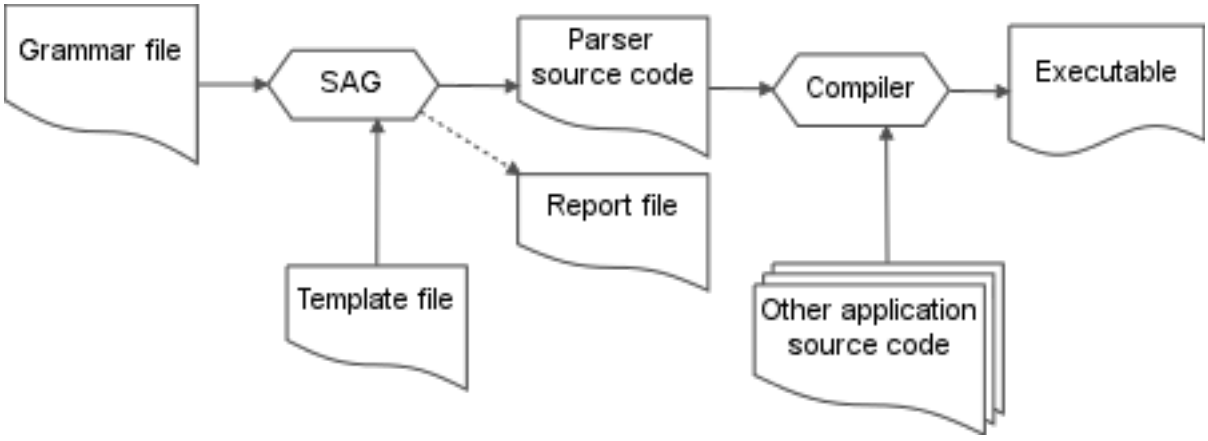
3. Architektura

Pro účely této práce byl vytvořen projekt SAG (z angl. Syntax Analyzer Generator), který obsahuje veškerou implementaci i dokumentaci výsledného nástroje. SAG je rozdělen na hlavní program, konstruktor, který zpracovává zadanou gramatiku, a parser, který je konstruktorem generován. Protože SAG podporuje více cílových programovacích jazyků, obsahuje implementace parseru pro všechny tyto jazyky. Hlavní rysy parseru jsou ve všech jazycích stejné, implementace se liší typicky jen v technických detailech, aby byly dodrženy požadavky a zvyklosti cílových jazyků.

Aby nebyly zdrojové kódy generovaného parseru přímo součástí spustitelného souboru konstruktoru, a bylo umožněno uživateli provádět případné menší změny bez nutnosti překompilovat konstruktor, používá SAG šablonovací systém. Konstruktor vybere šablonu pro požadovaný cílový jazyk, vyplní proměnlivá data do struktury, kterou předá šablonovacímu systému, a ten poté šablonu vyplní a uloží ji do souboru na disk. Tento soubor je samostatným zdrojovým kódem parseru, obsahuje vše, co potřebuje k běhu. Není zapotřebí žádné běhové podpory či nestandardní knihovny.

Na obrázku 3.1 je znázorněno, jak lze ze zadané gramatiky získat spustitelný parser. Gramatika je uložena v souboru, jehož název je předán konstruktoru jako parametr příkazové řádky. SAG použije soubor se šablonou parseru v požadovaném cílovém jazyce a vygeneruje zdrojový kód parseru. Volitelně také může vygenerovat popis automatu do textového souboru. Je-li cílový jazyk kompilovatelný, předá se soubor se zdrojovým kódem parseru, případně společně s ostatními zdrojovými soubory aplikace, kompilátoru, který vytvoří spustitelný soubor. Pokud je cílový jazyk interpretovaný, lze vygenerovaný zdrojový soubor parseru přímo použít s ostatními soubory aplikace.

Obrázek 3.1: Sestavení spustitelného parseru



4. Formát gramatiky

Vstupem konstruktoru je příkazová řádka a vstupní soubor s gramatikou. Název vstupního souboru je též součástí parametrů příkazové řádky. Při tvorbě jeho formátu bylo vycházeno z dobře zavedené struktury vstupního souboru pro konstruktor Bison.

Soubor je rozdělen do dvou, případně tří sekcí oddělených dvěma znaky procenta. První sekce obsahuje deklarace ovlivňující parametry konstruktoru a další potřebné informace. Druhá sekce obsahuje pravidla gramatiky včetně sémantických akcí a volitelná třetí sekce obsahuje uživatelský kód, který je přepsán přímo do zdrojového souboru parseru.

Pro parsování vstupního souboru v konstruktoru je použit parser vygenerovaný SAGem s lexikálním analyzátorem vygenerovaným pomocí nástroje flex [12]. Přesný formát vstupního souboru zapsaný pomocí gramatických pravidel je obsažen v příloze A. Nicméně hodně práce odvede již lexikální analyzátor, který také provádí část sémantiky.

4.1 Deklarace

Oproti konstruktoru Bison je přesněji definována podoba všech deklarací. Deklarace vždy začíná znakem procenta, následována názvem deklarace a jejími parametry. Nic jiného (kromě komentářů) být v první sekci nesmí.

Byly zavedeny tři tvary parametrů deklarace, rozlišené zápisem. Každá deklarace vyžaduje parametry v určitém tvaru a tvar parametru také často vyjadřuje, na co je jeho hodnota následně použita.

- Název je prostý identifikátor složený alfanumerickými znaky a podtržítky, přičemž nesmí začínat číslicí.
- Typ je identifikátor popsaný výše obalený špičatými závorkami. Je použit tam, kde se očekává název typu sémantické hodnoty definovaný v deklaraci `union`.
- Kód je libovolný text uzavřený do složených závorek. Jeho obsah je vždy přímo přepsán do zdrojového kódu výsledného parseru beze změny.

Parametr ve tvaru kódu může obsahovat cokoliv, typicky by ale měl obsahovat kód validní v cílovém jazyce. Podle nastavení cílového jazyka se lexer snaží kód zanalyzovat a počítá vnoření složených závorek. Nepočítá ovšem závorky, které jsou součástí textových řetězců a komentářů, či jiných konstrukcí v cílovém jazyce, které

nemusí dodržet správné uzávorkování. Lexer proto obsahuje stavy, které jsou specifické pro jednotlivé cílové jazyky, aby bylo možné pro každý cílový jazyk správně tento tvar parametrů zpracovat.

Protože je nutné vědět, pro jaký cílový jazyk se bude parser generovat, dříve, než se bude zpracovávat nějaká deklarace s parametrem ve tvaru kódu, musí být deklarace určující cílový jazyk umístěna v souboru jako první. Pokud zde umístěna není, použije se výchozí cílový jazyk, momentálně nastavený na C++.

4.2 Pravidla gramatiky

V této sekci se mohou kromě komentářů vyskytovat pouze sady pravidel začínající názvem neterminálního symbolu a dvojtečkou a vždy končící středníkem. Tato podmínka byla zavedena kvůli přehlednosti, je tak vždy jasné, kde pravidlo končí.

Před dvojtečkou se tedy nachází levá strana pravidla, pro bezkontextovou gramatiku je to vždy jeden neterminál. Zda název symbolu zde uvedený odpovídá terminálu či neterminálu rozhoduje lexikální analýza. Pokud by zde uživatel omylem uvedl název terminálu, byla by to syntaktická chyba. Proto je zde zavedeno ještě další pravidlo s terminálem, oznamující chybu s uživatelsky pochopitelnějším chybovým hlášením.

Za dvojtečkou může být jedna pravá strana přepisovacího pravidla, nebo více pravých stran oddělených svislicí. Na pravé straně mohou být symboly gramatiky (terminály či neterminály), zakončené sémantickou akcí zapsanou ve složených závorkách. Symboly nebo sémantická akce může úplně chybět. Navíc může pravá strana pravidla obsahovat vložené sémantické akce, které jsou ale konstruktorem hned překonvertovány na obyčejné koncové sémantické akce s pomocným neterminálem. Také je možné za seznamem symbolů změnit pomocí `%prec` následovaného názvem terminálu prioritu a asociativitu pravidla, která se uplatní v případném Shift/Reduce konfliktu.

Pro sémantické akce platí podobná pravidla jako pro kódové parametry deklarací, počítají se zde složené závorky podle syntaxe cílového jazyka. Navíc je zde možno používat značky pro přístup k sémantické hodnotě a hodnotě pozice jednotlivých symbolů pravidla tak, jak je zvykem již v konstruktoru Bison. Navíc SAG zavádí novou značku `$!`, kterou může uživatel využít pro inicializaci sémantické hodnoty neterminálu na levé straně, pokud používá deklaraci `%union`. Bez této značky totiž bylo nutné nejdříve zkonstruovat konstruktorem bez parametrů hodnotu daného typu

a až poté přiřadit požadovanou hodnotu. S použitím této značky následované běžným voláním funkce jsou parametry přímo předány do konstruktoru.

V Bisonu platí, že je sémantická hodnota odvozena od hodnoty prvního symbolu pravé strany pravidla, a pokud je pravá strana pravidla prázdná a chybí i sémantická akce, vypíše Bison varování. To je často nechtěné chování. SAG nikdy nekonstruuje explicitně sémantickou hodnotu. V objektově orientovaném návrhu se předpokládá, že typy použité pro sémantické hodnoty se umí samy inicializovat, případně si je uživatel inicializuje sám v sémantické akci. Chybějící sémantická akce je tedy brána stejně jako prázdná.

Pokud uživatel použije deklaraci `%union` pro určení různých typů sémantických hodnot, může u značek odkazujících se na sémantickou hodnotu používat vložený název typu. U symbolů, které nemají sémantický typ předem definovaný, je užití názvu typu povinné. Konstruktor SAG kontroluje shodu typu v definici a v případných referencích jednotlivých symbolů, a pokud dojde k nesouladu, informuje o tom uživatele.

5. Parser

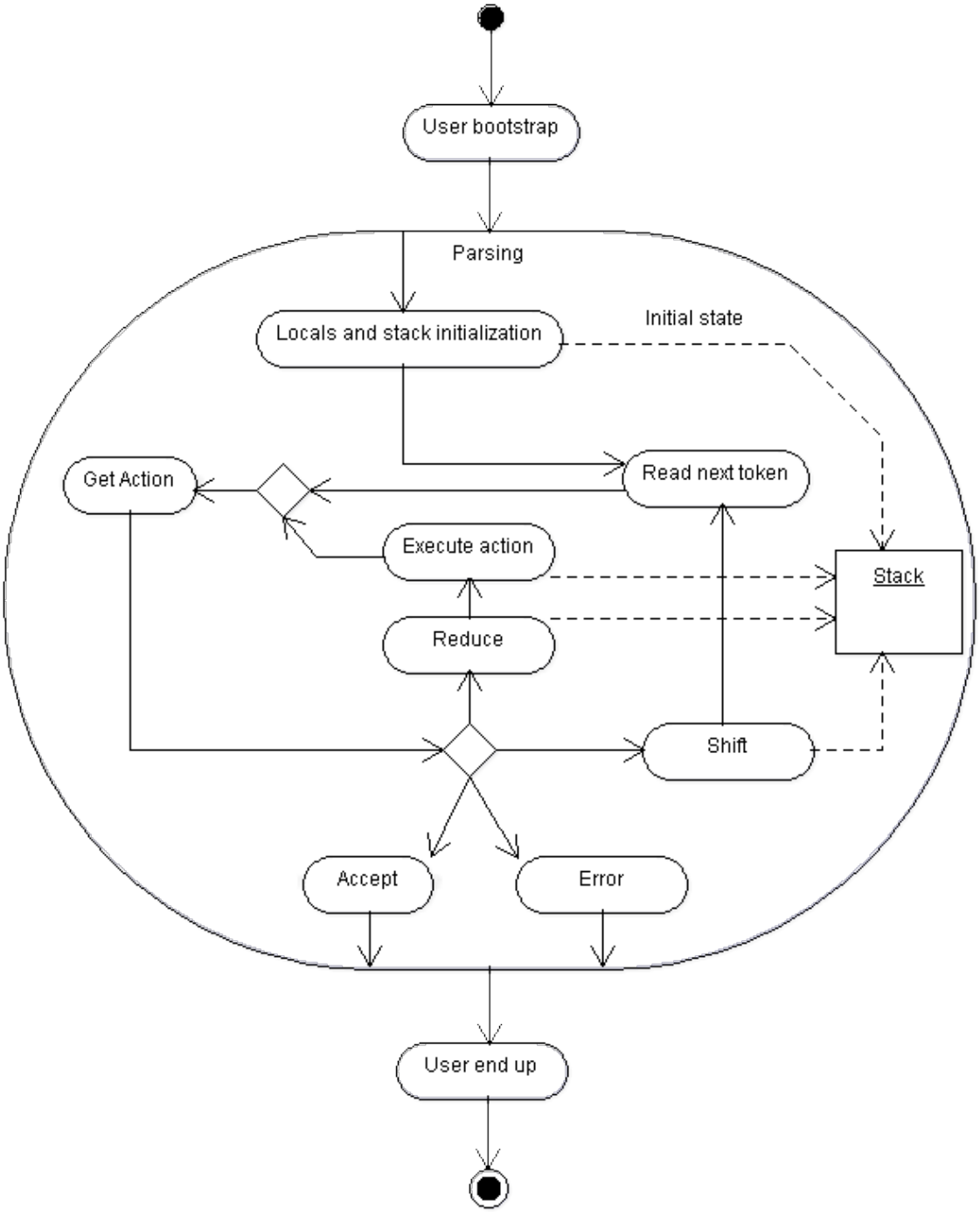
Parser je část programu, jejíž zdrojové kódy jsou generovány z konstruktoru. Tyto musejí být typicky doplněny o další zdrojové kódy výsledné aplikace. V této kapitole si popíšeme, jak vygenerovaný parser funguje, jak je implementován, a co musí uživatel dodat, aby byl parser funkční.

Než začne parser analyzovat vstup, musí být nastartován. Po instanciaci třídy `Parser` a zavolání metody `parse` se ujme řízení vygenerovaný kód. Na obrázku 5.1 je znázorněno, že při spuštění parsování dojde k inicializaci lokálních proměnných, včetně zásobníku automatu. Poté je přečten token ze vstupu, podle stavu automatu na vrcholu zásobníku a posledního přečteného tokenu se automat rozhodne, co má následovat.

- Accept - přijmout vstup a ukončit činnost
- Error - ohlásit syntaktickou chybu a ukončit činnost
- Shift
  - přidat stav reprezentující přečtený vstup na zásobník
  - přečíst další token
  - pokračovat další akcí podle nového stavu zásobníku
- Reduce
  - vyhledat podrobnosti o pravidle, které se má redukovat
  - připravit novou položku zásobníku
  - spustit uživatelskou akci, která může novou položku ovlivnit
  - odstranit z vrcholu zásobníku stavy odpovídající pravé straně pravidla
  - přidat na zásobník nový stav
  - pokračovat další akcí podle nového stavu zásobníku

Po ukončení činnosti parseru se řízení vrací uživateli, který se z návratové hodnoty metody `parse` dozví, zda přečtený vstup odpovídal požadované gramatice či nikoliv.

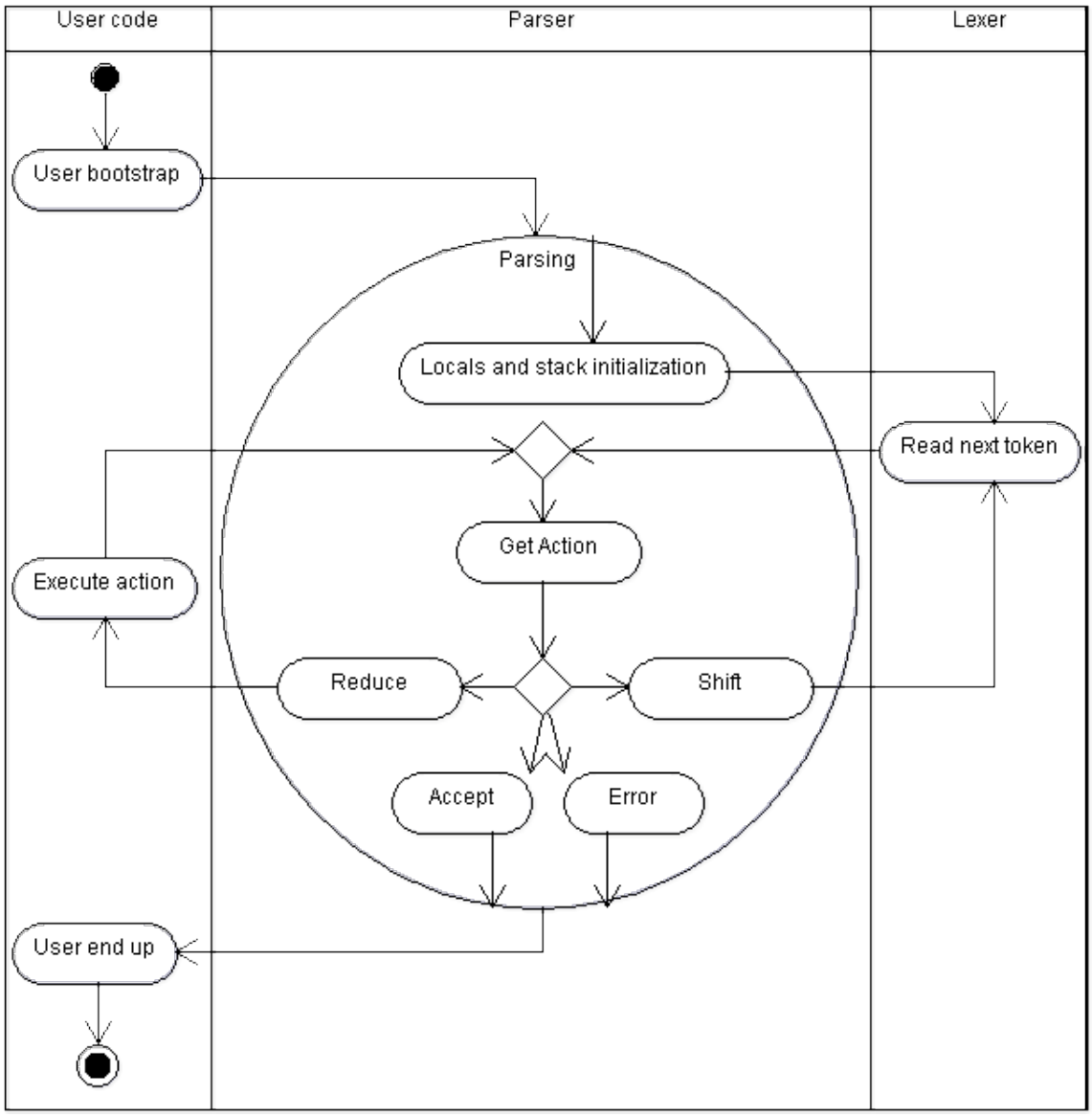
Obrázek 5.1: Průběh činností parseru



Následující obrázek 5.2 ukazuje, co je implementováno přímo vygenerovaným kódem, a co musí uživatel implementovat sám. Samozřejmě musí implementovat spouštěcí rutinu, která parser nastartuje a nakonec se do ní řízení zase vrátí. Také

implementuje sémantické akce, které jsou ale součástí vstupní gramatiky, čili jsou součástí vygenerovaného kódu parseru. Logiku sémantických akcí ale má uživatel pod svou kontrolou. Dále musí uživatel implementovat lexer, tedy funkcionalitu provádějící lexikální analýzu, která na vyžádání poskytne parseru jeden další token.

Obrázek 5.2: Rozdělení rolí v parseru



Veškerý vygenerovaný kód je obsažen v jednom (v případě C++ ve dvou, hlavičkovém a implementačním) souboru, aby bylo docíleno maximální zapouzdřenosti parseru a uživateli se tak se zdrojovým souborem lépe manipulovalo. SAG generuje

parser pro jazyky C++, Java a PHP, ale přidání dalšího, nejlépe objektového, jazyka je podporováno.

Aby bylo docíleno opravdu maximální zapouzdřenosti, jsou v jazycích, které to podporují (tedy ne v PHP), použity pro všechny entity vnitřní třídy. V C++ je navíc použit jmenný prostor, ve kterém je implementován i `Parser`, všechny ostatní třídy jsou implementovány uvnitř třídy `Parser`. V Javě je uvedení balíčku (package) nepovinné, všechny třídy jsou také definovány uvnitř třídy `Parser` jako statické (veřejné nebo privátní).

5.1 C++11

Zavedením nové normy C++11, předem pojmenované jako C++0x, se do C++ dostalo mnoho užitečných funkcionalit. Mnoho uživatelů ale nemůže tyto nové funkcionality využívat, nebo požadovat po překladači, protože překladače je zatím plně neimplementují. Nevyužít tyto funkcionality by ale také nebylo dobré, do budoucna se jistě stanou běžným standardem. Jedna varianta tedy byla, že by parser pro C++11 byl oddělený, tedy další podporovaný jazyk. To by ale vedlo k obrovské duplikaci kódu, většina jazyka a základní konstrukce zůstaly stejné. Navíc by ani to nebylo řešení, protože některé překladače se snaží C++11 podporovat, jen jim zatím chybí jen některé funkcionality.

Použité řešení tedy počítá s jednou implementací parseru pro jazyk C++ a použitím maker se vypořádá s novými vlastnostmi C++11.

```
#if SAG_CPP11
    #ifndef SAG_CPP11_VARIADIC_TEMPLATES
        #define SAG_CPP11_VARIADIC_TEMPLATES 1
    #endif
    #ifndef SAG_RVALUE_IF_CPP11
        #define SAG_RVALUE_IF_CPP11 &&
    #endif
    #ifndef SAG_FORWARD_IF_CPP11
        #define SAG_FORWARD_IF_CPP11(t, v) std::forward<t>(v)
    #endif
#else
    #ifndef SAG_RVALUE_IF_CPP11
        #define SAG_RVALUE_IF_CPP11 /* no RVALUE */
    #endif
    #ifndef SAG_FORWARD_IF_CPP11
        #define SAG_FORWARD_IF_CPP11(t, v) (v)
...
```

Pro uživatele to má obrovskou výhodu v tom, že vygenerovaný parser může využít právě ty funkcionality C++11, které použitý překladač podporuje. Pokud překladač nepodporuje ani základní funkcionality C++11, může definicí makra `SAG_CPP11=0` úplně podporu C++11 v parseru vypnout. Vygenerovaný parser bude tedy použitelný i v budoucnu, kdy překladače C++11 implementovat budou, i když se uživatel nyní rozhodl C++11 nevyužít.

5.2 Rozhraní pro lexer a hlášení chyb

Rozhraní pro lexikální analýzu musí obsahovat metodu, která při zavolání vrátí následující token. Dále by mělo podporovat zjištění sémantické hodnoty a pozice tokenu. Když uvážíme objektový návrh rozhraní, můžeme definovat třídu `Lookahead`[1], která bude zapouzdřovat tyto informace o následujícím tokenu a jediná metoda bude instanci této třídy vracet. Lexer si tak nemusí ukládat žádné informace o naposledy přečteném tokenu.

Rozhraní pro hlášení chyb musí také obsahovat jednu metodu, kterou je uživateli sděleno, na jaké pozici došlo k jaké chybě. Ponechali jsme zde tu možnost definovat i jiné chyby, prakticky se ale volá tato metoda pouze při syntaktické chybě.

Definovat dvě rozhraní s jedinou metodou, nebo definovat jedno rozhraní, které bude zahrnovat vše, co parser od uživatele potřebuje? SAG se vydal druhou cestou a definuje rozhraní `ContextInterface` s metodami `lex` a `error`. Navíc kontext parseru nahrazuje přídavná data, která by uživatel v parseru mohl potřebovat. V Bisonu se tato data deklarovala pomocí `%parse-param`. SAG žádný takový přepínač nemá, ale uživatel si může jednoduše vyrobit v implementaci kontextu parametry jaké potřebuje. Kontext je ze sémantických akcí vždy dostupný. Navíc, pokud bude lexikální analýzu přímo implementovat kontext parseru, budou tato data dostupná i v lexeru.

Parser při své inicializaci očekává instanci `ContextInterface`, a tu také pak poskytuje v sémantických akcích pod identifikátorem `context`. Pokud chce uživatel používat jiný typ, nebo jen podtyp daného rozhraní, může tento typ deklarovat pomocí `%context_type`. Parser pak vůbec nepoužívá definované rozhraní kontextu, ale přímo uživatelsky definovaný typ. Ten musí samozřejmě splňovat určité podmínky, aby se dal používat, nemusí ale rozhraní přímo implementovat.

[1]Názvy tříd a prvků implementace jsou zde uvedeny v angličtině tak, jak jsou pojmenovány přímo v kódu. Někdy jsou v kódu názvy prefixovány `sag, Sag nebo SAG`, aby se předešlo případným jmenným kolizím s uživatelským kódem. Zde budeme uvádět názvy bez prefixu.

5.3 Rozhraní parseru

Navenek se `Parser` prezentuje velmi jednoduše, má jediný konstruktor, který přijímá jako parametr instanci kontextu. Tu si uloží a dále ji poskytuje v sémantických akcích. V C++ je předávána instance ukazatelem, aby bylo zřejmé, že se nebude kopírovat, ale uloží se ukazatel na předanou instanci.

Dále obsahuje jedinou veřejnou metodu `parse` bez parametrů. Tu uživatel volá, když chce začít syntaktickou analýzu. Tato metoda si drží lokálně zásobník automatu, při opakovaném zavolání tedy vždy začne vstup rozpoznávat od začátku gramatiky. Je tedy možné metodu `parse` volat na jedné instanci parseru i rekurzivně.

Vnitřní rozhraní parseru je dáno rozhraním `ContextInterface`, popsaným výše. Metoda `lex` vrací instanci třídy `Lookahead`, kterou musí nejdříve správně vyplnit. Tato třída je jen jednoduchým zapouzdřením symbolu gramatiky, sémantické hodnoty tohoto symbolu a jeho pozice ve vstupu. Vlastní privátně tyto tři prvky a zpřístupňuje je podle zvyklostí objektového programování přes přístupové metody get a set. Typy jednotlivých prvků jsou popsány dále.

Do rozhraní parseru by se dala také zahrnout třída `Token`. Všechny tokeny, které uživatel deklaroval, mají svou pevnou hodnotu (přirozené číslo). Aby bylo možné používat názvy tokenů namísto těchto interních hodnot, obsahuje třída `Token` konstanty pojmenovaných tokenů. V C++ obsahuje výčtový typ se správně přiřazenými hodnotami, v Javě a PHP obsahuje přímo celočíselné konstanty pojmenované podle tokenů.

5.3.1 Sémantické hodnoty

Pro sémantické hodnoty má SAG na výběr dvě řešení. První řešení spočívá v použití stejného typu sémantické hodnoty pro všechny symboly gramatiky, druhé zavádí více různých typů sémantických hodnot. Pro oba přístupy platí, že je sémantická hodnota spolu s typem tokenu vracena z lexeru a ukládána na zásobník. Sémantická hodnota neterminálu se nastaví při redukci pravidla, jež má tento neterminál na levé straně, v sémantické akci uvedené u pravidla. Chybí-li sémantická akce, je prázdná nebo zkrátka nenastavuje sémantickou hodnotu výsledného neterminálu, zůstane sémantická hodnota neinicializovaná.

V případě C++ se u objektů použije v takovém případě výchozí konstruktor bez parametrů. Destruktor sémantické hodnoty symbolů na pravé straně pravidla se volá ihned po zavolání sémantické akce při redukci pravidla. Uživatel by měl vždy používat typy, které se automaticky konstruují i destruují.

V PHP proměnné pevně daný typ nemají, nemá tedy smysl tyto dva případy rozlišovat, zbytek této sekce se tedy parsery v PHP nezabývá, deklarace `%union` i `%stype` je dokonce při použití cílového jazyka PHP zakázaná.

Značky pro přístup k sémantickým hodnotám v sémantické akci přistupují přímo k hodnotě, typ výrazu, kterým je značka nahrazena, je tedy zadaný uživatelský typ.

Jednotný typ sémantických hodnot

Při použití deklarace `%stype` nebo při využití výchozího jednotného typu mají všechny symboly gramatiky stejný typ sémantických hodnot. Všechny symboly sémantickou hodnotu mají, jen ji uživatel nemusí využít.

Ve třídě `Lookahead` je přímo uložen typ zadaný uživatelem a tento typ je také přímo uložen v položce na zásobníku.

V C++ musí být typ použitý pro sémantickou hodnotu kopírovatelný (pokud jde o třídu, musí mít copy-constructor), protože při akci `Shift` se hodnota ze struktury `Lookahead` okopíruje na zásobník.

V Javě může být typ použitý pro sémantickou hodnotu jakýkoli, může to být třída, pole i primitivní typ.

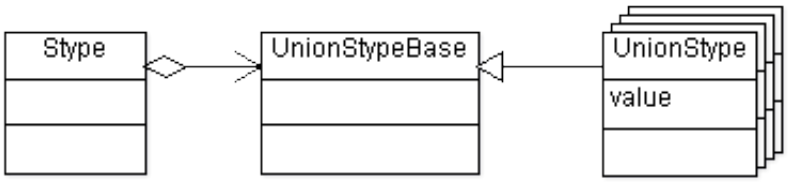
Typicky je jednotypová varianta nedostatečná. Použitelná může být ve velmi jednoduchých parserech, nebo s použitím polymorfismu a dědičnosti. Uživatel si jako typ sémantických hodnot zvolí jediného předka všech požadovaných typů a používá přetypování. Uživatel ve skutečnosti pracuje s více typy sémantických hodnot, z hlediska parseru se ale jedná o typ jeden. Jednotypová varianta je také výhodná, pokud uživateli nevyhovuje implementace vícetypové a implementuje si vlastní.

Různé typy sémantických hodnot

Při použití deklarace `%union` definuje uživatel několik typů, každý typ má název (identifikátor) a kód popisující v cílovém jazyce datový typ.

Implementace různých typů sémantických hodnot spočívá v definici několika pomocných tříd, které uživatelské typy skryjí. Na obrázku 5.3 je zachycena třída `Stype`, která je použita v parseru jako typ sémantické hodnoty pro všechny symboly. Tato třída drží referenci na společného předka `UnionStypeBase`, jehož potomci již mají veřejnou položku s názvem `value` požadovaného uživatelského typu.

Obrázek 5.3: Diagram tříd při použití %union



Zapouzdření je vymyšleno tak, aby uživatel pracoval pouze s třídou `Stype`. Ta obsahuje pro každý uživatelský typ dvě metody. Metoda `set` (její pravý název je ještě doplněn názvem uživatelského typu) vždy ruší hodnotu, která byla přiřazena v objektu dříve, a vytváří novou. Jako parametr přijme data, kterými je následně hodnota inicializována. Metoda `get` (opět má každý typ vlastní název) vrací vždy referenci na správného potomka `UnionStypeBase`. Pokud již objekt hodnotu požadovaného typu obsahoval, vrací tuto hodnotu, pokud neobsahoval žádnou, nová hodnota je automaticky vytvořena, a pokud obsahoval hodnotu špatného typu, selže program v rutině `assert` (pokud je zakázaná, vytvoří se tiše nová hodnota).

V C++ obsahuje `UnionStypeBase` výčtový typ `NameT`, pro každý uživatelský typ jedna položka. Tím se označí potomci této třídy, aby bylo možné kontrolovat typ hodnoty. `Stype` drží ukazatel na třídu `UnionStypeBase`, tuto dynamicky alokuje a dealokuje a ve spolupráci s ní provádí počítání referencí. Potomkem základní třídy je šablonovaná třída se dvěma parametry. První představuje uživatelský typ (uživatelem zadaný kód) a druhý jeho název (hodnota z `NameT`). Třída `Stype` pak definuje pro každý uživatelský typ jednu instanci šablonovaného potomka.

Při použití C++11 a `variadic templates` může mít metoda pro nastavování hodnoty libovolné množství parametrů a tyto parametry jsou předány přímo do konstruktoru uživatelského typu pomocí `std::forward`. Pokud `Stype` neobsahuje správná data, metoda `get` alokuje novou instanci šablonovaného potomka. Hodnota v něm je neinicializovaná, v případě objektu je použit konstruktor bez parametrů. Z toho důvodu je nutné, aby typ zde použitý měl veřejný konstruktor bez parametrů, i když ve skutečnosti nemusí být použit.

V Javě je situace velmi podobná, není ale potřeba počítat reference a explicitně dealokovat data. Java také nepodporuje šablony ve smyslu C++, navíc oproti C++ ale podporuje zjištění typu objektu za běhu, není tedy potřeba výčtový typ pro kontrolu. Třída `Stype` drží datovou položku typu `UnionStypeBase`, ta je ale zcela prázdná. Pro každý uživatelský typ je definována jedna podtřída s veřejnou datovou položkou uživatelského typu. Tyto podtřídy mají dva konstruktory, jeden bez

parametru, kde hodnota zůstává neinicializovaná, druhý s parametrem uživatelsky zadaného typu. Metoda `set` má taktéž jeden parametr, který předá do konstruktoru pro inicializaci hodnoty. Metoda `get`, když vytváří novou hodnotu, alokuje potomka třídy `UnionStypeBase` a použije jeho konstruktor bez parametrů.

Ve struktuře `Lookahead` je uložen typ `Stype`, s výchozím konstruktorem je inicializován na prázdnou hodnotu, tzn. nereprezentuje žádný uživatelský typ. V lexeru uživatel typicky jen hodnotu nastavuje, může tedy použít jednu z metod `set` vytvářející hodnotu požadovaného typu.

V sémantických akcích uživatel přistupuje k hodnotě pomocí značek, které jsou poté nahrazeny výrazem, který ze zásobníku nebo z připravované nové položky získá objekt typu `Stype`, správnou metodou `get` je vrácena reference na potomka `UnionStypeBase` a z ní následně zpřístupněna položka value požadovaného uživatelského typu. Tento výraz může být použit jak na pravé, tak i na levé straně přiřazení.

Navíc SAG zavádí novou značku `$!`, která je nahrazena voláním metody `set` odpovídající typu sémantické hodnoty neterminálu na levé straně pravidla. Uživatel tuto značku použije následovanou voláním metody s předáním parametrů. Především v C++ použití této značky předchází vytváření prázdného objektu následně přepsaného pomocí operátoru přiřazení.

5.3.2 Sledování pozice

Téměř každý parser potřebuje informaci o pozici zpracovávaných struktur ve vstupních datech. SAG vždy tuto funkcionalitu obsahuje, uživatel ji však může úplně ignorovat. Často jsou ale požadavky na sledování pozice náročné, záleží na použití parseru. Z toho důvodu je výchozí implementace velmi jednoduchá a zároveň je myšleno na to, aby si ji mohl uživatel jednoduše přizpůsobit.

Jeden z možných návrhů v C++ spočíval v šablonované třídě `Parser`, kde by se typ pozice předával jako parametr šablony. Implementace by byla velmi čistá a uživatel by si mohl jednoduše dosadit vlastní typ. Problém ale spočíval v tom, že sémantické akce také potřebují s pozicí pracovat, musely by tedy být implementovány v hlavičkovém souboru. Navíc toto řešení není přirozené pro ostatní objektové jazyky, přednost byla dána jiné implementaci.

Protože je pozice automaticky vytvářena parserem, je nutné, aby parser věděl, jaký konkrétní typ se má použít. Nelze tedy jen definovat rozhraní a to v parseru používat. Výchozí typ pozice je v parseru definovaná třída `Location`. Nevyhovuje-li uživateli tato implementace, může deklarací `%location_type` sdělit konstruktoru, aby negeneroval výchozí typ a používal uživatelský.

U terminálů musí nastavit pozici lexer. Ve struktuře `Lookahead` ji předá parseru, který ji okopíruje na zásobník. Při redukci pravidla, ještě před spuštěním sémantické akce, je pozice výsledného neterminálu zkonstruována z pozic symbolů na pravé straně. V sémantické akci již může uživatel použít značky pro přístup k pozici jakéhokoli symbolu pravidla.

Pro konstrukci pozice neterminálu na levé straně z pozic symbolů pravé strany pravidla je použit velmi obecný přístup. Zásobník poskytuje iterátor, kterým lze procházet pozice symbolů umístěné na zásobníku. Vzhledem k tomu, že pozice symbolů pravé strany pravidla jsou ve chvíli redukce na vrcholu zásobníku (pozice posledního symbolu pravidla je na vrcholu, předposledního pod ním atd.), je jednoduché vyrobit iterátor, kterým lze procházet právě pozice symbolů pravé strany pravidla. Pomocí tohoto iterátoru si již sám typ reprezentující pozici výsledného symbolu nastaví vlastní hodnotu.

V C++ je předávána do konstruktoru pozice dvojice iterátorů s náhodným přístupem, první iterátor ukazuje na první symbol pravé strany pravidla, druhý až za poslední symbol. Je-li tedy pravidlo prázdné, jsou oba předané iterátory totožné a uživatel je nesmí dereferencovat.

V Javě implementuje jediný iterátor předaný do konstruktoru pozice rozhraní `java.util.Iterator`. Tento iterátor začíná na prvním symbolu pravé strany pravidla a poslední přístupná pozice patří poslednímu symbolu pravidla. Má-li pravidlo prázdnou pravou stranu, předaný iterátor vrací hned při prvním zavolání metody `hasNext` neúspěch.

PHP 5 také zavádí iterátory, je tedy implementována třída `LocationIterator` implementující rozhraní `Iterator`. V PHP může mít třída pouze jeden konstruktor, ten je tedy ponechán na inicializaci pozice v lexeru, aby to měl uživatel v kódu přehlednější. Pro konstrukci pomocí iterátoru, která se typicky provádí pouze ve vygenerovaném kódu při redukci pravidla, se použije statická metoda `fromRHS` definovaná na typu pozice, které se jako parametr předá `LocationIterator`, a vrací nově zkonstruovanou pozici.

5.4 Výjimky

Moderní programovací jazyky často zavádí pojem výjimka (exception) a obsahují konstrukce, které mohou výjimky vypouštět a odchytávat. V objektových jazycích je typicky výjimka instance speciální (nebo i běžné) třídy, a podle typu se také výjimky při odchytávání filtrují.

Bez použití k tomu určených deklarací se v parseru s výjimkami nic nedělá, jen byla snaha v jednotlivých cílových jazycích o co nejlepší design, který by byl na případné vyvolání výjimky připraven. Především v C++ je tomu třeba věnovat pozornost, aby nedocházelo k únikům dynamicky alokované paměti, nebo se dokonce parser nedostal do nekonzistentního stavu.

Při použití deklarace `%catch` může uživatel specifikovat typy výjimek, které má parser odchytávat. Předpokládá se, že tyto výjimky mohou být vyvolány v sémantické akci. Při odchycení takové výjimky je vyvolána metoda kontextu určená k nahlášení výjimky, na zásobníku je poznamenáno, že došlo k výjimce a syntaktická analýza pokračuje dál. Protože se předpokládá, že výjimka vyplynula z nastavení sémantické hodnoty neterminálu na levé straně redukovaného pravidla, nemusí být tato hodnota použitelná. Z toho důvodu se při redukci pravidel, které mají na pravé straně symbol, jehož sémantická hodnota může být poznamenána výjimkou, vůbec nespouštějí sémantické akce. Spouštěny tedy typicky nejsou sémantické akce od symbolu, při jehož konstrukci sémantické hodnoty došlo k výjimce, až ke kořeni derivačního stromu. Porušit se to může u pravidla, které vůbec sémantickou akci nemá, nemůže být tedy poznamenána sémantická hodnota výsledného neterminálu (protože nejspíš žádnou nemá).

5.4.1 Výjimky v Javě

V Javě existují výjimky, které je nutné u metody deklarovat, je-li možné, že je bude metoda vypouštět. Při použití deklarace `%catch` je automaticky deklarováno u všech sémantických akcí, že typ deklarovaných výjimek mohou vypouštět. Protože jsou v metodě `parse` odchyceny, nikde jinde již být deklarovány nemusí.

V případě, že chceme v Javě nechat výjimky projít parserem, aniž by je odchytával, můžeme použít deklaraci `%throws`, která zaručí, že budou výjimky zde uvedené deklarovány u sémantických akcí i u metody `parse`. Podobný problém můžeme mít s lexerem, zde SAG umožňuje použít deklaraci `%lex_throws`, kde uživatel uvede výjimky, které budou deklarovány u metody kontextu `lex` i u metody parseru `parse`.

5.4.2 Hlášení o výjimkách

Dojde-li k výjimce, je třeba, aby o ní dal parser vědět. Podobně, jako je použita metoda kontextu k hlášení syntaktických chyb, je vyvolána metoda kontextu k hlášení o výjimkách.

V C++ a Javě obsahuje rozhraní `ContextInterface` pro každou výjimku deklarovanou pomocí `%catch` jednu metodu `error` se dvěma parametry. Prvním je pozice výsledného neterminálu, druhým odchycená výjimka deklarovaného typu. Pokud uživatel používá vlastní typ kontextu, může metodu `error` deklarovat jen jednu, například šablonovanou.

V PHP obsahuje `ContextInterface` pouze jednu metodu pro všechny deklarované výjimky nazvanou `exceptionHandler`. Jako parametr také dostane pozici výsledného neterminálu a odchycenou výjimku.

5.4.3 Možná rozšíření

Kdyby se tato funkcionalita ukázala jako užitečná, mohla by být rozšířena o několik menších úprav, které by přispěly k použitelnosti.

Protože je při použití deklarace `%catch` nutné kontrolovat při každé redukci pravidla každý symbol pravé strany pravidla, zda nebyl poznamenán výjimkou, může se výpočet parseru mírně zpomalit. Minimální optimalizace by mohla spočívat v globálním příznaku, který by říkal, jestli vůbec již na zásobníku nějaký poznamenaný symbol byl. Nedojde-li totiž vůbec k výjimce, což může být pravděpodobné při použití výjimek opravdu jen ve výjimečných situacích, nemusel by parser prověřovat každý symbol pravé strany.

Pro jazyky, které výjimky nemají, nebo v případech, kdy uživatel nechce výjimky použít, by mohla existovat alternativní cesta, jak označit sémantickou hodnotu jako poškozenou, a přimět parser, aby nespouštěl sémantické akce, ve kterých by mohla tato poškozená hodnota figurovat. Tedy použít již zavedenou funkcionalitu, jen by se nemusela ovládat pouze výjimkami.

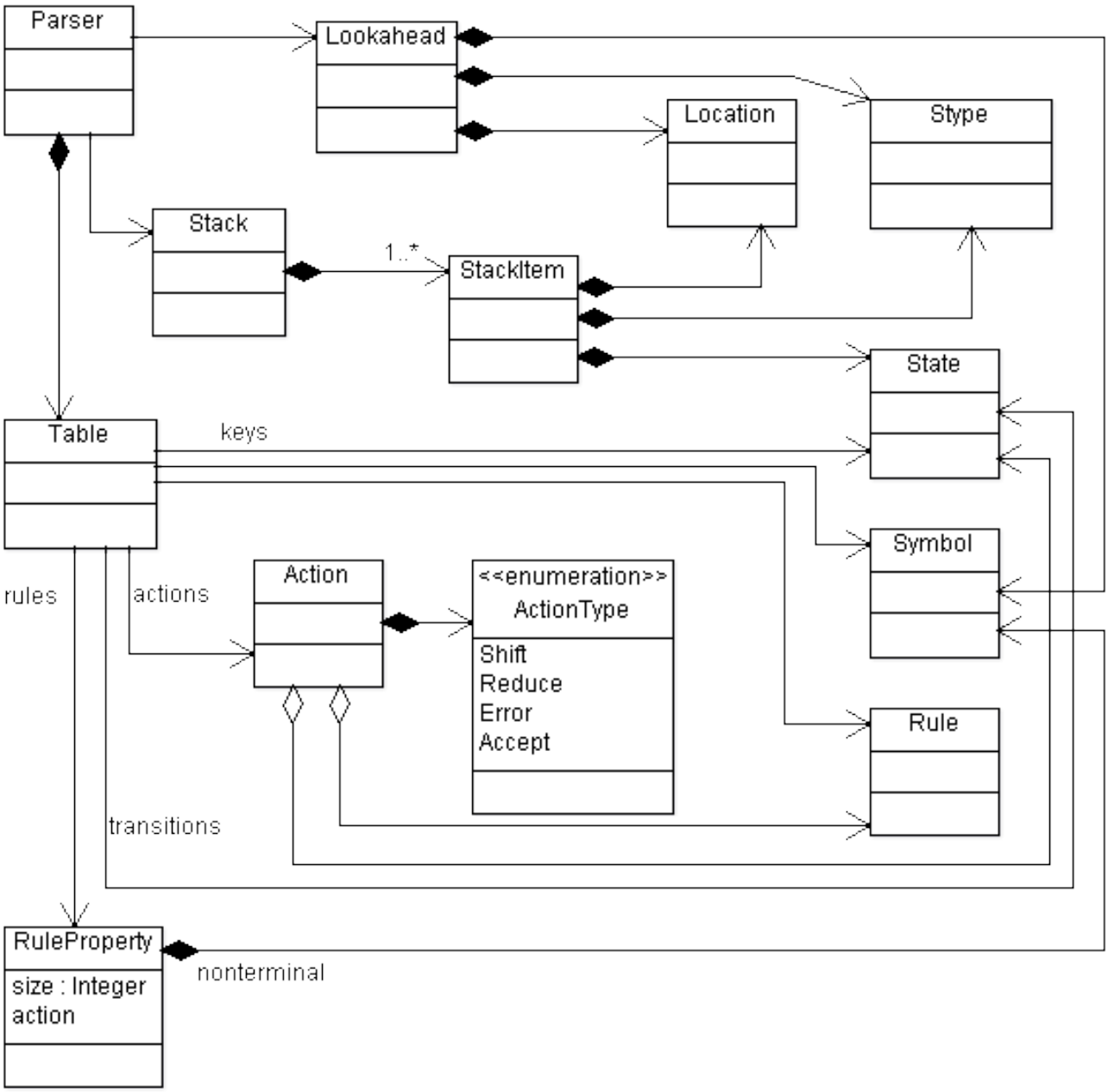
V některých případech by mohlo být užitečné, aby mohl uživatel z metody pro hlášení o výjimce ovlivnit následující průběh analýzy. Například návratovou hodnotou, podle které by se rozhodlo, zda bude parser pokračovat dále, nebo okamžitě skončí (kladně či záporně). Také by se zde dalo ovlivnit, zda se má sémantická hodnota považovat za poškozenou.

5.5 Vnitřní implementace

Uvnitř parseru je implementace také objektová, rozhraní jsou ale spíše volná, aby se dosáhlo vyšší rychlosti. Stále jsou ale logické části parseru dobře oddělené. Obrázek 5.4 znázorňuje rozložení důležitých tříd uvnitř parseru. Hlavní třída `Parser`

vlastní buď jako statickou položku třídu `Table`, nebo do ní přímo staticky přistupuje. Ta implementuje tabulky parseru a data poskytuje přes dobře definované rozhraní. Více o této třídě se dozvíme v 5.5.2. Dále parser komunikuje s již dříve popsanou třídou `Lookahead` obsahující položky pozice, sémantickou hodnotu a symbol. V metodě `parse` se lokálně používá zásobník `Stack`, na kterém jsou umístěny položky, kde každá položka obsahuje pozici, sémantickou hodnotu a stav automatu.

Obrázek 5.4: Vnitřní implementace parseru



5.5.1 Typ indexů

V konstruktoru jsou symboly gramatiky, pravidla a stavy struktury plné informací. Když je již parser zkonstruován, zbude z každé entity jen její index, podle kterého lze vyhledávat v tabulkách. Abychom ještě více ušetřili, snažíme se vybrat pro jednotlivé entity celočíselný typ takový, aby stačil všem potřebným hodnotám, ale zabíral minimum místa. Jedná se o typ následujících entit:

- Symbol - hodnota gramatického symbolu
- State - index stavu automatu
- Rule - index gramatického pravidla
- cíl akce - větší z typů State a Rule, používá se ve struktuře `Action`
- velikost pravé strany pravidla - použije se ve struktuře `RuleProperty`

V C++ jsou vždy použity celočíselné typy bez znaménka, v Javě jsou použity typy byte, char, int nebo long, v PHP se typy indexů neřeší.

V Javě jsou používány především primitivní typy pro indexy, při ukládání do šablonovaných struktur ale použít primitivní typy nelze, tam jsou použity obalové třídy primitivních typů.

5.5.2 Typy struktur

Čistě datové struktury jsou implementovány nejjednodušeji s přímým přístupem k položkám kvůli efektivitě, kontejnerové struktury používají objektové rozhraní kvůli zapouzdřenosti.

Datové struktury

K jednotlivým pravidlům je třeba si pamatovat, jaký neterminál mají na levé straně, kolik symbolů mají na straně pravé a jakou mají sémantickou akci. K tomu slouží třída `RuleProperty`, která tyto informace drží.

Dále je třeba mít uloženy k různým stavům automatu akce, které má provádět. Zde by byl čistší objektový návrh pomocí polymorfismu, kvůli efektivitě byl ale zvolen jiný přístup. Jednoduchá třída `Action` obsahuje hodnoty výčtového typu (v PHP je nahrazen konstantami), což určuje jednu ze čtyř variant `Shift, Reduce, Error, Accept`. Dále obsahuje cíl akce, který se ale použije jen u akcí Shift a Reduce. Pro akci `Shift` je zde uložen stav, do kterého se má automat přepnout, pro akci `Reduce`

pravidlo, podle kterého se redukce provede. Proto se musí do typu cíle akce vejít index stavu i pravidla.

Zásobník

Zásobník automatu má ve všech jazycích stejné rozhraní. Konstruktor inicializuje zásobník přímo s počátečním stavem, metoda `push` vloží jeden prvek na vrchol zásobníku, metoda `pop` odstraní z vrcholu zásobníku daný počet prvků, což se využije při redukci pravidla, kdy je odstraněna ze zásobníku celá pravá strana pravidla najednou. Dále je zajištěn náhodný přístup na zásobník podle vzdálenosti od vrcholu zásobníku a získání iterátoru pozic.

Jako podpůrné kontejnery jsou vybrány samostatně rostoucí pole ze standardních knihoven cílových jazyků, jejichž prvky jsou položky zásobníku. Zásobník vždy obsahuje pouze jeden kontejner.

Položka zásobníku obsahuje všechny tři entity, které je potřeba na zásobníku držet, tzn. stav automatu, sémantickou hodnotu a pozici. Při použití deklarace `%catch` navíc drží booleovskou hodnotu reprezentující příznak, zda je sémantická hodnota poškozena výjimkou. Při přístupu na zásobník je vracena reference na celou položku, jednotlivá data se získají přímým přístupem k prvku, nejsou zde definovány přístupové metody.

Tabulky parseru

V tabulkách parseru jsou uloženy staticky tři kontejnery, akce, přechody a pravidla tak, aby se při případné vícenásobné instanciaci parseru nevytvářely tabulky vícekrát.

V akcích je podle stavu a symbolu uložena akce, kterou má automat provést v daném stavu s daným symbolem ve výhledu. Přechody mají stejný typ klíče, tedy stav a symbol, jeho sémantika je ale jiná. Stav odpovídá stavu automatu, symbol je ale neterminál z levé strany pravidla, jež se právě redukuje. Hodnota přechodu je cílový stav, do kterého má automat přejít po redukci tohoto pravidla.

Pro pravidla je použito jen pole, kde k indexu pravidla můžeme najít strukturu `RuleProperty`, která toto pravidlo popisuje.

Akce a přechody mají stejný typ klíče, jejich struktury budou tedy velmi podobné. SAG neimplementuje perfektní hashování, čili data jsou ukládána přirozeně a v tabulkách se opravdu podle dvojitého klíče hledají.

V C++ je použita `std::unordered_map`, v Javě `java.util.HashMap` a v PHP je využito vestavěných polí, protože PHP u polí hashování řeší.

Tři veřejné metody tabulek zpřístupňují uložená data. Metoda pro získání akce vyhledá podle zadaného dvojklíče akci. Pokud ji nenajde, jedná se o syntaktickou chybu a je vrácena reference na akci reprezentující chybu (typ akce `Error`). Jinak je vrácena reference na nalezenou akci. Druhá metoda pro získání přechodu také vyhledá podle zadaného dvojklíče cílový stav, ten ale vždy najde. Aby jej nenašla, muselo by dojít k vnitřní chybě uvnitř parseru, což je ošetřeno pouze rutinou assert. Poslední metoda pro získání informací o pravidle podle zadaného indexu pravidla vrací příslušnou `RuleProperty`.

5.5.3 Sémantické akce

Sémantická akce je kód, který se má vykonat při redukci pravidla. Konstruktor nahradí značky, které uživatel může použít, kódem, čili parser již značky neřeší. SAG implementuje sémantické akce jako samostatné rutiny, které se při redukci volají. Má to výhodu možnosti lepší optimalizace při překladu a nevytváří se tak potenciálně obrovská funkce obsahující všechny akce.

V C++ jsou sémantické akce instance šablonované metody parseru. Jsou uloženy v `RuleProperty` jako ukazatel na metodu objektu, na objektu parseru jsou pak také volány.

V Javě jsou využity pro implementaci sémantických akcí anonymní vnitřní třídy. Je definováno rozhraní `SemanticAction` s metodou `execute`, která provádí sémantickou akci. Do `RuleProperty` je pak uložena anonymní instance tohoto rozhraní. Při redukci pravidla je pak metoda `execute` zavolána.

V PHP byly použity stejným způsobem anonymní funkce, je proto vyžadována verze PHP alespoň 5.3.0.

V `RuleProperty` se neukládá prázdná sémantická akce, ale zástupná hodnota `null`, podle které parser pozná, že pravidlo sémantickou akci nemá.

5.5.4 Inicializace dat

Data parseru jsou uložena v dynamických strukturách, je tedy třeba je dynamicky naplnit. Nejpřímočařejší řešení bylo naplnit kontejnery opakovaným voláním metody `insert`. To ale vedlo k obrovským rutinám, které dělaly problém při překladu i při běhu. Překladač `gcc` při překladu většího parseru se zapnutými optimalizacemi

vyžadoval příliš mnoho paměti, při běhu pak trvalo příliš dlouho, než se načetly všechny instrukce.

Aktuální řešení spočívá v tom, že je v kódu vytvořena statická struktura, která obsahuje všechna potřebná data, ale není vhodná pro vyhledávání. Procházením této struktury jednoduchým cyklem je při startu programu naplněna dynamická struktura.

6. Konstruktor

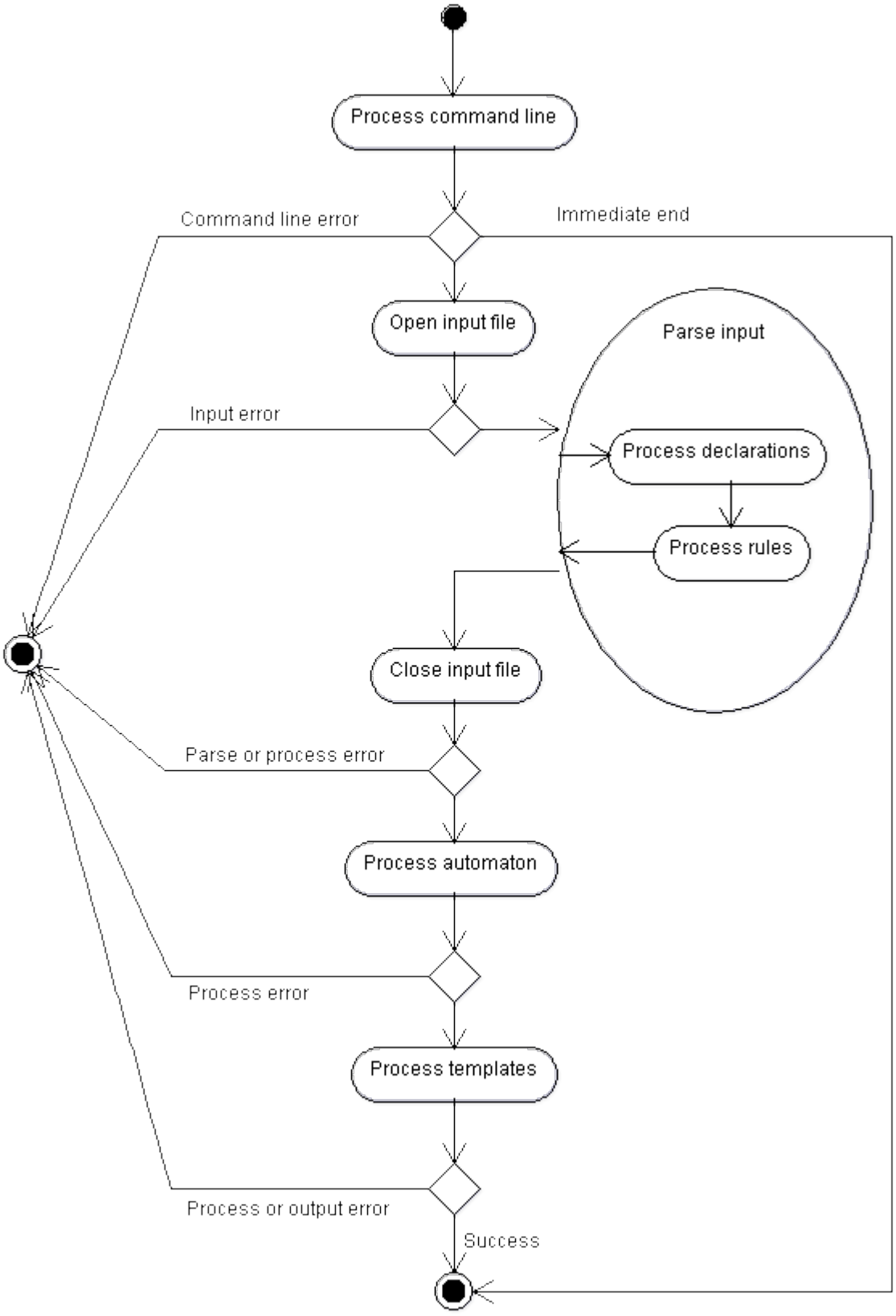
Konstruktor je naprogramován v jazyce C++ a jsou zde využity některé nové vlastnosti uvedené v normě C++11 [4]. Zdrojové kódy jsou rozděleny do podadresářů podle toho, co implementují.

Pro zpracování vstupního souboru je využit lexikální analyzátor generovaný nástrojem Flex [12] v kombinaci se syntaktickým analyzátorem generovaným pomocí SAGu. V počátcích vývoje byl syntaktický analyzátor generovaný Bisonem, až když byl projekt SAG propracovaný, byl nasazen SAG parser.

Pro formátování chybových hlášení je využita knihovna `Boost Format` a pro zpracování parametrů na příkazové řádce knihovna `Boost Program Options`. To se ukázalo jako dobré rozhodnutí, obě knihovny jsou velmi jednoduše použitelné a pracují stabilně. Pro generování zdrojových kódů výsledného parseru byl použit šablonovací systém `CTemplate`, který je podle tvůrců jednoduchý ale výkonný [5]. Po použití v projektu SAG byl zhodnocen jako příliš jednoduchý a zejména nemožnost implementovat v šablonách alespoň minimální logiku byla příčinou přílišného řešení výstupních dat v konstruktoru. To se ale podařilo alespoň dobře oddělit od výkonného kódu konstruktoru, vzniklé řešení je tedy uspokojivé.

Životní cyklus konstruktoru parserů je naznačen na obrázku 6.1. Práce začíná zanalyzováním argumentů příkazové řádky, kde můžou být přepínače, které způsobí okamžitý konec běhu programu (výpis nápovědy nebo verze programu). Také při chybě v předaných argumentech je program ukončen. Následuje otevření vstupního souboru s gramatikou, jeho analýza, během níž jsou zpracovávány deklarace a pravidla v něm uvedené. Po úspěšném zpracování je zkonstruován automat výsledného parseru, pak naplněny šablony a vygenerovány výstupní soubory. Po každé sekci je kontrolováno, jestli nenastala nějaká chyba, a pokud ano, následující sekce již prováděny nejsou.

Obrázek 6.1: Průběh činností konstruktoru



6.1 Parametry příkazové řádky

Na příkazové řádce lze ovlivnit pouze vstup či výstup konstruktoru. Toto rozhodnutí vede ke zjednodušení parametrů příkazové řádky. Je zavedeno několik skupin argumentů, které jsou dobře popsány v textu vypsaném programem při spuštění bez parametrů nebo s přepínačem `--help`. Je zde zavedeno několik přepínačů pro ovlivnění názvů výstupních souborů, které jsou bez použití přepínačů pojmenovány modifikací názvu vstupního souboru. Také zde lze změnit výchozí umístění šablon, uživatel tedy může podstrčit konstruktoru vlastní šablony.

6.2 Šablony

Již bylo zmíněno, že se pro generované zdrojové soubory používají šablony, aby nemusela být všechna data přímo součástí spustitelného souboru. Je tak možné mírně měnit implementaci generovaného parseru bez nutnosti překompilovat konstruktor.

Knihovna CTemplate zavádí šablonový systém, který řeší zpracování šablonových souborů, vyplnění proměnných částí daty připravenými v konstruktoru a uložení vyplněných šablon do souboru na disk. V konstruktoru je jen vybrán soubor se šablonou, předána data, která se mají do šablony vyplnit a určen název výstupního souboru. Knihovna se o ostatní již postará.

V šablonách jsou uváděny značky, které jsou následně nahrazeny proměnlivými daty. Aby nemusely být názvy značek uváděny v kódu konstruktoru jako textové řetězce, což by mohlo vést k těžko odhalitelným chybám při překlepu, zavádí CTemplate postup, kterým se ze šablony názvy značek extrahují a je vytvořen hlavičkový soubor obsahující konstanty, které následně mohou být v kódu použity.

SAG připravuje pouze jednu strukturu s daty vyplňovanými v šablonách pro všechny výstupní soubory, je tedy vhodné, aby konstanty nebyly rozlišeny podle šablon, ze kterých pocházejí, jak je to v CTemplate zavedeno. SAG tedy při výrobě hlavičkového souboru s názvy značek postupuje tak, že se nejdříve každá šablona zvlášť zkontroluje, zda neobsahuje chyby, pak se všechny šablony spojí do jednoho souboru a tento soubor je předán programu `make_tpl_varnames_h`, který vygeneruje jediný hlavičkový soubor `varnames.h`.

6.2.1 Umístění šablon

Aby bylo možné spouštět konstruktor odkudkoli, musí být soubory se šablonami referencovány absolutní cestou, která musí být v konstruktoru známa. Výchozí přístup

je takový, že musí být známa absolutní cesta již při překladu. Ta je dána jako makro překladači. Pokud toto makro není definované, je uměle vytvořena chyba při překladu. To není žádný problém na unixových systémech, kde může být před instalací konstruktor zkompilován a cesta správně vyplněna na budoucí umístění.

Na systémech MS Windows je spíše zvykem distribuovat již přeložený spustitelný soubor, do kterého již není možné pevné umístění po instalaci vpravit. Je tedy zavedeno další makro, které při překladu zpřístupní rozhraní `Windows Registry`, pomocí kterého je při každém spuštění konstruktoru absolutní cesta k šablonám zjišťována. Stačí tedy, když je během instalace vyplněna příslušná položka registrů.

6.3 Implementace

Konstruktor SAG je implementován objektově. Hlavní třídou je `Context`, který seskupuje všechny specializované komponenty programu. V celém programu je jen jedna instance této třídy, bylo by tedy možné ji implementovat jako singleton, pro lepší kontrolu nad tím, odkud je do kontextu přistupováno ale bylo rozhodnuto jinak. Instance třídy `Context` je vytvářena ve funkci `main` a je předávána referencí všude, kde je potřeba. Sám kontext příliš funkčnosti neimplementuje, spíše drží jednotlivé komponenty programu, které k sobě navzájem přistupují. Každá komponenta je ve třídě `Context` jako datová položka a má uloženou zpětnou referenci na kontext. Tato reference je předávána do konstruktoru, což nemusí být úplně bezpečné, protože v době konstrukce položek kontextu ještě není kontext plně inicializován (následující položky ještě zkonstruovány nejsou). Na to bylo při vývoji pamatováno a položky jsou poskládány tak, aby to nedělalo problémy. Komponenty kontextu jsou následující:

- Implicity - přidává implicitní pravidla a symboly gramatiky
- InputOutput - provádí vstupně-výstupní operace, řeší pojmenování souborů
- Parser - zpracovává vstupní soubor s gramatikou
- CommandlineProcessor - zpracovává parametry příkazové řádky
- DeclarationProcessor - zpracovává deklarace ve vstupním souboru
- RuleProcessor - zpracovává pravidla ve vstupním souboru
- AutomatonProcessor - zpracovává automat výsledného parseru
- TemplateProcessor - zpracovává data pro šablony
- Language - řeší odlišnosti cílových jazyků

Komponenta cílového jazyka `Language` je zvláštní tím, že je jako jediná dynamicky alokovaná a v průběhu programu se může změnit.

Dále obsahuje kontext instance důležitých tříd, které jsou hojně využívány po celém programu:

- ErrorReporter - tři samostatné instance
  - ladící výpisy
  - hlášení upozornění
  - hlášení chyb
- NamesTable - tabulka ukládající texty
- GrammarSymbols - tabulka ukládající symboly gramatiky
- RulesT - seznam pravidel gramatiky

Tímto rozdělením funkcionalit je docíleno dostatečného zapouzdření, protože i na vyšší úrovni jsou definována rozhraní pro přístup k datům jednotlivých částí programu.

6.3.1 Cílové jazyky

Cílových jazyků podporuje SAG hned několik a bylo by vhodné navrhnout design tak, aby bylo umožněno přidat další.

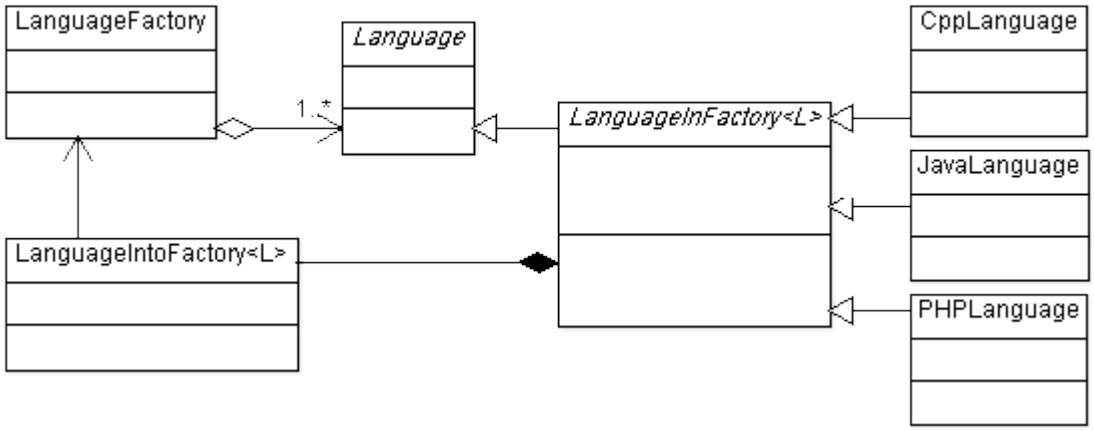
Třídy

Cílový jazyk je implementován ve třídě, jež je potomkem třídy `Language`. Ta specifikuje některé abstraktní metody, které musí třída cílového jazyka implementovat, dále pak metody, které mají nějakou výchozí implementaci, ale typicky je vhodné, aby je cílová třída reimplementovala také.

O vytvoření instance třídy cílového jazyka se stará singleton `LanguageFactory`. Aby vše fungovalo automaticky, je zaveden mechanismus, který třídu cílového jazyka ve factory zaregistruje. Cílová třída nedědí přímo od třídy `Language`, ale od šablonované třídy `LanguageInFactory`, která zajistí registraci třídy předané šablonovaným parametrem do `LanguageFactory`.

`LanguageInFactory` se šablonovým typem L má statickou položku typu LanguageIntoFactory taktéž parametrizovanou typem L. Ta při statické inicializaci zaregistruje třídu cílového jazyka L do `LanguageFactory`.

Obrázek 6.2: Třídy registrující cílový jazyk



Lexikální analýza

Zásadním problémem přidání dalšího jazyka je nutný zásah do analýzy vstupního souboru, ve kterém se typicky nachází části kódu v cílovém jazyce. Tomuto uživatelskému kódu sice konstruktor plně rozumět nemusí, musí ale poznat, kdy uživatelský kód končí, v případě sémantických akcí v něm dokonce musí rozpoznat zástupné značky. To je prováděno v lexikální analýze, která je tím do značné míry znepřehledněna.

Ve zdrojovém souboru lexikální analýzy jsou umístěna pravidla, podle kterých je následně vygenerován lexikální skener nástrojem `Flex`. Protože vstupní soubor s gramatikou, který chceme analyzovat, obsahuje mnoho, z pohledu lexeru různých, sekcí, jsou zde použity stavy lexeru (Exclusive Start States [12, s. 136]), kterými je působnost pravidel omezena. Tato funkcionalita je použita i pro odlišení lexikálních pravidel jednotlivých cílových jazyků. V lexeru jsou tedy pravidla, jak konkrétní cílový jazyk analyzovat, a správná pravidla se vyberou podle vybraného cílového jazyka. Proto je také nutné, aby byl cílový jazyk deklarován ihned na začátku vstupního souboru.

Jména stavů lexeru jsou ve vygenerovaném kódu jména maker, která mají přiřazenu konkrétní celočíselnou hodnotu. Tato hodnota musí být známa ve třídě cílového jazyka, aby bylo možné jednoduše přepínat stavy korespondující s vybraným cílovým jazykem. K tomu jsou určeny virtuální metody, které musí třída cílového jazyka implementovat. Aby bylo možné využít makra definovaná Flexem, jsou implementace těchto metod pro všechny cílové jazyky umístěny na konci souboru s pravidly lexeru.

Přidání nového cílového jazyka J

- Vytvořit třídu $JLanguage$ (potomek $LanguageInFactory < JLanguage >$)
  - vytvořit soubory $src/languages/JLanguage.(hpp|cpp)$
  - zaregistrovat je v $Makefile.am$ (sag_SRC) a ve $windows/Sag/Sag.vcxproj.*$
  - implementovat potřebné metody
  - zanést třídu do UML diagramů v $uml/sag.uml$
- Vytvořit šablony pro cílový jazyk
  - vytvořit soubory $templates/j\_parser.*.tpl$
  - zaregistrovat je v $Makefile.am$ (dist_templates_DATA)
- Upravit $grammar/sag.lex$
  - přidat jazykově specifické stavy lexikálního analyzátoru $J\_.*$
  - vytvořit (nebo využít existující) jazykově specifická lexikální pravidla
  - implementovat metody třídy $JLanguage$ vracející lexikální stavy

6.3.2 Znakové tokeny

V konstruktoru Bison je zavedena funkcionalita znakových tokenů [8, s. 53], která usnadňuje použití tokenů reprezentujících jeden znak ve vstupních datech. Ačkoli Bison tento požadavek nijak nevynucuje, zneužití této funkcionality může zmást jiného čtenáře.

Aby SAG vyhověl uživatelům, kteří jsou na tuto funkcionalitu zvyklí a také usnadnil práci s těmito jednoduchými tokeny, vyšetřil hodnoty, kterých může nabývat znak v cílovém jazyce, těmto speciálním tokenům. Pojmenované terminály a neterminály mají hodnoty vždy vyšší. Hodnota, od které jsou číslovány pojmenované terminály je určena metodou `Language::getMinNoncharacterTokenIndex`, která může být ve třídě cílového jazyka reimplementována. Výchozí hodnota je 256, ale například pro Javu je to 2^{16}, protože ta má dvoubytové znaky. V konstruktoru zatím dvoubytové znakové tokeny implementovány nebyly, i když by to neměl být problém.

Také s přenositelností by mohl být problém, obecně by se dalo říci, že nasazené řešení funguje jistě, pokud se na platformě, na které byl sestaven konstruktor, i na cílové platformě používá kódování ASCII. S menší námahou by se ale pomocí externích tabulek pro zjištění hodnoty znaku dala implementovat i přenositelná varianta.

Aktuálně je možno zadat všechny běžné znaky i některé speciální znaky pomocí zpětného lomítka.

6.3.3 Deklarace

Ve třídě `DeclarationProcessor` se zpracovávají deklarace z úvodní sekce vstupního souboru. Je zde struktura, která k danému názvu deklarace přiřazuje metodu procesoru, která danou deklaraci řeší. Tak lze jednoduše přidávat a měnit deklarace a zároveň je zpracování jednotlivých deklarací dobře oddělené.

Je třeba také podporovat jazykově specifické deklarace a také umožnit, aby se pro konkrétní cílový jazyk mohlo zpracování běžných deklarací změnit nebo zakázat. Třída `Language` proto také disponuje strukturou, která může obsahovat názvy deklarací a metody, které je řeší. Třída cílového jazyka danou strukturu případně naplní vlastními daty. Při zpracování deklarace se procesor nejdříve podívá, zda cílový jazyk danou deklaraci neumí zpracovat. Pokud ne, až potom je hledána metoda procesoru deklarací, která by to uměla. Pokud ani ta není nalezena, je nahlášena neznámá deklarace.

Data z deklarací jsou ukládána přímo v procesoru, nebo je rovnou provedena akce, která je deklarací požadována.

6.3.4 Pravidla

Gramatická pravidla jsou během parsování vstupního souboru zpracována ve třídě `RuleProcessor`. Ten překontroluje sémantické akce, nastaví reference mezi pravidly a neterminálem na levé straně a přidá pravidla do seznamu gramatických pravidel v kontextu.

`RuleProcessor` také zpracovává akce vnořené v pravé straně pravidla. Taková akce se provádí v parseru dříve, než je celé pravidlo zanalyzováno. Prakticky je třeba vnitřní akce přepracovat na obyčejné, aby s nimi už ani algoritmus na vytvoření automatu nemusel počítat. Procesor pravidel vytvoří pomocný neterminál a pravidlo, které má tento neterminál na levé straně. Pravá strana nového pravidla je prázdná. Obsahuje pouze koncovou sémantickou akci, což je právě uživatelova vnitřní akce. Pokud jsou zde použity značky, jejich indexy musí být posunuty do záporných hodnot, aby referencovaly požadované symboly nadřazeného pravidla. Do původního pravidla je pak namísto vnořené akce vložen nově vytvořený pomocný neterminál.

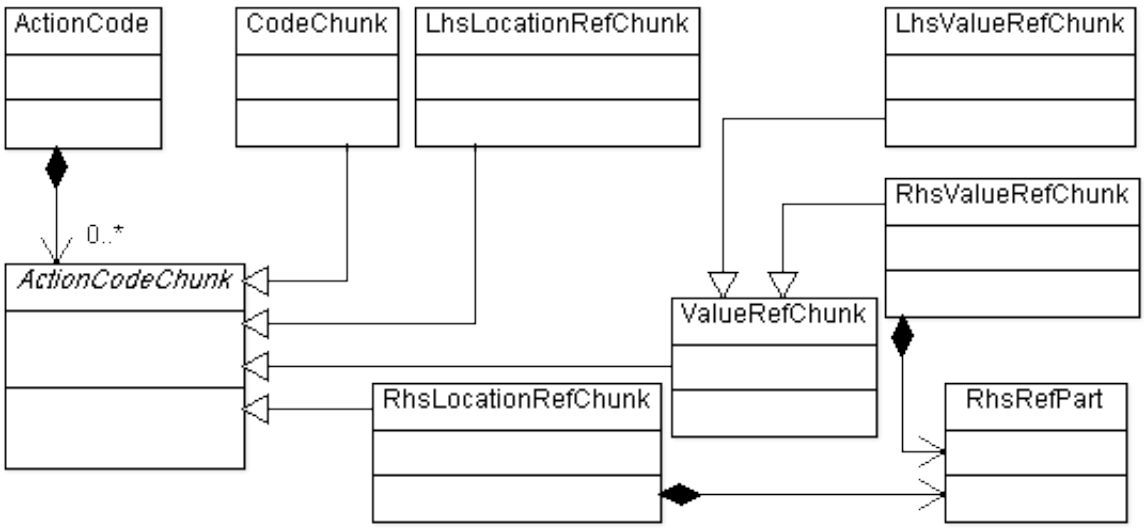
Po zpracování vstupního souboru je s pravidly dále nakládáno při vytváření automatu a konstrukce jeho stavů.

6.3.5 Sémantické akce

Sémantická akce je lexikální analýzou rozdělena na kusy, následně syntakticky zanalyzována a jednotlivé syntaktické části přidány do seznamu. Syntaktické části jsou kusy kódu v cílovém jazyce nebo značky referencující sémantickou hodnotu či pozici některého symbolu pravidla. Sémantická akce je reprezentována jako posloupnost abstraktních kusů, potomků třídy `ActionCodeChunk`. Od ní dědí přímo `CodeChunk`, `LhsLocationRefChunk a RhsLocationRefChunk`, reprezentující obecný kus kódu cílového jazyka, referenci pozice levého neterminálu a referenci pozice konkrétního symbolu pravé strany. Přes `ValueRefChunk` od ní také dědí `LhsValueRefChunk a RhsValueRefChunk`, referencující sémantické hodnoty konkrétních symbolů pravidla. Součástí reference na symbol (pozici nebo jeho sémantickou hodnotu) pravé strany pravidla je položka typu `RhsRefPart`.

Tato bohatá struktura dědičnosti je zavedena především kvůli sdílení kódu, který implementuje vlastnosti jednotlivých referencí. Přirozeně by byla i reference na pravou stranu pravidla jedním z předků, tím by ale vznikl tzv. `Diamond problem` vícenásobné dědičnosti, protože `RhsValueRefChunk` by dvakrát dědil od `ActionCodeChunk`. To lze v C++ řešit virtuální dědičností, což bylo i úspěšně vyzkoušeno. Virtuální dědičnost je ale často považována za kontroverzní řešení, proto byla dána přednost aktuálnímu řešení pomocí privátní položky.

Obrázek 6.3: Třídy implementující části sémantické akce



Části sémantické akce se samy starají o kontrolu svých parametrů, včetně typu sémantické hodnoty ve třídě `ValueRefChunk` a hodnoty indexu ve třídě `RhsRefPart`. Jednotlivé části se také starají o vyplnění dat, která mají být použita v šablonách.

6.3.6 Vytvoření automatu

Automat je konstruován po dokončení analýzy vstupního souboru, kdy jsou již všechny symboly gramatiky i pravidla známy. Ve třídě `AutomatonProcessor` je připravena gramatika pro samotnou konstrukci automatu, ten je pak zkonstruován v konstruktoru třídy `Automaton`, která je dynamicky vytvářena v procesoru a následně vrácena k dalšímu použití procesorem šablon.

Příprava gramatiky spočívá v její redukci, unikátním očíslování neterminálů a pravidel gramatiky a vytvoření množin `FirstSet` jednotlivých neterminálů.

Na obrázku 6.4 je znázorněno zapojení tříd, které implementují automat. Ten má po zkonstruování několik stavů, v každém stavu jsou uloženy položky a může také obsahovat konflikty. Položka stavu automatu je implementována dvojicí tříd `DottedRule` a `FirstSet`, což reprezentuje otečkované pravidlo a množinu terminálů, které ho mohou následovat (výhled). Otečkované pravidlo drží pozici tečky a referenci na obyčejné pravidlo gramatiky, které má neterminál na levé straně, může mít několik gramatických symbolů na pravé straně a sémantickou akci, jejíž implementace je zobrazena na obrázku 6.3.

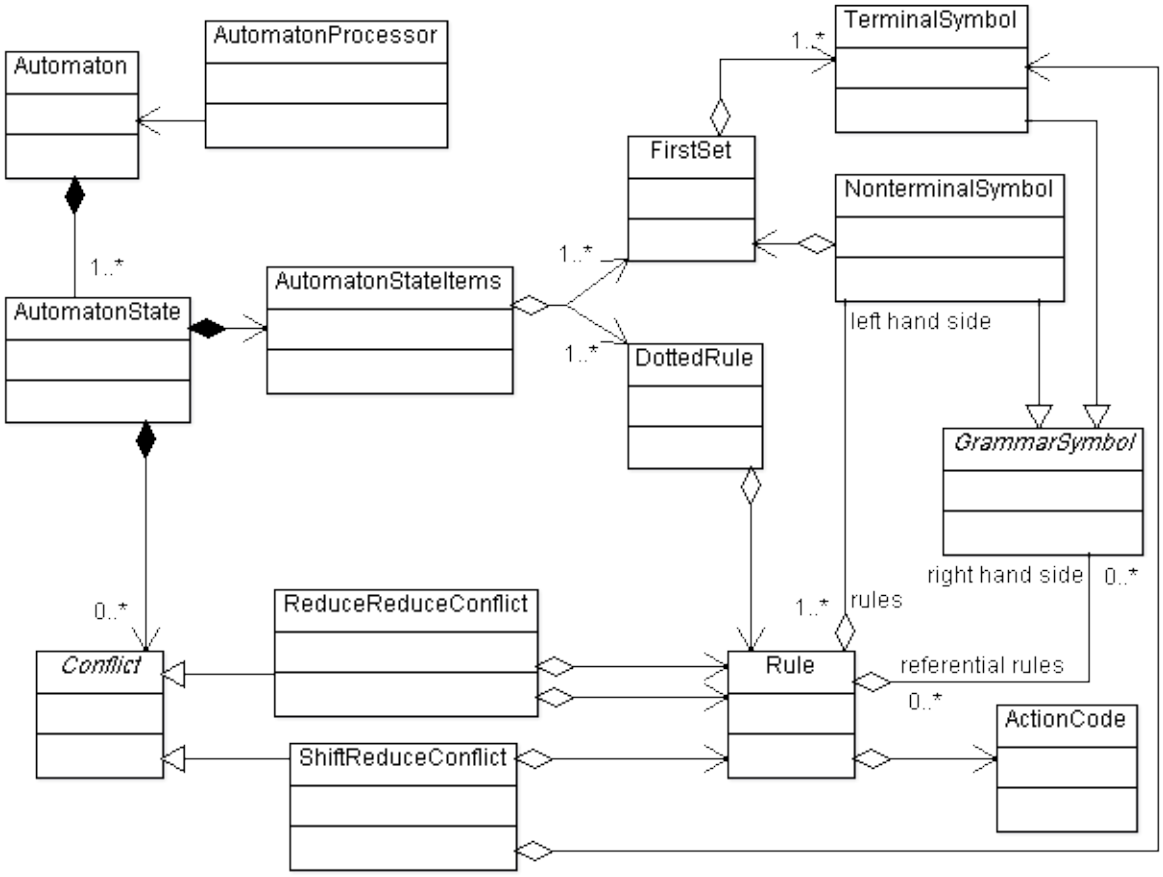
Redukce gramatiky

Uživatel může zadat jakoukoli bezkontextovou gramatiku, tedy i takovou, která obsahuje zbytečná pravidla a symboly. To je vhodné detekovat, aby se uživatel dozvěděl, že se něco ve výsledném automatu vůbec nepoužije. Důležitější je ale fakt, že použité algoritmy mohou pracovat správně pouze na redukované gramatice.

Redukce gramatiky pracuje ve dvou krocích, které není možné obrátit. V prvním kroku dojde k nalezení neterminálů, které negenerují terminální slovo. To je typicky způsobeno nekonečným cyklem použitých pravidel, nebo tím, že uživatel zapomněl pravidla pro některé neterminály uvést.

V druhém kroku se odstraní nedosažitelné symboly. Dosažitelné symboly jsou nalezeny procházením pravidel od startovního neterminálu, symboly, které nejsou dosažitelné, jsou označeny jako nedosažitelné.

Obrázek 6.4: Třídy implementující automat



Nepotřebné symboly gramatiky jsou ponechány v původním kontejneru, jen jsou označeny. Nepotřebná pravidla jsou ze seznamu gramatických pravidel v kontextu přesunuta do seznamu nepotřebných pravidel uloženém v procesoru automatu.

Očíslování symbolů a pravidel gramatiky

Neterminály jsou číslovány společně s ostatními symboly gramatiky, jejich hodnoty navazují na hodnoty terminálů. Pravidla jsou číslována zvlášť s tím, že nejdříve jsou očíslována potřebná pravidla a pak nepotřebná, která byla z gramatiky odstraněna.

Přiřazené hodnoty jsou v parseru použity jako indexy do struktur, kde se podle symbolů nebo pravidel vyhledává.

Konstrukce množin FIRST

Před konstrukcí automatu jsou také vytvořeny množiny `FirstSet` jednotlivých neterminálů, které slouží k určení terminálů, kterými může začínat slovo generované

daným neterminálem. Tato třída je také použita při konstrukci automatu pro reprezentaci terminálů, kterými může začínat terminální slovo generované celou posloupností symbolů. Třída `FirstSet` kromě množiny terminálů také obsahuje informaci, zda daná posloupnost symbolů může generovat prázdné slovo.

Ke konstrukci množin FIRST jednotlivých neterminálů je použita unikátní fronta pravidel, což je datová FIFO[1] struktura, která navíc hlídá unikátnost svých prvků, tedy nepřidá prvek, který již obsahuje. Využijí se zde také vazby vedoucí od symbolu k pravidlům, u nichž se vyskytuje daný symbol na pravé straně.

Algoritmus začíná s prázdnými množinami FIRST všech neterminálů s tím, že jsou postupně doplněny. Po skončení algoritmu jsou množiny všech neterminálů vyplněny správně. Do fronty jsou přidána všechna pravidla gramatiky a dokud není prázdná, opakují se následující kroky:

- Odeber jedno pravidlo z fronty
- Přidej do množiny FIRST neterminálu na levé straně pravidla následující
  - Pokud je aktuální symbol pravé strany pravidla terminál, přidej ho
  - Pokud jde o neterminál, přidej symboly z jeho množiny FIRST, obsahuje-li prázdné slovo, pokračuj dalším symbolem pravé strany
  - Pokud jsme se dostali až na konec pravidla, přidej prázdné slovo
- Pokud byla množina FIRST levého neterminálu změněna, přidej do fronty všechna pravidla obsahující na pravé straně zpracovávaný neterminál

6.3.7 Konstrukce stavů

Konstrukce stavů automatu je provedena v konstruktoru třídy `Automaton`. Ten má přístup k privátním položkám třídy `AutomatonState`, která jen drží data pro položky stavu, přechody do dalších stavů, redukce pravidel a konflikty. Data jsou ale vyplněna metodami třídy `Automaton`, protože jejich smysl je spíše v kontextu celého automatu, nikoli jen jednoho stavu.

Pro implementaci konstrukce stavů LALR(1) parseru byl použit algoritmus spojování stavů LR(1) parseru během jejich konstrukce. Ten byl již překonán efektivnějším algoritmem od DeRemera [7], jehož implementace by ale byla složitější.

Uzávěr množiny otečkovaných pravidel, rozšířený o dopočítávání terminálů, které mohou za pravidlem následovat, je implementován ve třídě `AutomatonStateItems`

[1]FIFO - First In First Out, tedy prvek, který přijde první do struktury, jde první ven

a prováděn ihned při přidání pravidla s výhledem. Zde je také implementováno spojování dvou množin položek, které obsahují stejná otečkovaná pravidla s potenciálně různými výhledy. Dále vytvoření stavu, který vznikne přechodem přes daný symbol gramatiky, a také porovnávání dvou množin položek podle jádra, tedy pouze podle otečkovaných pravidel (dvě instance `AutomatonStateItems` jsou shodné, pokud obsahují stejná otečkovaná pravidla).

Algoritmus začíná zkonstruováním stavu obsahující startovní pravidlo. Ostatní stavy jsou zkonstruovány rekurzivně, jako přechody přes všechny symboly, které se vyskytují za tečkou nějakého pravidla. Po tom, co jsou všechny stavy zkonstruovány, jsou zpracovány v jednotlivých stavech pravidla, která se mohou redukovat, a vyřešeny případné konflikty.

Jeden rekurzivní krok konstrukce stavu ze zadaných položek (otečkovaných pravidel s výhledem) spočívá v nalezení stavu podle jeho otečkovaných pravidel. Pokud takový již existuje, jeho výhledy se spojí s konstruovaným, pokud neexistuje, dojde k vytvoření nového stavu. Došlo-li k nějaké změně, jsou nalezeny všechny symboly gramatiky, které se nacházejí za tečkou nějakého pravidla uvnitř stavu, a pro každý symbol je rekurzivně zkonstruován (nebo alespoň nalezen) stav, který obsahuje položky vytvořené přechodem podle daného symbolu. Výsledkem rekurzivního volání je vždy stav (nově vytvořený nebo existující), který obsahuje dané položky. Ten je pak uložen jako výsledek přechodu s daným symbolem do konstruovaného stavu.

Koncový stav, tedy stav, ve kterém má být slovo přijato, se nekonstruuje. Automat výsledného parseru tedy takový stav vůbec neobsahuje, přechod do tohoto stavu je ve výsledném parseru nahrazen akcí `Accept` (bez cílového stavu).

Po konstrukci všech stavů je ještě potřeba v každém stavu určit, která pravidla se budou redukovat a jaký výhled k tomu bude požadován. Mezi všemi položkami stavu se najdou ty, jejichž otečkovaná pravidla mají tečku na konci. Ty je možné redukovat se všemi terminály z výhledu položky. Pokud je možné se stejným terminálem ve výhledu redukovat jiné pravidlo, vzniká Reduce/Reduce konflikt mezi těmito pravidly, pokud je možné s tímto terminálem přejít do jiného stavu automatu, vzniká Shift/Reduce konflikt mezi daným terminálem a pravidlem.

6.3.8 Řešení konfliktů

SAG řeší konflikty automaticky, ale pokud nejsou vyřešeny pomocí přednosti a asociativity operátorů, vždy vypíše upozornění, že ke konfliktu došlo.

Na obrázku 6.4, který jsme již viděli, je zachycena i struktura tříd, které reprezentují existující konflikt. Od abstraktní třídy `Conflict`, která je ukládána do seznamu

ve stavu automatu, dědí třídy `ReduceReduceConflict a ShiftReduceConflict`. Ty dále drží reference na konfliktní entity a v konstruktoru se snaží zjistit, jak konflikt vyřešit.

Reduce/Reduce konflikt

Konflikt mezi dvěma pravidly nastane, existují-li v jednom stavu dvě položky, jejichž otečkovaná pravidla mají tečku na konci a ve výhledu shodný terminál. Parser tedy neví, při daném výhledu, jaké pravidlo redukovat.

Konflikt je vyřešen odstraněním redukce pravidla, které je uvedeno ve vstupním souboru později. SAG tento konflikt nikdy nevyřeší tiše, je tedy vždy vypsáno upozornění.

Shift/Reduce konflikt

Konflikt mezi přečtením dalšího tokenu a redukcí pravidla nastane, existuje-li v jednom stavu možnost pro jeden token ve výhledu provést akci Shift i redukovat nějaké pravidlo. Parser tedy neví, při daném výhledu, jakou akci provést.

SAG může vyřešit tento konflikt tiše, pokud má terminál ve výhledu i redukované pravidlo nastavenu přednost operátorů. Ta může vyvolat akci Shift, Reduce nebo Error. V takovém případě není upozornění vypsáno.

Pokud není konflikt vyřešen předností operátorů, dostane přednost akce Shift (kvůli udržení kompatibility s Bisonem), a upozornění je vypsáno.

Při použití jedné z akcí je druhá z nich ze seznamu akcí ve stavu odstraněna, parser ji tedy pro daný výhled neprovede. Pokud je konflikt vyřešen provedením akce Error, parser musí při daném výhledu vyvolat syntaktickou chybu. To je docíleno tím, že jsou obě akce ve stavu zrušeny, parser tedy bude mít ve svých tabulkách prázdné místo a vyvolá chybu.

Vícenásobný konflikt

Vícenásobný konflikt, tedy více možností (než dvě) jakou akci při daném terminálu ve výhledu provést, je řešen postupně. Nejdříve nastane pro daný výhled konflikt mezi dvěma entitami, ten je vyřešen podle předchozích pravidel, a další entita (pravidlo) vyvolá další konflikt s tím, co zbylo.

V případě, že je mezi dvěma entitami vyřešen Shift/Reduce konflikt akcí Error, konfliktní přechod i redukce pravidla jsou ze seznamu akcí ve stavu odstraněny, další redukce se stejným výhledem tedy konflikt nevyvolá.

Je proto ještě v každém stavu ukládána množina terminálů, které ve výhledu musí způsobit syntaktickou chybu. Do této množiny je terminál přidán pouze, pokud je vyřešen Shift/Reduce konflikt, ve kterém figuroval, akcí Error. Další případné pravidlo, které by se s tímto terminálem ve výhledu mělo redukovat, pak již přidáno do seznamu redukcí není.

7. Měření

Měření bylo provedeno na platformě GNU/Linux, na stroji s procesorem Intel Core i3 s frekvencí 2,26 GHz. Výsledky byly porovnávány s konstruktorem Bison, verze 2.4, s rozšířením pro PHP [3].

Jako příklad gramatiky byl ve všech případech použit jednoduchý jazyk matematických výrazů, kde je na každém řádku jeden matematický výraz a v sémantických akcích se provádí jeho výpočet. Výsledek je pak vypsán na standardní výstup.

Byl vytvořen jednoúčelový shell skript, který automaticky pro zadané konstruktory a cílové jazyky vygeneruje parser a ten pak pro vstupy různých velikostí spouští. Měřena je reálná doba, po kterou běží konstruktor generující parser, pro každý cílový jazyk zvlášť. Dále je měřena doba strávená v metodě `parse` vygenerovaného parseru, tedy samotné parsování vstupu, taktéž pro každý cílový jazyk zvlášť. Parser je spouštěn na vstupech různých velikostí, aby byla zřejmá závislost na velikosti vstupu.

Takto popsaný postup lze spustit vícekrát, ze získaných výsledků je následně jiným skriptem vytvořen aritmetický průměr, a ten vykreslen pomocí nástroje `gnuplot` do grafů.

7.1 Rychlost vygenerování parseru

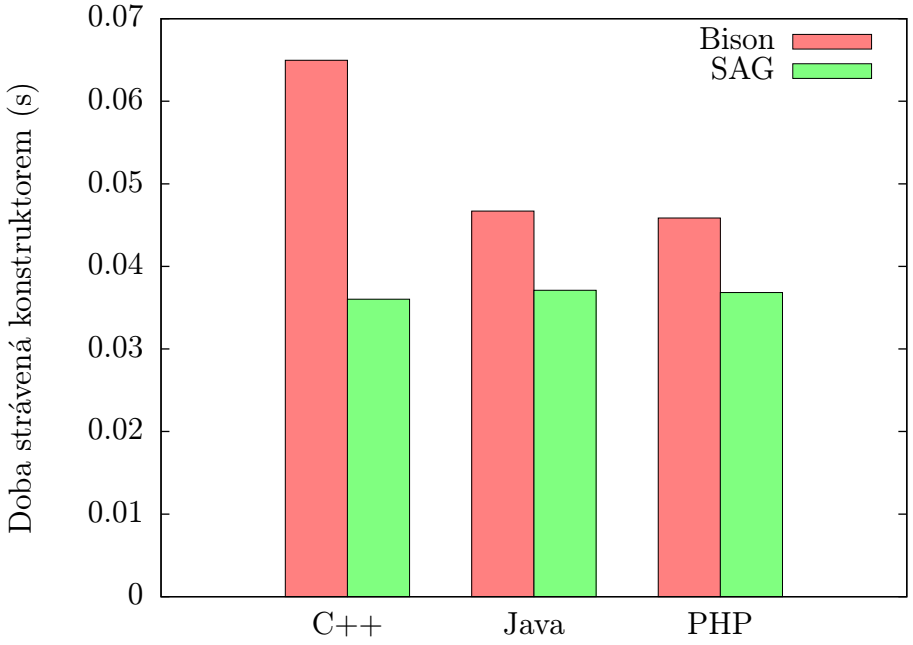
V první řadě bylo měřeno, jak rychle vygeneruje konstruktor parser. Běžným způsobem je spuštěn konstruktor na dané gramatice pro určitý cílový jazyk, aniž by do něj bylo jakkoli zasahováno. Pomocí vestavěného příkazu shellu `time` je měřeno, kolik času tento běh konstruktoru zabere. Do grafu (Obrázek 7.1) je poté zanesena průměrná reálná doba pro každý konstruktor a cílový jazyk.

Ve všech případech je pro takto jednoduchou gramatiku parser vygenerován za méně než desetinu sekundy a oba konstruktory mají srovnatelnou rychlost. Pro jazyk C++ je Bison mírně pomalejší, SAG je pro všechny cílové jazyky stejně rychlý.

7.2 Výkon parserů

Součástí souboru s gramatikou je vždy i kód, který inicializuje parser a měří, jak dlouho běžela metoda `parse`. Tuto informaci pak vypíše program na standardní chybový výstup, který je odchytáván do souboru. Pro každý konstruktor, cílový jazyk

Obrázek 7.1: Rychlost vygenerování parseru



a velikost vstupu se spočítá aritmetický průměr všech zaznamenaných běhů. Pro jednotlivé jazyky si poté můžeme prohlédnout srovnání výkonu parserů od různých konstruktorů, v závislosti na velikosti vstupu.

Oba parsery v C++ řešily problém srovnatelně rychle. Při použitém překladači gcc verze 4.6.2 a vypnuté optimalizaci (Obrázek 7.2) byl Bison mírně pomalejší. Při optimalizacích úrovně 1 a 2 byl pomalejší SAG, při nejvyšší optimalizaci úrovně 3 byl opět výsledek velmi vyrovnaný.

U parserů v jazyce Java je zajímavé, že závislost na velikosti vstupních dat není ani v jednom případě lineární (Obrázek 7.3). Tedy alespoň na vstupech menších než 100 KiB. Mohlo by to být způsobeno JIT[1] kompilací, která v Javě za běhu postupně překládá a optimalizuje kód, který je potřeba. Na menších datech tedy JIT kompilace spíše zdržuje, na větších už se vyplatí a parser běží v závislosti na velikosti vstupu rychleji. I v tomto případě byl výsledek porovnání téměř nerozhodný, SAG strávil v metodě `parse`, pro vstup velikosti od 20 KiB do 160 KiB, přibližně o 30 ms méně, nezávisle na velikosti vstupu. I tento rozdíl je téměř zanedbatelný.

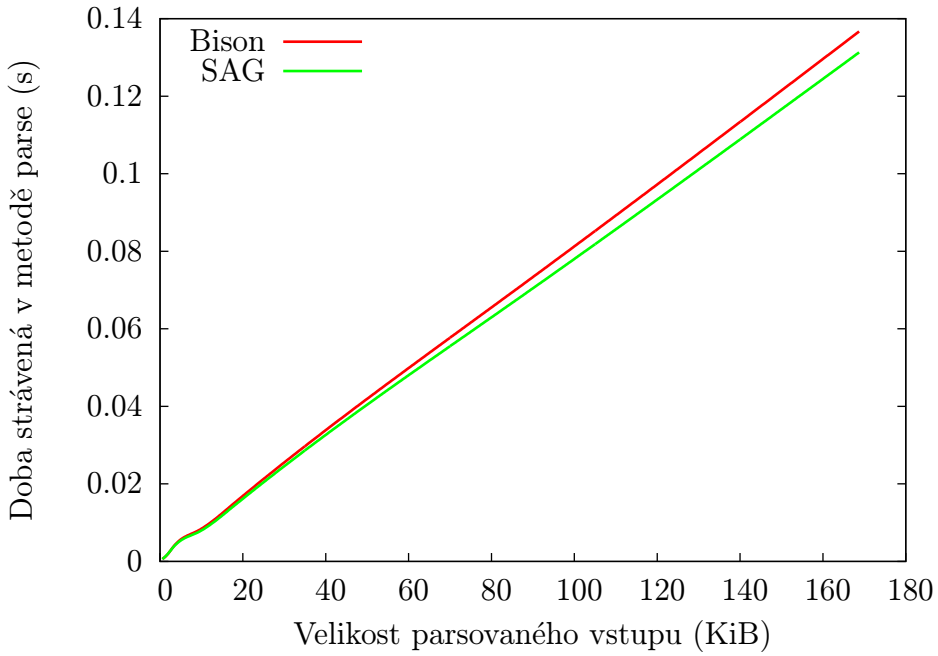
V případě parserů v PHP je situace odlišná (Obrázek 7.4). Při spuštění přes příkazovou řádku na PHP 5.3.8, Zend Engine v2.3.0 byl parser generovaný Bisonem cca 2,5krát rychlejší, nicméně stále jsou oba parsery lineární v souvislosti s velikostí vstupu. Důvodů, kvůli kterým by mohlo ke zpomalení dojít, může být hned několik.

[1]JIT - Just In Time, tedy právě v čase, kdy je potřeba.

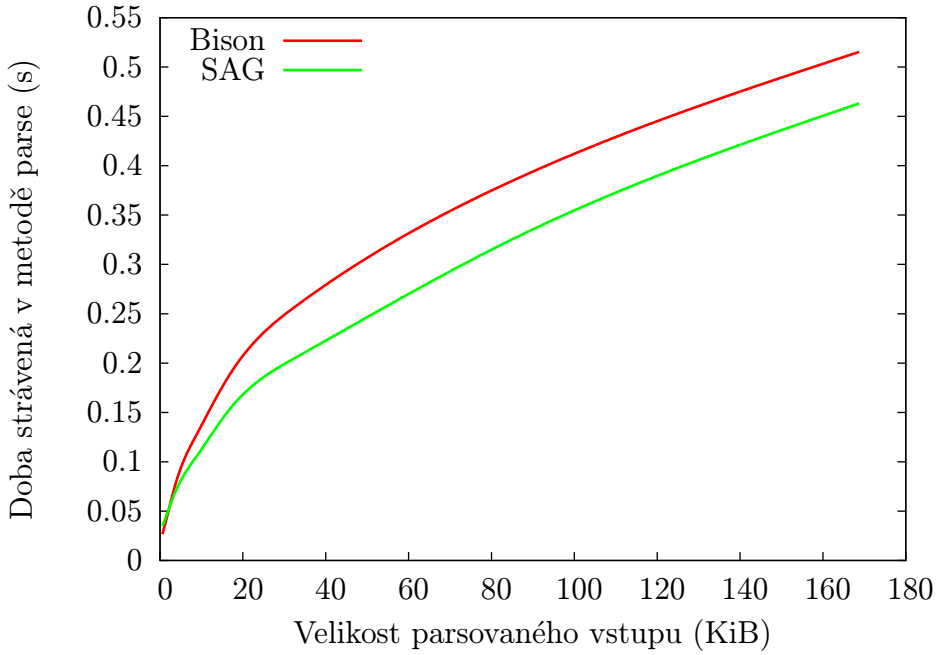
Například vyšší režie při použití objektů v PHP, pomalejší přístup do řídkého pole, ve kterém jsou uloženy tabulky parseru, nebo pomalejší volání sémantických akcí, které jsou naprogramovány jako anonymní funkce.

Vzhledem k tomu, že parsery generované konstruktorem SAG nejsou primárně optimalizovány na rychlost, jsou výsledky měření uspokojující, v případě Javy dokonce vynikající.

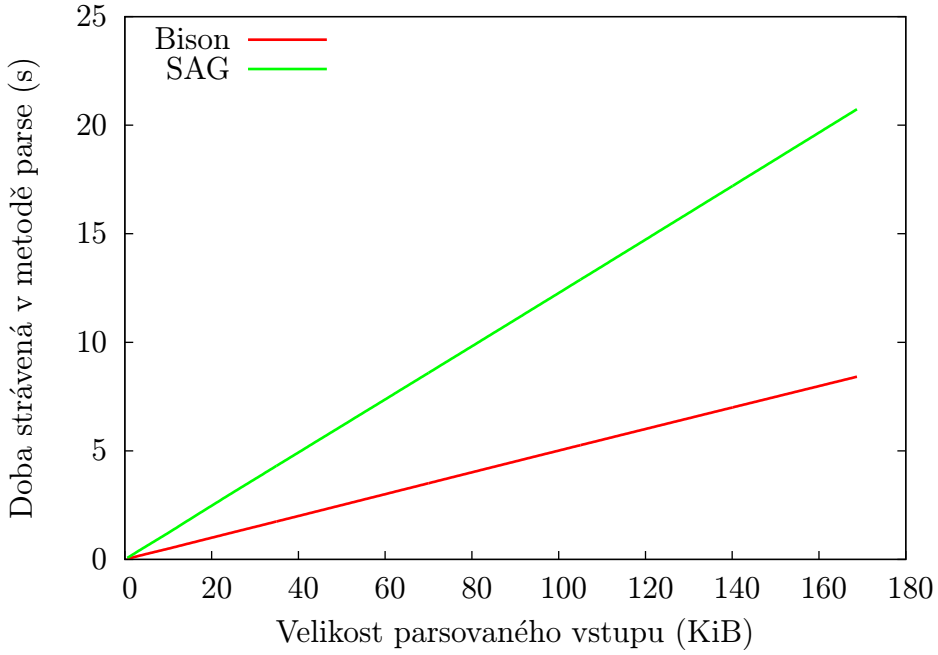
Obrázek 7.2: Výkon parseru v jazyce C++



Obrázek 7.3: Výkon parseru v jazyce Java



Obrázek 7.4: Výkon parseru v jazyce PHP



8. Závěr

Cílem práce bylo vyvinout nástroj pro generování LALR(1) parserů, což se zajisté podařilo. SAG je moderně implementován, podporuje tři cílové jazyky a je připraven na rozšíření o další. Rozhraní definovaná v generovaných parserech jsou přehledná a napříč cílovými jazyky podobná, díky využití především objektově orientovaného přístupu.

Ačkoli nebyl kladen hlavní důraz na rychlost parserů, je jejich efektivita dostatečná pro běžné použití a srovnatelná s běžně používanými nástroji. Velkým přínosem by byla implementace perfektního hashování tabulek parseru, což by vedlo ke zjednodušení uložení dat uvnitř parseru a nejspíš i ke zrychlení přístupu k nim.

Knihovna použitá pro šablonování zdrojových kódů parserů byla v průběhu vývoje zhodnocena jako slabý článek, systém nedisponuje dostatečnou pružností, a tak je výroba dat pro šablonový systém složitější, než bylo původně v plánu. Aktuální stav je ale stabilní a k výměně knihovny zatím dostatečný důvod nebyl. Do budoucna by to ale mohla být oblast vylepšení.

Nová funkcionalita automatického odchytávání výjimek je zajímavá, funkční a v některých případech může být velmi užitečná. V 5.4.3 je nastíněno několik možných vylepšení, která by se v této oblasti mohla v budoucnu implementovat.

Literatura

[1] ANTLR - About The ANTLR Parser Generator.
`http://www.antlr.org/about.html`

[2] Bison - GNU parser generator. `http://www.gnu.org/software/bison/`

[3] Bison-PHP - Extension to bison to generate PHP code.
`https://github.com/scfc/bison-php`

[4] C++ language reference. `http://cppreference.com`

[5] CTemplate - Powerful but simple template language for C++.
`http://code.google.com/p/ctemplate/`

[6] DeRemer Frank: *Practical translators for LR(k) languages.*
Ph.D. dissertation, Dep. Electrical Engineering, Massachusetts Institute of Technology, Cambridge, 1969.

[7] DeRemer Frank, Pennello Thomas:
Efficient Computation of LALR(1) Look-Ahead Sets.
Transactions on Programming Languages and Systems, vol. 4, no. 4: 615–649, 1982.

[8] Donnelly Charles, Richard Stallman:
The Bison Manual: Using The YACC-compatible Parser Generator.
Boston, MA: GNU Press, 2003. ISBN: 1-882114-23-X

[9] Grune Dick, Jacobs Ceriel: *Parsing techniques: a practical guide.*
New York: Springer, 2008. ISBN: 978-0-387-20248-8

[10] Knuth Donald: *On the translation of languages from left to right.*
Information and Control 8: 607-639, 1965.

[11] LaLonde Wilf: *Constructing LR Parsers for Regular Right Part Grammars.*
Acta Informatica 11: 731-741, 1977.

[12] Levine John: *flex & bison: Unix Text Processing Tools.*
O'Reilly, 2009. ISBN: 978-0-596-15597-1

[13] LIME - Parser generator for PHP.
`http://sourceforge.net/projects/lime-php/`

A. Gramatika vstupního souboru

Gramatika vstupního souboru konstruktoru SAG je zde uvedena ve formě gramatických pravidel tak, jak je zpracovává SAG. Vygenerovaný parser pak slouží jako parser konstruktoru. Pro zachování přehlednosti jsou vynechány sémantické akce. Symboly psané velkými písmeny jsou terminály (tokeny z lexikálního analyzátoru). Pravidla lexikální analýzy zde přímo uvedena nejsou pro jejich složitost, následuje alespoň popis všech terminálů.

- ANGLE_OPEN, ANGLE_CLOSE - otevírací a uzavírací úhlová závorka
- CURLY_OPEN, CURLY_CLOSE - otevírací a uzavírací složená závorka
- PERCENT(%), COLON(:), SEMICOLON(;), VERTICAL_BAR(|), AT_SIGN(@), DOLLAR($), EXCLAMATION(!) - znaky uvedené v závorce
- SECTION_DELIMITER - dvojité procento (%%), oddělovač sekcí
- PRECEDENCE - %prec, v sekci pravidel
- LANGUAGE - %language, pokud jde o první deklaraci na začátku souboru
- CODE - část kódu uvnitř složených závorek
- NAME - název, jméno, identifikátor (typicky $[a-zA-Z\_][a-zA-Z\_0-9]*$)
- CHAR - znakový token, typicky jeden znak uvnitř apostrofů
- NUMBER - celé číslo (posloupnost číslic s případným znaménkem mínus)
- TERMINAL, NONTERMINAL - název terminálu, resp. neterminálu. V sekci pravidel se identifikátory již v lexeru vyhledávají v tabulce gramatických symbolů a do parseru je token přímo předáván jako terminál či neterminál.

```
all: declarations_section SECTION_DELIMITER
        rules_section opt_user_section ;


/* Declarations section */

declarations_section: /* empty */
        | declarations_section declaration ;

declaration: LANGUAGE declaration_params
        | PERCENT NAME declaration_params ;

declaration_params: /* empty */
        | declaration_params declaration_param ;

declaration_param: NAME
        | CHAR
        | ANGLE_OPEN NAME ANGLE_CLOSE
        | CURLY_OPEN declaration_code CURLY_CLOSE  ;

declaration_code: /* empty */
        | declaration_code CODE ;


/* User section */

opt_user_section: /* empty */
        | SECTION_DELIMITER user_section ;

user_section: /* empty */
        | user_section CODE ;
```

```
/* Rules section */

rules_section: rule_set
        | rules_section rule_set ;

rule_set: NONTERMINAL COLON rules SEMICOLON
        | TERMINAL COLON rules SEMICOLON /* Error */ ;

rules: rule
        | rules VERTICAL_BAR rule ;

rule: rule_symbols opt_action
        | rule_symbols PRECEDENCE TERMINAL opt_action ;

rule_symbols: /* empty */
        | rule_symbols opt_action grammar_symbol ;

grammar_symbol: TERMINAL
        | NONTERMINAL ;

opt_action: /* empty */
        | CURLY_OPEN action_code CURLY_CLOSE ;

action_code: /* empty */
        | action_code CODE
        // Value reference
        | action_code DOLLAR opt_value_type DOLLAR
        | action_code DOLLAR opt_value_type EXCLAMATION
        | action_code DOLLAR opt_value_type NUMBER
        // Line reference
        | action_code AT_SIGN DOLLAR
        | action_code AT_SIGN NUMBER ;

opt_value_type: /* empty */
        | ANGLE_OPEN NAME ANGLE_CLOSE ;
```

B. Obsah přiloženého CD

K této práci je přiloženo médium, na kterém jsou uloženy veškeré materiály k práci a k projektu SAG. Obsahuje dokumentaci včetně textu práce v elektronické podobě, instalační balíčky, zdrojové kódy projektu, externí knihovny a příklady. Obsahuje také skripty, kterými je možno zrekonstruovat měření popsaná v kapitole 7.

- `docs` ..... Dokumenty k projektu
  - `User-manual.pdf` ..... Uživatelský manuál
  - `Diploma-thesis.pdf` ..... Diplomová práce (tento text)
  - `Doxygen-documentation.zip` ..... Referenční html dokumentace
- `install` ..... Instalační balíčky konstruktoru SAG
  - `gnu` ..... Distribuční balíček pro Unix/Linux
  - `windows` ..... Instalační program pro platformu MS Windows
- `devel` ..... Pro vývojáře projektu SAG
  - `grammar` ..... Zdrojové gramatiky pro lexer i parser
  - `src` ..... Zdrojové kódy konstruktoru
  - `templates` ..... Šablony zdrojových kódů parserů
  - `unittest` ..... Testování jednotek konstruktoru
  - `uml` ..... UML diagramy pro nástroj ArgoUML
  - `windows` ..... Projekt pro Visual Studio 2012 a Inno Setup
- `examples` ..... Příklady gramatik
  - `negative` ..... Příklady záměrně obsahující chyby
  - `positive` ..... Fungující příklady
    - `cpp` ..... Příklady pro cílový jazyk C++
      - `calc.yy` ..... Kalkulátor s deklarací %union
      - `calc-stype.yy` ..... Kalkulátor s deklarací %stype
    - `java` ..... Příklady pro cílový jazyk Java
    - `php` ..... Příklady pro cílový jazyk PHP
    - `incomplete` ..... Příklady generující nekompletní parser
      - `minimal.yy` ..... Minimální gramatika
      - `students.yy` ..... Jednoduchá gramatika
      - `python.ypp` ..... Praktická gramatika programovacího jazyka
      - `noslr.ypp` ..... LALR(1) gramatika, která není SLR(1)
      - `intermediate.ypp` ..... Sémantické akce uvnitř pravé strany
      - `useless.yy` ..... Nepotřebná pravidla
- `benchmark` ..... Měření výkonu a generování grafů
  - `input` ..... Vstupní data
  - `bison` ..... Gramatiky pro konstruktor Bison
  - `sag` ..... Gramatiky pro konstruktor SAG
  - `run.sh` ..... Spuštění konstruktoru a parserů
  - `plot.sh` ..... Zpracování výsledků měření
- `libs` ..... Externí knihovny CTemplate a Boost