VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

POKROČILÉ GENEROVÁNÍ SYNTAKTICKÝCH ANALYZÁTORŮ

ADVANCED PARSER GENERATORS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **DANIEL HAVRANEK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. ZBYNĚK KŘIVKA, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Havranek Daniel**
Program: Informační technologie
Název: **Pokročilé generování syntaktických analyzátorů**
Advanced Parser Generators
Kategorie: Překladače
Zadání:

1. Seznamte se s existujícími generátory syntaktických analyzátorů založených na pokročilých metodách syntaktické analýzy (např. Generalized LR, LL(*) či PEG).
2. Proveďte porovnání těchto generátorů z hlediska schopností i omezení.
3. Dle konzultací s vedoucím vyberte dva různé generátory (jeden může být klasický, např. Bison) a implementujte pro ně analyzátor pro programovací jazyk zvolený tak, aby demonstroval silné i slabé stránky zvolené metody pokročilé syntaktické analýzy. Vše demonstrujte na dostatečném počtu testů.
4. Na základě výstupů z bodu 3 proveďte srovnání generování v obou generátorech i obou vygenerovaných analyzátorů (např. ohledně časové efektivity či efektivitě zápisu analyzovaného jazyka). Výsledky zhodnoťte a pokuste se navrhnout řešení některých omezení či nevýhod použitých generátorů.

Literatura:

- Economopoulos, G. R.: Generalised LR parsing algorithms. Thesis. Department of Computer Science Royal Holloway, University of London, 2006.
- Parr, T., Fisher, K.: LL(*): The foundation of the ANTLR parser generator. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, New York, USA, 2011, pp. 425-436. ISBN 978-1-4503-0663-8.
- *Elkhound: A GLR Parser Generator* [online]. Dostupné z: http://scottmcpeak.com/elkhound/ [cit. 2021-10-08].

Pro udělení zápočtu za první semestr je požadováno:

- Body 1, 2 a část bodu 3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Křivka Zbyněk, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 25. října 2021

Abstrakt

Tato práce se zabývá generováním syntaktických analyzátorů pomocí nástrojů používajících pokročilé metody syntaktické analýzy. Tyto metody jsou porovnány jak z teoretického, tak z praktického hlediska. Podrobně je pak zkoumán nástroj GNU Bison, který používá metodu LALR(1) a Generalizovanou LR analýzu, a nástroj ANTLR používající modernější metodu ALL(*). Pro porovnání efektivity těchto nástrojů je pomocí nich implementován syntaktický analyzátor pro smyšlený programovací jazyk, který demonstruje silné a slabé stránky jednotlivých přístupů. Provedeným výzkumem bylo zjištěno, že je nástroj GNU Bison mnohem výkonnější, zatímco ANTLR jej předčí z hlediska funkcionality a přívětivosti implementace. Výsledky této práce mohou pomoci při rozhodování, který přístup či nástroj zvolit při implementaci syntaktického analyzátoru.

Abstract

This bachelor thesis deals with parser generation by tools that use advanced parsing techniques. These techniques are compared from both theoretical and practical point of view. The GNU Bison tool, which uses the LALR(1) method and Generalized LR method, and the ANTLR tool, which uses the more modern ALL(*) method, are examined in detail. To compare the effectiveness of these tools, a parser for a fictional programming language is implemented using them to demonstrate the strengths and weaknesses of each approach. As the results, GNU Bison is much more powerful, but the ANTLR outweighs it in terms of implementation friendliness and functionality. The results of this thesis can help deciding which approach or tool to choose when implementing a parser.

Klíčová slova

Syntaktická analýza, lexikální analýza, překladač, generátor syntaktických analyzátorů, gramatika, GNU Bison, ANTLR

Keywords

Parsing, lexical analysis, compiler, parser generator, grammar, GNU Bison, ANTLR

Citace

HAVRANEK, Daniel. *Pokročilé generování syntaktických analyzátorů*. Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Zbyněk Křivka, Ph.D.

Pokročilé generování syntaktických analyzátorů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Zbyňka Křivky, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Daniel Havranek
10. května 2022

Poděkování

Rád bych poděkoval vedoucímu práce panu Ing. Zbyňku Křivkovi, Ph.D. za připomínky a rady při jejím řešení.

Obsah

Kapitola 1

Úvod

Jelikož počítač rozumí pouze strojovému kódu ve formě jednoduchých instrukcí, který je pro člověka těžko čitelný, byly navrženy abstraktnější způsoby zápisu těchto instrukcí, které se blíží lidskému jazyku — programovací jazyky. Tato uchopitelnější forma se však musí překládat do onoho strojového kódu, kterému rozumí počítač. K tomuto úkonu slouží v informatice překladač.

Programování překladačů je náročná činnost, která se skládá z mnoha dílčích částí, a proto krátce poté, co vznikly první překladače, začali informatici uvažovat nad její automatizací. Hlavní myšlenka spočívala ve tvorbě nástroje, který by byl schopný některé části překladače vygenerovat na základě jejich abstraktnějšího popisu.

Generátor překladače je tedy nástroj, který, na základě formálního popisu jazyka a cílového stroje, dokáže vytvořit interpret nebo překladač. Ne všechny části překladače je snadné generovat, a proto podobné nástroje nejčastěji poskytují pouze funkce pro generování analytických částí překladače.

Vstupem takovýchto generátorů je soubor obsahující gramatiku, která popisuje syntaxi cílového programovacího jazyka. Výstupem je pak zdrojový kód syntaktického analyzátoru takového programovacího jazyka. Přeložením vygenerovaných souborů získáme syntaktický analyzátor, jehož vstupem je zdrojový kód cílového programovacího jazyka a výstupem může být například abstraktní syntaktický strom.

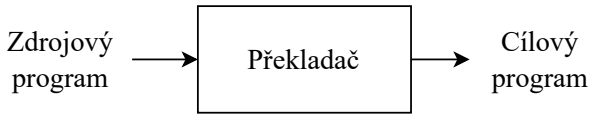
Cílem této práce je nastudování existujících nástrojů pro generování syntaktických analyzátorů založených na pokročilých metodách syntaktické analýzy a jejich porovnání z hlediska schopností a omezení. Nástroje budou porovnány nejprve teoreticky, na základě jejich přístupu k syntaktické analýze, a následně budou dva z nich vybrány k implementaci syntaktického analyzátoru, který bude demonstrovat jejich silné a slabé stránky. Porovnávání bude probíhat na základě časové efektivity a efektivity zápisu analyzovaného jazyka.

V kapitole 2 je vysvětlena struktura a fungování překladače pro zasazení syntaktické analýzy do kontextu překladu. Kapitola 3 podrobněji popisuje syntaktickou analýzu a přístupy k jejímu provádění. Dále jsou zde popsány pokročilé metody syntaktické analýzy, mezi které patří LALR(1), GLR, PEG a LL(*). Na konci této kapitoly jsou představeny některé existující generátory syntaktických analyzátorů používající tyto metody. Kapitola 4 se věnuje návrhu porovnání těchto generátorů. V kapitole 5 je porovnáván nástroj GNU Bison s nástrojem ANTLR. Porovnání je nejprve provedeno z formálního hlediska a následně jsou zde rozepsány rozdíly při implementaci syntaktického analyzátoru. Poslední kapitola 6 obsahuje shrnutí celé práce a návrh jejího pokračování.

Kapitola 2

Struktura překladače

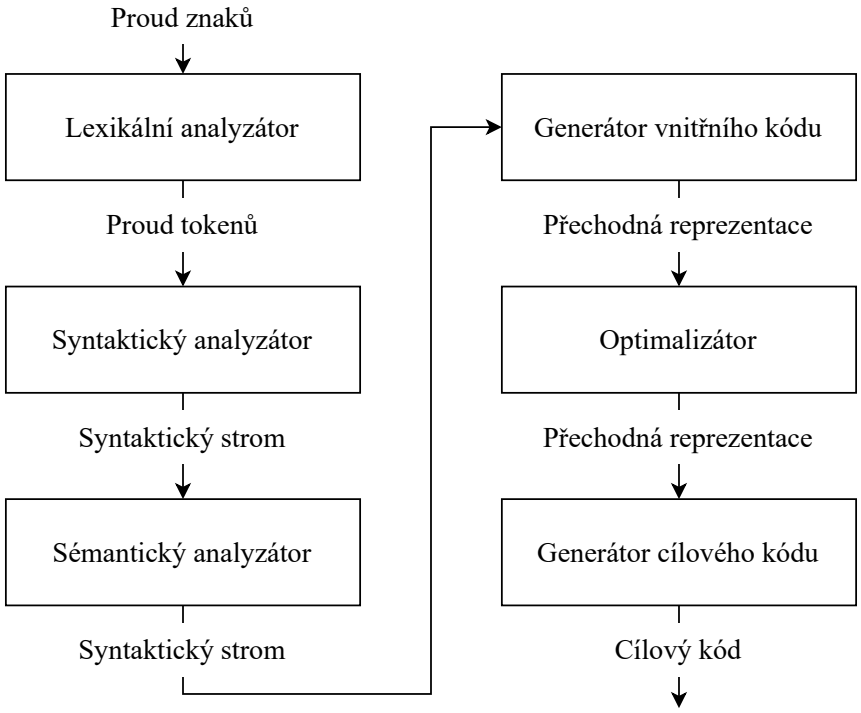
Celá následující kapitola je převzata z článku [1]. Jednoduše řečeno, překladač je program, který umí přečíst program v jednom jazyce — zdrojovém — a přeložit jej do shodného programu v jazyce jiném — cílovém (Obrázek 2.1). Důležitou rolí překladače je hlášení všech chyb ve zdrojovém programu, které se objeví během procesu překladu.



Obrázek 2.1: **Překladač**

Pokud nahlédneme dovnitř takového překladače, zjistíme, že se skládá ze dvou hlavních částí: analýzy a syntézy. Analýza rozloží zdrojový program na menší části a aplikuje na ně gramatickou strukturu. Následně překladač používá tuto strukturu k vytvoření přechodné reprezentace zdrojového programu. Pokud analytická část detekuje, že je zdrojový program zapsán syntakticky nebo sémanticky špatně, musí poskytnout informativní zprávu, aby mohl uživatel chybu opravit. Analýza také shromažďuje informace o zdrojovém programu a ukládá je do datové struktury zvané tabulka symbolů, která je předávána s přechodnou reprezentací k syntéze.

Syntéza konstruuje požadovaný cílový program z přechodné reprezentace a informací v tabulce symbolů. Analýza je často v angličtině nazývána jako *front-end* překladače, zatímco syntéza jako *back-end*. Pokud prozkoumáme kompilační proces více do detailu, zjistíme, že funguje jako sekvence několika fází, kde každá transformuje jednu reprezentaci zdrojového programu na jinou. Typická dekompozice překladače je vyobrazena v obrázku 2.2. V praxi můžou být některé fáze složené do jedné a přechodná reprezentace nemusí být nutně zahrnuta. Tabulka symbolů, která uchovává informace o programu, je používána ve všech fázích překladu.



Obrázek 2.2: **Fáze překladu**

2.1 Lexikální analýza

První fáze překladu se nazývá lexikální analýza neboli skenování (angl. **scanning**). Lexikální analyzátor čte proud znaků, které tvoří zdrojový program, a sjednocuje je do smysluplných sekvencí, které se nazývají lexémy. Pro každý lexém produkuje tento analyzátor výstup v podobě tzv. tokenu skládajícího se z názvu a hodnoty a předává jej do následující fáze, syntaktické analýzy.

Pro vstupní řetězec běžného programovacího jazyka ve tvaru

```
position = initial + rate * 60
```

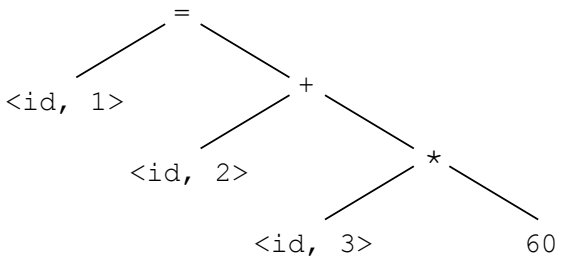
vytvoří lexikální analyzátor následující posloupnost tokenů (oddělené znaky '<' a '>')

```
<id, 1> <=> <id, 2> <+> <id, 3> <*> <60>
```

2.2 Syntaktická analýza

Druhou fází překladu je syntaktická analýza. Syntaktický analyzátor používá tokeny získané od lexikálního analyzátoru k vytvoření přechodné reprezentace v podobě stromové struktury, která vyobrazuje gramatickou strukturu proudu tokenů. Typická reprezentace je syntaktický strom, ve kterém každý vnitřní uzel představuje operaci a potomek tohoto uzlu reprezentuje argument dané operace.

Pro zmíněnou posloupnost tokenů by syntaktický strom vypadal jako na obrázku 2.3.

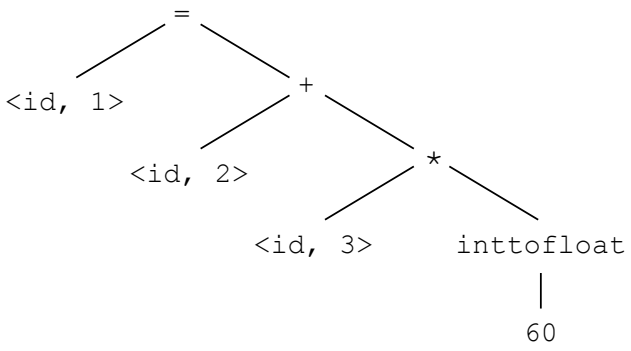


Obrázek 2.3: **Syntaktický strom**

2.3 Sémantická analýza

Sémantický analyzátor používá syntaktický strom a informace v tabulce symbolů ke kontrole zdrojového programu z hlediska sémantické správnosti s definicí jazyka. Také shromažďuje informace o datových typech a ukládá je buďto do syntaktického stromu, nebo do tabulky symbolů pro následné použití při generování vnitřního kódu.

Důležitou částí sémantické analýzy je kontrola typů, při které překladač kontroluje, zda má každý operátor odpovídající operandy. Syntaktický strom po sémantické analýze se nachází v obrázku 2.4.



Obrázek 2.4: **Syntaktický strom po sémantické analýze**

2.4 Generování vnitřního kódu

Při procesu překladu zdrojového programu na cílový kód může překladač zkonstruovat jednu nebo více přechodných reprezentací, které mohou mít různé formy. Syntaktické stromy jsou také formou vnitřní přechodné reprezentace kódu a jsou často používány během syntaktické a sémantické analýzy.

Po syntaktické a sémantické analýze zdrojového programu mnoho překladačů vygeneruje nízkoúrovňovou přechodnou reprezentaci, kterou můžeme vnímat jako program pro abstraktní stroj. Tato reprezentace by měla být jednoduchá k vytvoření a také jednoduchá k překladu pro cílový stroj.

Vnitřní kód vytvořený podle syntaktického stromu (obrázek 2.4) může vypadat například takto:

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

2.5 Optimalizace

Optimalizace kódu nezávislá na cílovém stroji zkouší vylepšit přechodný kód, aby byl následně vytvořen lepší cílový kód. Obvykle je lepší myšleno jako rychlejší, ale mohou být žádoucí i jiné cíle, jako třeba kratší kód, který spotřebuje méně energie.

Vygenerovaný vnitřní kód lze optimalizovat do následující podoby:

```
t1 = id3 * 60.0
id1 = id2 + t1
```

2.6 Generování cílového kódu

Generátor cílového kódu přijímá jako vstup přechodnou reprezentaci zdrojového programu a mapuje ji na cílový jazyk. Pokud je cílovým jazykem strojový kód, jsou vybrány registry a místa v paměti pro každou proměnnou, kterou program používá. Dále jsou přechodné instrukce přeloženy do sekvence strojových instrukcí, které provádějí stejné úlohy. Klíčovým aspektem generování cílového kódu je vhodné přiřazení registrů pro uchování proměnných.

Při použití registru `R1` a `R2` může být přechodný kód přeložen do následujícího strojového kódu:

```
LDF  R2,  id3
MULF R2,  R2,  #60.0
LDF  R1,  id2
ADDF R1,  R1,  R2
STF  id1, R1
```

Kapitola 3

Pokročilé metody syntaktické analýzy

Při vzniku teorií o syntaktické analýze byly výpočetní zdroje ještě stále velmi vzácné, a tak byla efektivita syntaktických analyzátorů hlavním tématem při jejich tvorbě. Bylo tedy logické nechávat programátory jejich gramatiky upravovat tak, aby byly kompatibilní s metodami jako je LL(1) či LALR(1), které tyto požadavky splňují. Dnes již mají počítače takovou výpočetní kapacitu, že je naopak mnohem důležitější efektivita programátora. Z tohoto důvodu byly navrženy mocnější nedeterministické algoritmy a metody, které jsou sice mnohdy náročné na výpočetní zdroje, ale nabízí mnohem větší flexibilitu a uplatnění při tvorbě složitých gramatik. [9]

3.1 Dva přístupy

Základní konexí mezi větou a gramatikou, ze které je derivována, je derivační strom (angl. *parse tree*), který popisuje, jakým způsobem byla gramatika použita k vytvoření oné věty. Pro rekonstrukci této konexe potřebujeme nějakou metodu syntaktické analýzy. Při studování literatury o syntaktické analýze nalezneme takových metod desítky, přesto existují pouze dva přístupy, jak ji lze provést, přičemž zbytek jsou pouze technické detaily.

První metoda zkouší napodobit výrobní proces derivačního stromu znovuzískáním věty z počátečního symbolu. Tato metoda je nazývána shora dolů (angl. *top-down* nebo *LL*), protože je takto derivační strom konstruován.

Druhá metoda zkouší obrátit výrobní proces a redukovat větu zpět na počáteční symbol. Docela přirozeně je tato metoda nazývána zdola nahoru (angl. *bottom-up* nebo *LR*). [4]

LL analýza

Tento typ analýzy je prováděn zleva doprava a hledá nejlevější derivaci. Je to řešeno tak, že začíná na startovacím symbolu a opakovaně rozšiřuje nejlevější neterminál, dokud nedosáhne na cílový řetězec.

Při LL analýze se překladač neustále rozhoduje mezi dvěma akcemi:

1. **Predict**: na základě nejlevějšího neterminálu a nějakého množství načtených tokenů vybere, která produkce by měla být aplikována pro přiblížení se ke vstupnímu řetězci.

2. **Match**: sloučí nejlevější předpovězený terminální symbol s nejlevějším symbolem nezpracovaného vstupu.

LR analýza

Tento přístup je prováděn zleva doprava a hledá nejpravější derivaci. Analyzátor neustále zkouší redukovat část vstupního řetězce zpátky na neterminál.

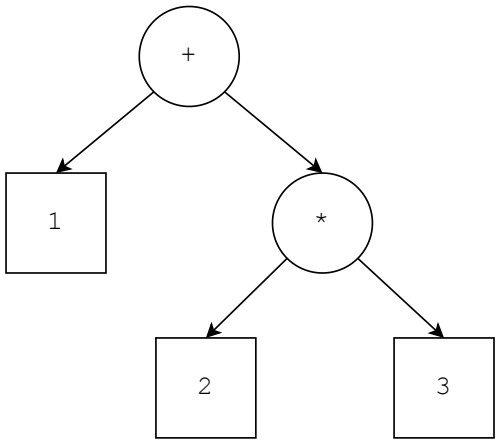
LR analýza se rozhoduje mezi následujícími akcemi:

1. **Shift**: přidá další token ze vstupu do zásobníku pro porovnání.
2. **Reduce**: zredukuje skupinu terminálů a neterminálů v tomto zásobníku zpátky na jiný neterminál obrácenou aplikací pravidla.

Tři hlavní typy LR analyzátorů jsou LR(k), Simple LR(k) a LookAhead LR(k) (zkráceně LR(k), SLR(k) a LALR(k)), kde k je počet tokenů, které musí analyzátor znát. [8, 6]

Podle článku *LL and LR Parsing Demystified* [5], který zveřejnil Josh Haberman, LL analýza přímo odpovídá polské (prefixové) notaci, zatímco LR analýza odpovídá reverzní polské (postfixové) notaci.

Aritmetický výraz `1 + 2 * 3` je zkonstruovaný jako syntaktický strom na obrázku 3.1.



Obrázek 3.1: **Syntaktický strom aritmetického výrazu**

Existují tři způsoby průchodu binárním stromem: *in-order*, *pre-order* a *post-order*. Liší se v tom, zda je nadřazený uzel zpracován před (*pre-order*), po (*post-order*) nebo mezi (*in-order*) zpracováním svých potomků. Toto přesně odpovídá infixové, postfixové a prefixové notaci:

```
1 + 2 * 3  // infixový výraz; in-order průchod.
+ 1 * 2 3  // prefixový výraz; pre-order průchod.
1 2 3 * +  // postfixový výraz; post-order průchod.
```

3.2 LALR

Protože kanonický LR(1) analyzátor (angl. *canonical LR(1) parser*) rozděluje stavy na základě lišících se množin pohledu vpřed, může nabývat mnohem více stavů, než odpovídající SLR(1) nebo LR(0) analyzátor. Potenciálně by mohl vyžadovat rozdělení stavu pouze s jednou položkou do jiného stavu pro každou podmnožinu možných pohledů dopředu. V praxi

to nikdy není tak zlé, ale kanonický LR(1) analyzátor pro programovací jazyk může mít řádově více stavů než SLR(1) analyzátor.

S LALR (lookahead LR) analýzou zkoušíme zredukovat počet stavů v LR(1) analyzátoru pomocí sloučení podobných stavů. Toto sníží počet stavů na stejný jako v SLR(1), ale stále se zachová část výkonu LR(1). [7]

3.3 Generalizovaná LR analýza

Zobecněné deterministické překladače jsou deterministické překladače rozšířené o algoritmus prohledávání do šířky (angl. *breadth-first search*). Díky tomu jsou schopné operovat s tabulkami, které obsahují konflikty (nejednoznačnosti).

Přestože většina jazyků pro praktické použití je LR (deterministických), většina gramatik takových není. A pokud zkusíme navrhnout LR gramatiku pro jeden z těchto jazyků, zjistíme, že je těžké ji zkonstruovat, nebo gramatika neposkytuje správné strukturování pro sémantiku, nebo obojí dohromady. Proto je praktické využití čisté LR syntaktické analýzy hodně omezené. [4]

Na druhou stranu je většina prakticky využitelných gramatik tzv. skorodeterministická, což znamená, že obsahuje pouze malé množství nejednoznačností.[4] Pro rozpoznání takových gramatik vznikla Generalizovaná LR analýza (angl. *Generalized LR*) neboli GLR, která do LR přidává prohledávání do šířky. Později bylo zjištěno, že GLR fungují dobře i na gramatiky s větším množstvím nejednoznačností.

Celý název tedy znamená, že se jedná o syntaktickou analýzu zleva doprava, zdola nahoru s rozšířením o prohledávání do šířky. Prohledávání do šířky je limitováno informacemi o nejednoznačnostech. Poprvé byl algoritmus popsán Langem v roce 1974 bez zájmu veřejnosti. V roce 1984 metodu zpopularizoval Tomita díky své knize [11]. [2]

Algoritmus

1. Pro každou možnou redukci je vytvořena kopie zásobníku, na které je daná redukce aplikována. To odstraní část pravé strany zásobníku a přesune na její místo neterminál. Podle tohoto neterminálu dále nalezneme nový stav, který můžeme položit na vrchol zásobníku. Pokud tento stav znovu umožňuje redukce, je tento krok opakován, dokud nebude postaráno o všechny redukce.

2. Zásobníky, které mají nejpravější stav, který nepodporuje posun dalšího tokenu na vrchol, jsou zahozeny, přičemž jsou poté kopie dalšího tokenu ze vstupu jsou pak přesunuty na zbývající zásobníky.

Zde se však musíme zaměřit na mnoho věcí. Pokud automat používá *lookahead* metodu, mělo by toho být využito v prvním kroku, aby nevznikaly zbytečné kopie zásobníků. Pokud gramatika obsahuje neterminály, které se převádí samy na sebe, bude tato operace nepřetržitě prováděna. Gramatiky se skrytou levou rekurzí vyústí v nekonečné ϵ-redukce. Pokud byly všechny zásobníky zahozeny, vstup obsahoval chybu na tomto specifickém místě. [4]

3.4 Parsing Expression Grammar (PEG)

Zatímco většina jazykových teorií využívá generativní paradigma, většina praktických užití jazyků v informačních technologiích zahrnují rozpoznání a strukturální dekompozici řetězců. Vyplnění této mezery mezi generativními definicemi a praktickými rozpoznávacími

nástroji je důvodem stále rostoucí škály metod syntaktické analýzy s různými možnostmi a kompromisy.

Chomského generativní systém gramatik, ze kterého vychází bezkontextové gramatiky (CFG) a regulární výrazy (RE), byl původně navržen jako formální nástroj pro modelování a analýzu přirozených (lidských) jazyků. Pro jejich eleganci a vyjadřovací schopnosti, informatici přijali generativní gramatiky i pro popis strojově orientovaných jazyků. Schopnost bezkontextových gramatik vyjadřovat nejednoznačnou syntaxi je důležitým a mocným nástrojem pro přirozené jazyky. Tato moc ale vstoupí do cesty, když používáme bezkontextové gramatiky pro strojově orientované jazyky, které by měly být přesné a jednoznačné. Je však složité se vyhnout nejednoznačnostem v bezkontextových gramatikách, a to vytváří v obecné bezkontextové analýze problém superlineárního času.

PEG nabízí alternativu, formální popis syntaxe jazyka založený na rozpoznávání. PEG jsou stylově stejné jako CFG s přidanými vlastnostmi regulárních výrazů, podobné EBNF (Rozvinuté Backusově–Naurově formě). Klíčový rozdíl je ten, že namísto operátoru neuspořádaného výběru '|', který je používán pro zápis alternativních rozkladů neterminálu v EBNF, PEG využívá operátor upřednostněného výběru '/'. Tento operátor zajišťuje, aby byly alternativy testovány popořadě a aby byla bezpodmínečně použita první úspěšná shoda.

Pravidla EBNF 'A → a b | a' a 'A → a | a b' jsou shodná v CFG, ale pravidla PEG 'A ← a b / a' a 'A ← a / a b' jsou rozdílná. Druhá alternativa v posledním PEG pravidle nikdy neuspěje, protože první možnost je vybrána vždy, když vstupní řetězec k rozpoznání začíná znakem 'a'. PEG může být chápán jako formální popis syntaktické analýzy shora dolů.

PEG mají mnohem více vyjadřovacích schopností ohledně syntaxe, než LL(k) třída jazyků, často spojovaná s analýzou shora dolů, a dokáže vyjádřit všechny deterministické LR(k) jazyky a mnoho dalších, včetně některých kontextově závislých jazyků. Přes jejich značnou vyjadřovací sílu, všechny PEG mohou být vykonány v lineárním čase, a to při použití tabulkového nebo memorovaného analyzátoru. [3]

Tento formalismus představil Bryan Ford v roce 2004 a je blízký rodině jazyků pro analýzu shora dolů z počátku 70. let. PEG připomíná syntaxí CFG (bezkontextové gramatiky), ale narozdíl od CFG nemůže být nejednoznačná. Vždy vybere první shodu a pokud řetězec projde syntaktickou analýzou, má pouze jeden validní syntaktický strom.

3.5 LL(*)

Podle Terence Parra, v článku o metodě LL(*) [9], není syntaktická analýza vyřešeným problémem, i přes její důležitost a dlouhou historii akademických studií. Generátory syntaktických analyzátorů se stále potýkají s problémy ohledně použitelnosti a vyjadřovacích metod.

Výrazná výhoda PEG (sekce 3.4) a GLR (sekce 3.3) spočívá v tom, že přijmou všechny gramatiky, které odpovídají jejich meta-jazyku (kromě levé rekurze u PEG). Programátoři už se tedy nemusí prodírat velkým množstvím chybových zpráv. I přes tuto výhodu nejsou GLR ani PEG zcela vyhovujícími z řady důvodů.

GLR a PEG nedělají vždy to, co je očekáváno. GLR tiše akceptuje nejednoznačné gramatiky, ty, které přijmou stejný vstup různými způsoby, což nutí programátory detekovat nejednoznačnosti dynamicky. PEG nemají žádný koncept konfliktu, protože vždy vyberou první shodu, což může vést k nečekanému, či nevyhovujícímu chování. Jak bylo zmíněno

v sekci 3.4, některá pravidla nemusí být nikdy použita. V rozsáhlých gramatikách nejsou taková rizika vždy zřejmá, a i zkušení vývojáři je mohou bez komplexních testů přehlédnout.

Ladění nedeterministických analyzátorů může být velmi složité. S analýzou zdola nahoru reprezentuje jeden stav většinou více míst v gramatice, což programátorům ztěžuje možnost předpovídat, jaká situace bude následovat. Analyzátory shora dolů jsou jednodušší na pochopení, protože je zde mapování jedna ku jedné z LL gramatiky na jejich operace. Rekurzivní sestupné LL implementace dovolují programátorům používat standardní ladící nástroje na úrovni zdrojového kódu k procházení kroků analyzátoru a vestavěných akcí, což usnadňuje jejich pochopení. Tato výhoda je však výrazně oslabena pro rekurzivní sestupné packratové analyzátory, používající backtracking (angl. *backtracking recursive-descent packrat parsers*). Vnořený backtracking je velmi těžké sledovat.

Generování chybových hlášení vysoké kvality je v nedeterministických analyzátorech rovněž složité, ale pro vývojáře velmi důležité. Poskytování užitečných zpráv o syntaktických chybách závisí na kontextu analyzátoru. Například pro správné zotavení se z nesprávně zapsaného výrazu, analyzátor potřebuje vědět, jestli zpracovává index pole, nebo třeba přiřazení. V prvním případě by se měl analyzátor synchronizovat přeskočením na znak pravé hranaté závorky (']'). V druhém případě by měl přeskočit na znak středníku (';'). LL analyzátory mohou hlásit věci jako "nesprávný výraz v indexu pole". LR analyzátory naopak vědí jistě jen to, že zpracovávají výraz. Typicky nejsou schopné se tak dobře vypořádat s chybovým vstupem. Packratové analyzátory mají také nejednoznačný kontext. Navíc se nedokáží zotavit z chybného vstupu, dokud jej nezpracují celý.

Nedeterministické strategie syntaktické analýzy nemohou jednoduše podporovat libovolné vestavěné sémantické akce, které jsou důležité pro práci s tabulkou symbolů, konstruování datových struktur a podobně. "Spekulativní"analyzátory nemohou vykonávat vedlejší akce, jako je například příkaz výpisu na obrazovku, protože zvažovaná akce možná nemá být doopravdy vykonána. V GLR analyzátorech může dojít k problému při výpočtu sémantické hodnoty pravidla. Například když může analyzátor zpracovat stejné pravidlo více způsoby, měl by vykonat rovněž více výpočtů, které se pak mohou lišit.

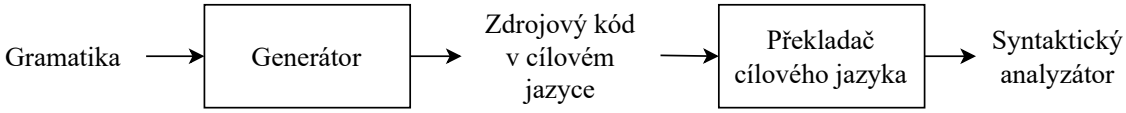
GLR a PEG řeší tyto nevýhody buďto nepodporováním těchto akcí, nepovolováním libovolných akcí, nebo spoléháním na programátora, aby se vyhýbal vedlejším účinkům v akcích, které mohou být vykonány spekulativně.

LL(*) algoritmus konstruuje cyklický deterministický konečný automat s pohledem dopředu (angl. *cyclic lookahead deterministic finite automaton*) pro zpracování konstrukcí, které nejsou LL(k) a následně selže do backtrackingu přes syntaktické predikáty, pokud nedokáže najít vyhovující deterministický konečný automat.

Syntaktické analyzátory používající LL(*) disponují stejnými, někdy i lepšími, vyjadřovacími schopnostmi jako PEG, díky sémantickým predikátům. Zatímco GLR akceptuje gramatiky s levou rekurzí, nedokáží rozeznat jazyky citlivé na kontext, jako to dokáže LL(*). Na rozdíl od PEG nebo GLR, LL(*) analyzátory umožňují libovolné vykonávání akcí a poskytují dobrou podporu pro ladění a řešení chybových vstupů. [9]

3.6 Generátory syntaktických analyzátorů

Jak již bylo zmíněno v úvodu, generátory syntaktických analyzátorů dokáží na základě formálního popisu jazyka vygenerovat zdrojový kód takového analyzátoru v určitém cílovém jazyce. Ten je pak možné přeložit a získat tak spustitelný syntaktický analyzátor, či překladač (Obr. 3.2).



Obrázek 3.2: **Generování syntaktických analyzátorů**

Generátorů syntaktických analyzátorů je celá řada. V této sekci jsou vybrány pouze některé z nejznámějších.

GNU Bison

GNU Bison je univerzální generátor syntaktických analyzátorů, který převádí bezkontextovou gramatiku na deterministický LALR analyzátor (sekce 3.2) nebo GLR analyzátor (sekce 3.3). Je kompatibilní s YACC, takže správně zapsané gramatiky v YACC budou bez problému přeloženy také s GNU Bison. Tento generátor původně napsal Robert Corbett v roce 1985 v jazyce C v kombinaci s m4. Zvládne generovat cílový kód pro C, C++ a Javu. V nejnovější verzi je i podpora jazyka D, ale stále se jedná pouze o experimentální vlastnost. Nejnovější stabilní verze je momentálně 3.8.2. Tento generátor je poskytován pod GPL licencí a je kompatibilní se všemi systémy, především však s UNIXovými. Často je používán spolu s nástrojem Flex pro cílové jazyky C a C++.

GNU Bison implicitně používá metodu LALR(1). Pokud mají být zpracovávány nejednoznačné gramatiky, musí být nastaveno použití GLR a případně očekávaný počet *reduce/reduce* nebo *shitf/reduce* konfliktů.

Flex

Flex je nástroj, který generuje zdrojový kód pro lexikální analyzátor v jazyce C. Jeho součástí je i nástroj flex++ pro generování zdrojového kódu v C++. Často je používán s generátory YACC nebo GNU Bison, přičemž jej lze lépe přizpůsobit s GNU Bison a výsledný kód je výrazně rychlejší. Jeho autorem je Vern Paxson, který jej vytvořil kolem roku 1987 a je poskytován pod BSD licencí.

Vždy je vygenerována funkce `yylex`, kterou pak volá syntaktický analyzátor pro získání tokenů.

PEG.js

PEG.js je jednoduchý generátor překladačů pro JavaScript s kvalitním hlášením chyb. Jak název napovídá, využívá pro analýzu metodu PEG. Lze jej použít k jednoduché tvorbě transformátorů, interpretů, překladačů a jiných nástrojů. Byl vytvořen a do roku 2017 vyvíjen Davidem Majdou. Od roku 2017 převzal vývoj Futago-za Ryuu.

PEG.js je kompletně napsaný v JavaScriptu. Jeho velkou výhodou je tedy to, že není závislý na žádné platformě a pro jeho fungování postačí pouze prohlížeč s podporou tohoto jazyka. Vzhledem k tomu, že oficiální web[1] nabízí i online verzi tohoto překladače, je bez jakékoli instalace dostupný odkudkoliv. Je možné ho však spustit i z terminálu, například

[1] https://pegjs.org/

pomocí JavaScriptového interpretu Rhino. Je poskytován pod licencí MIT. Podle oficiálního webu se na PEG.js stále ještě pracuje a do verze 1.0 nelze zaručit kompatibilitu.

Jeho využití je zejména při potřebě syntaktické analýzy na straně klienta ve webových aplikacích.

Nerozlišuje lexikální a syntaktickou analýzu a nejednoznačnosti řeší prioritizací gramatických pravidel.

Syntaxe gramatiky je hodně podobná JavaScriptu. Vstupní gramatika je psána v souborech s příponou `.pegjs`. Pravidlům může předcházet sekce inicializátoru, oddělená složenými závorkami. Tato část kódu je psána v JavaScriptu a provedena ještě před začátkem analýzy. Všechny proměnné a funkce z této části jsou pak dostupné v akcích samotných pravidel a sémantických predikátech. Tokeny se definují stejně jako pravidla samotná. PEG.js je přímo nerozlišuje.

ANTLR

ANTLR (zkratka pro anglický název *ANother Tool for Language Recognition*) pracuje s jednou z nejnovějších metod syntaktické analýzy zvanou ALL(*). Tato metoda vychází z metody LL(*) (sekce 3.5) a navrhl ji Terence Parr, který také se svým týmem vyvíjí samotný generátor. ANTLR je napsán v Javě a podporuje generování cílového kódu jak pro Javu, tak i pro C#, Python, JavaScript, Go, C++, PHP, Swift nebo Dart. Nejnovější stabilní verze je momentálně 4.9.2. Tento generátor je poskytován pod 3-bodovou BSD licencí a protože je napsán v Javě, je také multiplatformní.

Jeho vstupem je bezkontextová gramatika rozšířená o syntaktické a sémantické predikáty a vložené akce. Syntaktické predikáty umožňují libovolně se dívat dopředu, zatímco sémantické predikáty řídí syntaktickou analýzu. Akce jsou psány v jazyce analyzátoru a mají přístup k aktuálnímu stavu.

Experimenty prokázaly, že ANTLR generuje efektivní syntaktické analyzátory, eliminující skoro všechen backtracking. ANTLR je v praxi hojně používán, což indikuje dobré schopnosti metody ALL(*). [9]

Oak

Oak je generátor syntaktických analyzátorů pro Rust založený na PEG (sekce 3.4). Tento projekt začal s myšlenkou automatického odvození typů syntaktického stromu generovaného výrazy gramatiky. Generátor je napsán jako procedurální makro a může být vložen do kódu bez komplikací při sestavení. Je poskytován pod licencí Apache License 2.0.

LALRPOP

LALRPOP je rovněž generátor pro jazyk Rust. Jedná se o populárnější a aktivnější projekt, než je Oak. Zaměřuje se především na použitelnost a čitelnost gramatiky. I přes zavádějící název LALRPOP implicitně používá metodu LR(1), ale může být nastaven k používání LALR(1). Je však ještě stále ve fázi vývoje a aktuální verze je 0.19.8 poskytována pod licencí MIT/Apache License 2.0.

YACC

YACC (zkratka pro anglický název *Yet Another Compiler Compiler*) je generátor syntaktických analyzátorů pro jazyk C vyvinutý Stephenem C. Johnsonem pro UNIXové systémy. Používá metodu LALR. Dnes se už místo něj používá například GNU Bison.

PLY

PLY (zkratka pro anglický název *Python Lex-Yacc*) je obdoba YACC napsaná kompletně v Pythonu, pro který rovněž generuje cílový kód. Jeho autorem je David M. Beazley a stejně jako YACC používá tento generátor metodu LALR. Je poskytován pod licencí LGPL.

Elkhound

Elkhound nabízí efektivní implementaci GLR, kde je parsovací rychlost při gramatikách podobných LALR(1) podobná YACC. Podporuje jazyky C++ a Ocaml. Je poskytován pod BSD licencí.

Kapitola 4

Zhodnocení současného stavu a návrh řešení

Tato práce se zabývá pokročilými metodami syntaktické analýzy a nástroji, které je využívají při generování syntaktických analyzátorů. Pro demonstraci jejich rozdílů a pokroku v této oblasti bylo zvoleno porovnání určitého generátoru používajícího metodu klasickou s generátorem používajícím modernější přístup.

Jako klasický generátor byl vybrán jeden z nejznámějších, GNU Bison (viz sekce 3.6). Tento nástroj nabral na popularitě hlavně díky tomu, že byl zahrnut jako výchozí generátor syntaktických analyzátorů v distribucích systému GNU Linux. Používá tabulkami řízený překlad, který byl v době jeho vzniku zásadní z důvodu úspory zdrojů jak časových, tak paměťových. GNU Bison může tvořit na základě bezkontextové gramatiky deterministické LR překladače (viz sekce 3.2), nebo zobecněné LR překladače (viz sekce 3.3), využívající LALR(1) tabulky. Mezi jeho cílové jazyky patří především C a C++.

S vybraným klasickým generátorem není vhodné porovnávat například ty generující překladače pro JavaScript, a to z důvodu jejich odlišného zaměření. PEG.js je dnes značně rozšířený nástroj, nicméně používá se především pro analýzu souborů a vstupů ve webových aplikacích, zatímco GNU Bison nalezne využití především při tvorbě výkonných překladačů programovacích jazyků a jiných desktopových aplikací. Pro generátor s moderním přístupem byl tedy vybrán ANTLR 4 (viz sekce 3.6), jakožto jeden z dnes nejaktivnějších projektů tohoto typu. Tento nástroj používá ojedinělou metodu ALL(*) a zaměřuje se především na čitelnost generovaného kódu. Disponuje celou řadou cílových jazyků.

4.1 Způsob porovnávání

Pro zhodnocení schopností těchto generátorů bude s jejich pomocí implementován syntaktický analyzátor pro smyšlený programovací jazyk. Jejich výstupem bude buďto lokalizace chyby v případě neúspěchu, nebo bude program úspěšně ukončen v případě úspěchu.

Sémantická analýza nebude zahrnuta, protože by na rychlost mělo vliv pouze to, jak, a v jakém jazyce, by byly sémantické akce zapsány. Určitě ale bude porovnáno, jakým způsobem lze s daty v konstrukcích pravidel při analýze pracovat. Toto bude vyobrazeno na jednodušším příkladě.

Měřit se bude časová a paměťová náročnost jak generování zdrojových souborů překladače, tak výsledná rychlost analyzátoru při různě zapsaných a různě velkých vstupních souborech. K měření využití zdrojů bude použit unixový nástroj `time`.

Dále bude srovnávána složitost přepisu jazyka do jednotlivých gramatik na základě jeho popisu, přístup k separaci kódu a gramatiky, řešení lexikální analýzy, debugging a hlášení chyb.

Testování při implementaci těchto gramatik bude probíhat porovnáváním očekávaného výstupu a obdrženého výstupu unixovým nástrojem `diff`. Bude napsán dostatečný počet testů pro ověření správnosti zápisu gramatik.

Pro implementaci bude použit pouze textový editor se zvýrazněním syntaxe a terminál pro překlad. Celá implementace bude provedena na notebooku Acer Swift SF514-52T se systémem Fedora Linux 36 (Workstation Edition). Při měření časové náročnosti bude spuštěn pouze terminál. Nástroj GNU Bison bude používán ve verzi 3.8.2, nástroj ANTLR ve verzi 4.9.2 a nástroj Flex ve verzi 2.6.4. Jako překladač cílových jazyků bude použit GCC ve verzi 12.0.1 a OpenJDK ve verzi 17.0.2.

4.2 Předpoklady

GNU Bison provádí tabulkami řízený překlad, zatímco ANTLR přesouvá logiku překladače do kódu, takže jsou předpokládány velké rozdíly v rychlosti jak při generování zdrojových kódů, tak výsledných produktů. Ze stejného důvodu však nebude možné porovnávat kvalitu generovaného kódu. Je předpokládáno, že ANTLR bude časově i paměťově náročnější, zejména při větších vstupních souborech. Rozhodující bude, zda-li je rozdíl rychlostí tak velký, aby bylo výhodnější i v dnešní době používat víc než 36 let starý nástroj, jako je GNU Bison.

Kapitola 5

Porovnání nástrojů ANTLR 4 a GNU Bison

Už jen to, že je GNU Bison mnohem starším nástrojem, napovídá, kde budou hlavní rozdíly v porovnání s ANTLR. Byl vytvořen v době, kdy hrál výkon tu největší roli, což se většinou projevuje na použitelnosti. Takto dlouho udržovaný software je navíc těžké přizpůsobovat moderním trendům. Dnes je spíše udržován bez přidávání nových funkcí.

ANTLR je naopak aktivně vyvíjen a zaměřen na poskytování nových užitečných funkcí z této oblasti. Komunita vývojářů, kteří tento nástroj používají, je dnes také mnohem větší, což je důležité jak pro jeho vývoj, tak pro řešení problémů na různých fórech. Pro oba nástroje je poskytována kvalitní dokumentace.

V článku [10] jsou popsány praktické výhody nástroje ANTLR oproti GNU Bison. Ten například, na rozdíl od ANTLR, přímo nepodporuje Unicode a tuto funkci není snadné implementovat. Nástroj Flex, který je často používán jako lexikální analyzátor pro GNU Bison, podporuje regulární výrazy, zatímco ANTLR podporuje v definicích lexikálních pravidel tzv. bezkontextové výrazy. Vyjadřovací schopnosti bezkontextových výrazů bývají mnohem kvalitnější a mohou zahrnovat více pravidel. Například je možné jednoduše definovat rekurzivní lexikální pravidla. Při použití GNU Bison s Flex může navíc nastat problém z právního hlediska, neboť jsou oba nástroje poskytovány pod jinou licencí.

5.1 Instalace

GNU Bison byl nainstalován jediným příkazem pomocí balíčkového manažeru DNF a připraven k použití. ANTLR 4 vyžaduje pro svůj běh nainstalovanou Javu ve verzi 1.7 nebo vyšší. Dále musí být stažen soubor `.jar` obsahující samotný nástroj, runtime knihovnu pro překladače v Javě, a testovací nástroj TestRig. Pro jednodušší použití je vhodné jej přidat do systémových proměnných a vytvořit alias `antlr4` pro spouštění nástroje a `grun` pro testování, jak je uvedeno na oficiálním webu[1].

5.2 Zápis gramatiky

Jeden ze zásadních rozdílů mezi těmito nástroji je ve způsobu zápisu gramatik. GNU Bison používá Backusovu–Naurovu formu (BNF), zatímco ANTLR používá její rozvinutou verzi (EBNF).

[1] https://www.antlr.org/

Vstupní gramatika pro GNU Bison je psána v souborech s příponou `.y` v následující formě:

```
%{
/** Prolog */
%}

/** Deklarace */

%%
/** Pravidla gramatiky */
%%
/** Epilog */
```

`%{`, `%}` a `%%` je interpunkce, která odděluje jednotlivé sekce. V prologu se definují makra a proměnné. Také se zde deklarují funkce, lexikální analyzátor a program pro výpis chyb. V části deklarace se deklarují terminální a neterminální symboly, priority operátorů a datové typy sémantických hodnot symbolů. Pravidla gramatiky určují, jak sestavit každý neterminál z jeho částí. Epilog může pak obsahovat jakýkoli jiný kód, například definice funkcí deklarovaných v prologu. Celá tato část je zkopírována do výsledného kódu.

Vstupní gramatika nástroje Flex je psána v souborech s příponou `.l`. Tyto soubory používají stejnou interpunkci jako soubory s gramatikou pro GNU Bison. V prologu se deklarují a definují proměnné pro následné použití. Vše mezi symboly `%{` a `%}` je pak přímo zkopírováno do výsledného souboru. Dále následují pravidla gramatiky ve formě *výraz akce*. V epilogu může být zapsán libovolný uživatelský kód.

ANTLR používá pro vstupní gramatiku syntax podobný YACC s EBNF. Gramatika je zapisována v souborech s příponou `.g4` (pro verzi 4) v následující formě:

```
grammar Název;

/** Nastavení */

/** Pravidla */
```

Název gramatiky musí být shodný s názvem souboru. V sekci nastavení se importují balíčky a knihovny a definují tokeny, pro které neexistují lexikální pravidla, uživatelské funkce a podobně. Následují samotná pravidla gramatiky a na jejich konci pak lexikální pravidla, která jsou zapisována stejně. Povinně se musí definovat pouze název gramatiky a alespoň jedno pravidlo.

5.3 Aritmetické výrazy

Pro představu o tom, jakým způsobem se tvoří syntaktický analyzátor pomocí těchto nástrojů, byla nejprve vyřešena jednodušší úloha, na kterou bude navázáno s rozsáhlejším problémem. Jedná se o analýzu a výpočet běžného způsobu zápisu aritmetických výrazů, tedy v infixové notaci.

Příklad vstupu:

```
(7 - 4) * 2 ^ (4 + 1)
2 + 3
40 / (7 * 3 + 6)
```

Řešení pomocí GNU Bison

Gramatika pro GNU Bison může pro tuto úlohu vypadat například takto:

```
input   :   %empty
        |   input line
        ;

line    :   '\n'
        |   exp '\n'
        ;

exp     :   NUM
        |   exp '+' exp
        |   exp '-' exp
        |   exp '*' exp
        |   exp '/' exp
        |   '-' exp  %prec NEG
        |   exp '^' exp
        |   '(' exp ')'
        ;
```

Využívají se zde tři skupiny pravidel pro jednotlivé neterminály. První skupina říká, že vstup (neterminál `input`) může být buďto prázdný řetězec, anebo vstup následovaný řádkem (neterminál `line`), což ve výsledku znamená nula a víc řádků. Řádek lze rozložit na prázdný řádek (symbol `'\n'`), a nebo výraz (neterminál `exp`) ukončený novým řádkem. Výraz lze poté rozložit pomocí mnoha pravidel reprezentujících jednotlivé operace, nebo na číslo.

Dále je nutné předem deklarovat některé použité prvky této gramatiky:

```
%token NUM
%left '-' '+'
%left '*' '/'
%precedence NEG
%right '^'
```

Toto je zapsáno v deklarační části souboru, mezi prologem a pravidly gramatiky. Čísla načítána ze vstupu nebudou vždy jednoznaková, a proto je deklarován token NUM, který reprezentuje desetinné číslo. Pro tokeny, které mají mít přiřazenou levou, či pravou asociativitu, je použita deklarace `%left` nebo `%right`.

Priorita operátorů je určena pořadím řádku, na kterém jsou zapsány, tak, že čím vyšší je číslo řádku, tím vyšší bude priorita operátoru. Negace (`NEG`), tedy unární mínus, není nijak

asociativní, ale aby se dalo určit jakou má prioritu, zapíše se pomocí deklarace `%precedence`, na kterou je odkazováno v gramatice pomocí direktivy `%prec`.

Protože GNU Bison nemá vestavěný generátor lexikálního analyzátoru, musí být lexikální analýza implementována uživatelem. Zde už záleží, pro který jazyk bude analyzátor generován, jelikož bude celý kód epilogu do výsledného produktu zkopírován. Vzhledem k tomu, že je GNU Bison vytvořen primárně pro jazyk C, je tento příklad vyřešen v tomto jazyce. Pro aritmetické výrazy stačí jednoduchý lexikální analyzátor, který ignoruje mezery a tabulátory, čte desetinná čísla a ostatní znaky posílá jako zvláštní tokeny.

```
#include <ctype.h>
#include <stdlib.h>

int yylex (void) {
  int c = getchar ();
  while (c == ' ' || c == '\t') {
    c = getchar ();
  }
  if (c == '.' || isdigit (c)) {
    ungetc (c, stdin);
    if (scanf ("%lf", &yylval) != 1) {
      abort ();
    }
    return NUM;
  } else if (c == EOF) {
    return YYEOF;
  } else {
    return c;
  }
}
```

Aby byl překlad po spuštění programu vykonán, musí být implementována řídící funkce `main`, která zavolá funkci pro parsování (`yyparse`). V tomto případě od této funkce není nic jiného zapotřebí, takže vypadá následovně:

```
int main (void) {
  return yyparse ();
}
```

Pokud funkce `yyparse` detekuje chybu, zavolá funkci pro hlášení chyb (`yyerror`). Ta musí být rovněž implementována, aby mohl uživatel rozeznat, zda se chyba vyskytla. Funkce přijímá od `yyparse` řetězec s chybovou zprávou, takže nejjednodušší způsob je tuto zprávu vypsat.

```
#include <stdio.h>

void yyerror (char const *s) {
  fprintf (stderr, "%s\n", s);
}
```

Takto zapsaný zdrojový kód generátoru, nacházející se v souboru `expressions.y`, lze přeložit pomocí příkazu `bison expressions.y`, který ve výchozím nastevení vygeneruje zdrojový soubor jazyka C s názvem `expressions.tab.c`. Ten už je možné dále přeložit pomocí C překladače.

Výsledný program přijímá řetězce ze standardního vstupu.

Řešení pomocí GNU Bison s využitím Flex

U složitějších překladačů nemusí být implementace lexikálního analyzátoru tak triviální, jako v tomto příkladě, a proto se často využívá GNU Bison spolu s nástrojem Flex, který řeší lexikální analýzu.

Zdrojový soubor Flex má podobnou strukturu jako zdrojové soubory GNU Bison. V jeho deklarační části je nutné zahrnout hlavičkový soubor syntaktického analyzátoru. Ten GNU Bison vygeneruje v případě spuštění s přepínačem `-d`.

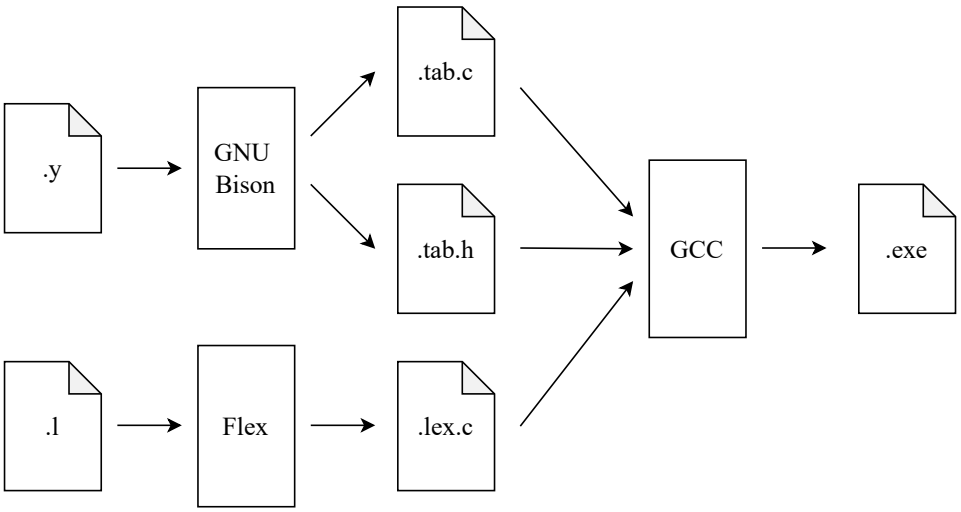
Pravidla souboru Flex se zapisují jako regulární výraz, ke kterému je přiřazena akce ve složených závorkách zapsána v cílovém jazyce. Nejčastěji se jedná pouze o vrácení znaku, či tokenu definovaného v souboru pro GNU Bison. Lexikální pravidla pro tento příklad vypadají takto:

```
[0-9]+              { return NUM; }
\n                  { return '\n'; }
"-"                 { return '-'; }
"+"                 { return '+'; }
"*"                 { return '*'; }
"/"                 { return '/'; }
"("                 { return '('; }
")"                 { return ')'; }
[ \t\v\f]+          {  }
.                   { return yytext[0]; }
```

Soubor pro GNU Bison zůstane skoro nezměněn, akorát už nesmí obsahovat funkci `yylex`. Je také nutné přesunout do souboru Flex funkci `yyerror`.

V epilogu je možné implementovat funkci `yywrap`, která je zavolána na konci vstupu. V případě, že taková funkce není zapotřebí, musí být přidán do deklarační části řádek `%option noyywrap`.

Lexikální analyzátor lze vygenerovat pomocí příkazu `flex expressions.l`, jehož výstupem při výchozím nastavení je soubor `lex.yy.c`. Tento soubor je pak zahrnut při překladu programu, jak znázorňuje obrázek 5.1.



Obrázek 5.1: **Generování s využitím flex**

Řešení pomocí ANTLR

Gramatika pro ANTLR může vypadat například takto:

```
input   :   line* ;

line    :   NEWLINE
        |   exp NEWLINE
        ;

exp     :   <assoc=right> exp '^' exp
        |   '-' exp
        |   exp ('*'|'/') exp
        |   exp ('+'|'-') exp
        |   NUM
        |   '(' exp ')'
        ;

NUM : [0-9]+('.'[0-9]+)?;
NEWLINE :   '\r'? '\n' ;
WS      :   [ \t]+ -> skip ;
```

Na první pohled je tato gramatika velmi podobná té, která byla použita pro GNU Bison. Využívají se zde stejné tři neterminální symboly. Díky EBNF však stačí pouze jedno pravidlo pro neterminál `input`, které říká, že vstup obsahuje nula a víc řádků, a neterminál `exp` lze zapsat pomocí šesti pravidel namísto osmi. Záleží však na pořadí, a proto jsou operace v pravidlech výrazu zapsány sestupně dle priority. Implicitně se počítá s levou asociativitou. Pro nastavení pravé asociativity je použit přepínač `assoc`. Pravidla neterminálu `line` jsou totožné s pravidly GNU Bison, akorát používají token `NEWLINE` namísto konkrétního znaku.

Následuje sekce lexikálních pravidel. Ta obsahuje tři tokeny pro popis čísel, odřádkování a prázdných znaků. Tyto pravidla se zapisují podobně jako ostatní pomocí bezkontextových výrazů. Token `WS` (z angl. *whitespace*) používá příkaz `skip`, pro ignorování bílých znaků. Protože ANTLR obsahuje vestavěný generátor lexikálního analyzátoru, nic víc se za účelem lexikální analýzy psát nemusí.

Pro tento příklad rovněž stačí výchozí hlášení chyb, které je přednastaveno ve velice dobré formě.

V tomto stavu je již možné překladač vygenerovat a otestovat. K tomu byly použity předdefinované aliasy. Testovací nástroj `grun` funguje pouze pro překladače vygenerované pro Javu, ve které je nástroj napsán. Tento jazyk byl tedy zvolen jako cílový. Příkaz `antlr4 Expressions.g4` vygeneruje několik zdrojových souborů s příponou `.java`. Ty lze přeložit pomocí java překladače příkazem `javac Expressions*.java`. Následně je možné překladač otestovat pomocí nástroje `grun`, který přijímá řetězce ze standardního vstupu. Jako první parametr musí být uveden název gramatiky a jako druhý parametr startovací pravidlo, tedy v tomto případě lze překladač spustit příkazem `grun Expressions input`.

Pro výsledný analyzátor je však nutné napsat vlastní řídící funkci v cílovém jazyce. Hlavní třída s řídící funkcí pak může v jazyce Java vypadat například takto:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.stream.Collectors;

import org.antlr.v4.runtime.ANTLRInputStream;
import org.antlr.v4.runtime.CommonTokenStream;

class ExpressionsMain {
    public static void main(String[] args) throws Exception {

        BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
        String inputStream = reader.lines().collect(
            Collectors.joining("\n"));

        ANTLRInputStream input = new ANTLRInputStream(inputStream);
        ExpressionsLexer lexer = new ExpressionsLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        ExpressionsParser parser = new ExpressionsParser(tokens);

        parser.input();
    }
}
```

Funkce nejprve načte řetězec ze standardního vstup pomocí knihovních funkcí Javy. Dále se vytvoří několik instancí tříd jak vygenerovaných, tak dostupných v knihovně ANTLR, potřebných pro fungování překladače. Syntaktická analýza se spustí voláním funkce instance analyzátoru s názvem kořenového pravidla (`input`).

Pro běh překladače je nutná runtime knihovna ANTLR, protože vygenerované soubory používají některé její funkce. Pro překladače vygenerované pro Javu jsou tyto funkce dostupné ve stejném souboru `.jar`, ve kterém jsou i ostatní nástroje pro ANTLR. Soubor s řídící funkcí lze přeložit příkazem `javac ExpressionsMain.java`. Spuštění výsledného překladače je pak možné příkazem `java ExpressionsMain`.

Až do verze 4 ANTLR nepodporoval přímou levou rekurzi, a proto nebylo možné takto gramatiku zapsat. V předchozích verzích by tedy musela být pravidla pro neterminál `exp` rozložena do dalších několika neterminálů tak, jako je zvykem například v PEG. V takovém způsobu zápisu je priorita zajištěna tak, že jsou neterminály rozkládány od operací s prioritou nejnižší, až po nedělitelný prvek, v tomto případě celé číslo.

```
expression
   : multiplyingExpression (('+'|'-') multiplyingExpression)*
   ;

multiplyingExpression
   : powExpression (('*'|'/') powExpression)*
   ;

powExpression
   : signedAtom ('^' signedAtom)*
   ;

signedAtom
   : MINUS signedAtom
   | atom
   ;

atom
   : NUM
   | '(' expression ')'
   ;
```

Tento zápis už není tak intuitivní jako ten předchozí, což mohlo být u starších verzí považováno za nevýhodu. Ve verzi 4 by mohl být tento příklad zapsán oběma ukázanými způsoby. Pokud se tedy gramatika exportuje do ANTLR například z nějakého PEG parseru, může zůstat zapsána takto.

Testování vstupu a rychlost

Jedna z důležitých vlastností překladače je jeho rychlost. Toto, společně s využitím paměti, bylo hlavním prvkem měření běhu výsledných překladačů. Všechny výsledky jsou průměrem deseti spuštění měření.

Kromě výsledné rychlosti překladače je důležitá i rychlost generování z důvodu testování a ladění aplikace při vývoji. Jako první bylo tedy měřeno využití zdrojů při generování syntaktických analyzátorů těmito nástroji. Průměrnou rychlost a využití paměti popisuje tabulka 5.1.

Překlad vygenerovaných souborů je zaznamenán v tabulce 5.2.

Nástroj	Čas	Paměť
GNU Bison	0,034 s	3 413 kB
Flex	0,011 s	3 290 kB
ANTLR	0,450 s	91 395 kB

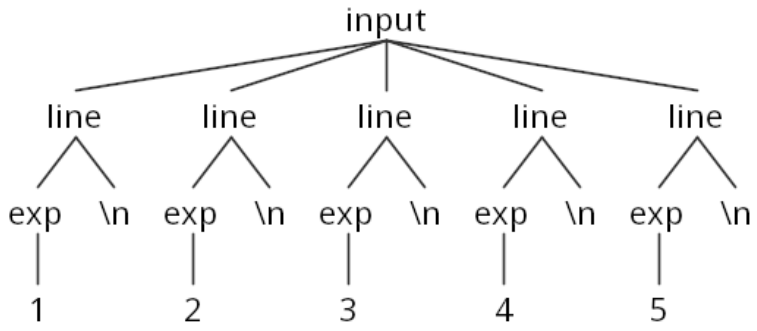
Tabulka 5.1: Využití zdrojů při generování syntaktického analyzátoru aritmetických výrazů

Nástroj	Čas	Paměť
GCC (GNU Bison)	0,065 s	24 792 kB
GCC (GNU Bison / Flex)	0,112 s	25 659 kB
JACAC (ANTLR)	0,915	116 157 kB

Tabulka 5.2: Využití zdrojů při překladu vygenerovaných souborů

Generování zdrojových souborů bylo pomocí GNU Bison a Flex 10krát rychlejší. Tyto nástroje také používali zhruba 27krát méně paměti.

Při testování vygenerovaných syntaktických analyzátorů a tvorbě abstraktního syntaktického stromu jsou zřetelné výhody gramatik zapsaných ve formě EBNF. Pro vstupní soubor o pěti očíslovaných řádcích ANTLR vygeneruje rovnoměrný syntaktický strom (obrázek 5.2). Pro stejný soubor vygeneruje GNU Bison, díky rekurzivnímu zápisu v BNF, syntaktický strom, který je na první pohled méně čitelný (obrázek 5.3).



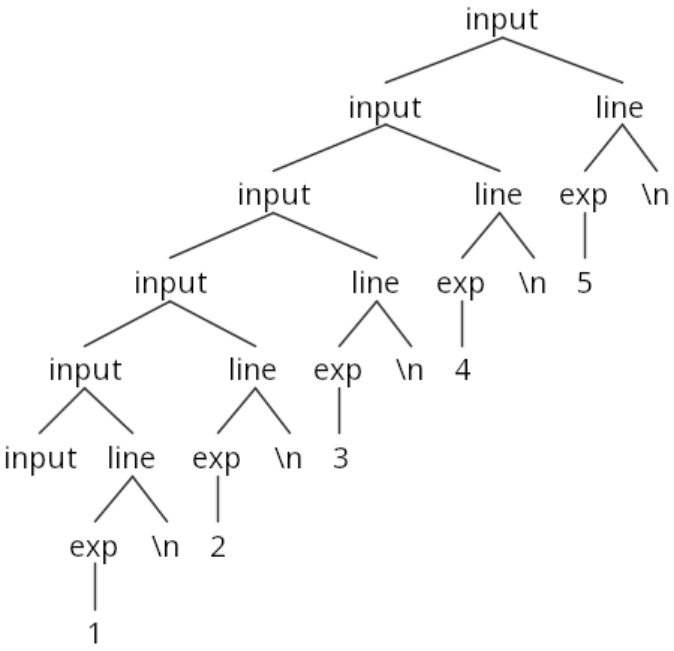
Obrázek 5.2: **Syntaktický strom vygenerovaný pomocí ANTLR.**

Využití zdrojů ve výchozích cílových jazycích

Zpracování souboru o 10 řádcích, kdy měl každý řádek v průměru 24 sloupců, pomocí jednotlivých nástrojů vyobrazuje tabulka 5.3.

Měření zpracování většího souboru o 1000 řádcích, kde měl každý řádek v průměru 60 sloupců, je popsáno v tabulce 5.4.

Při kratším souboru byl GNU Bison víc jak 30krát rychlejší, při delším zhruba 20krát. V kombinaci s Flex byl dokonce rychlejší než bez něj. Poměrem se tedy s rostoucím souborem zhoršovala rychlost více u GNU Bison, nicméně byla stále znatelně vyšší. Zajímavé je, že na rozdíl od ANTLR využíval pro zpracování obou souborů přibližně stejné množství paměti.



Obrázek 5.3: **Syntaktický strom vygenerovaný pomocí GNU Bison.**

Nástroj	Čas	Procesor	Paměť
GNU Bison	0,003 s	90 %	1 591 kB
GNU Bison / Flex	0,003 s	100 %	1 372 kB
ANTLR	0,106 s	151 %	43 666 kB

Tabulka 5.3: Využití zdrojů pro vstupní soubor o 10 řádcích

Nástroj	Čas	Procesor	Paměť
GNU Bison	0,010 s	88 %	1 590 kB
GNU Bison / Flex	0,008 s	85 %	1 350kB
ANTLR	0,194 s	234 %	59 990 kB

Tabulka 5.4: Využití zdrojů pro vstupní soubor o 1 000 řádcích

ANTLR spotřeboval při souboru o 10 řádcích o 3 083 % paměti více a při souboru o 1 000 řádcích bylo využití větší o 4 344 %. GNU Bison byl také méně náročný na procesor.

Detekování chyby

Aby byl překladač užitečný, musí rovněž poskytovat kvalitní hlášení chyb. To je důležité jak pro koncové uživatele, tak pro vývojáře při implementaci překladače z důvodu ladění gramatik.

Mějme pro tento příklad následující chybový vstup, kde se na druhém řádku ve čtvrtém sloupci nachází neznámý token navíc a na třetím řádku v desátém sloupci je neznámý token namísto čísla:

```
(7 - 4) * 2 ^ (4 + 1)
2 + a3
40 / (7 * b + 6)
```

ANTLR hlásí defaultně chyby v následující formě:

```
line 2:4 token recognition error at: 'a'
line 3:10 token recognition error at: 'b'
line 3:12 extraneous input '+' expecting {'-', '(', INT}
```

Je vypsána pozice chyby a její popis. ANTLR pokračuje v analýze i po nalezení chyby, což umožňuje vypsání všech vyskytujících se chyb. V tomto případě je na druhém řádku neznámý lexém `a` navíc. ANTLR toto oznámí, zahodí jej a pokračuje v analýze. Na třetím řádku je chybný lexém `b` namísto čísla. ANTLR udělá to stejné, co předtím, ale tím, že jej zahodí, chybu způsobí následující znak plus, který se nesmí nacházet bezprostředně za znakem krát. ANTLR tedy vypíše očekávané lexémy na této pozici.

Pro přizpůsobení chybových zpráv je nutné implementovat vlastní třídu, která bude sbírat chyby a přidávat je do seznamu. Při vytváření parseru v řídící funkci je pak nutné u jeho instance zavolat funkci `removeErrorListeners`, která odstraní stávající třídu pro zpracování chyb a dále funkci `addErrorListener`, která přiřadí instanci implementované třídy. Taková třída může vycházet z třídy `BaseErrorListener`, kterou poskytuje ANTLR.

GNU Bison naopak při předvedené implementaci (viz sekce 5.3) hlásí chybu pouze zprávou *syntax error*. Taková zpráva je mnohdy nedostačující, a proto je zde možnost ji rozšířit zapsáním následujicí definice do deklarační části:

```
%define parse.error verbose
```

V tomto případě už chybová zpráva vypadá následovně:

```
syntax error, unexpected invalid token, expecting NUM or '-' or '('
```

Toto je mnohem užitečnější hlášení, avšak stále není uvedeno, na jaké pozici se chyba nachází. Pro lokalizaci chyby je v prologu nutné deklarovat globální proměnné pro počítání řádků a sloupců, inkrementovat je ve funkci `yylex` na základě přijatých znaků a následně je vypsat při výskytu chyby. Takto už je možné dosáhnout zprávy ve tvaru:

```
line 2:4 syntax error, unexpected invalid token, expecting NUM or '-'
or '('
```

Takové hlášení už se vyrovná tomu od ANTLR, nicméně je po první nalezené chybě program ukončen. Toto neumožňuje získat představu o tom, kde všude se ve vstupním souboru vyskytují chyby, a proto je dobré implementovat zotavení z chyb. GNU Bison k tomu poskytuje klíčové slovo `error`, které je možné zasadit do pravidel gramatiky. Pokud je dostačující hlásit pouze první chybu na řádku, je možné jej zapsat do pravidla pro `line` takto:

```
line    :   '\n'
        |   exp '\n'
        |   error '\n'
        ;
```

Tento zápis říká, že pokud se při analýze objeví v neterminálu `exp` chyba, bude nahlášena a následující vstup bude ignorován, až do prvního výskytu odřádkování. Pro výše zmíněný chybový vstup tedy už vypadá výstup takto:

```
line 2:4 syntax error, unexpected invalid token, expecting NUM or '-'
or '('
line 3:10 syntax error, unexpected invalid token, expecting NUM or '-'
or '('
```

ANTLR však dokáže lokalizovat i více chyb na jednom řádku. Aby mohl GNU Bison poskytnout stejnou funkcionalitu, musela by být kompletně přepsána pravidla neterminálu `exp`.

Sémantické akce

Při používání těchto nástrojů často není cílem pouze syntaktická analýza, ale také na základě zadaného vstupu získat specifický výstup. K tomu slouží implementace sémantických akcí. Každý generátor k této problematice přistupuje trochu jiným způsobem.

GNU Bison umožňuje přiřadit ke každému pravidlu nějakou akci tak, že se za něj do složených závorek smí napsat program v cílovém jazyce. Například pro vypsání shody s šestým pravidlem této gramatiky by zápis vypadal takto:

```
exp: exp '+' exp   { printf("Matched rule 6\n"); } ;
```

Většinou je účelem těchto akcí výpočet sémantické hodnoty celé konstrukce na základě sémantických hodnot jejich částí. K takovým výpočtům slouží direktiva začínající znakem dolar ('$'). Za tímto znakem následuje buďto číselná hodnota, která značí pozici tokenu v pravidle, nebo další znak dolaru značící výslednou hodnotu celého pravidla. Pro výpočet sémantické hodnoty celé konstrukce ve výše zmíněném pravidle by zápis vypadal takto:

```
exp: exp '+' exp   { $$ = $1 + $3; } ;
```

Nejprve však musí být v deklarační části programu definován datový typ sémantické hodnoty. V tomto příkladě stačí stejný datový typ pro všechny akce:

```
%define api.value.type {double}
```

Podobně lze pak u neterminálu `exp` zapsat sémantické akce pro zbylých šest pravidel. Program může následně vypsat hodnotu každého výrazu ze vstupu pomocí následující akce u pravidla pro neterminál `line`:

```
line: exp '\n' { printf ("\t%.10g\n", $1); }
```

ANTLR také nabízí zápis sémantických akcí do složených závorek za pravidlem přímo v gramatice. Obyčejné vypsání shody by pro stejný cílový jazyk vypadalo identicky s GNU Bison. Pro výpočet sémantické hodnoty musí být rovněž specifikován datový typ, ten se ale definuje jako návratová proměnná u každého neterminálu zvlášť následujícím způsobem:

```
exp returns [double val] : ...
```

K této proměnné a k dílčím prvkům pravidla se přistupuje stejně jako v GNU Bison pomocí znaku dolar. Dílčí prvky jsou objekty obsahující různé atributy. Atribut `text` vrátí řetězec tokenu. Pravidlo, které převádí lexém čísla na desetinnou sémantickou hodnotu pak vypadá takto:

```
exp returns [double val] : NUM { $val = Double.parseDouble($NUM.text); }
```

Tím, že neterminál `exp` obsahuje definovaný atribut `val`, je možné k němu v dalších pravidlech přistupovat:

```
exp returns [double val] : '(' exp ')' { $val = $exp.val; }
```

V pravidlech, kde se nachází více prvků stejného názvu je nutné, aby měli popisek (prvkům předchází identifikátor a rovnítko). Díky tomu na ně lze v sémantické sekci odkazovat:

```
exp returns [double val] : left=exp op=('*'|'/') right=exp {
    $val = ($op.text.charAt(0) == '*') ?
        $left.val * $right.val : $left.val / $right.val;
}
```

Podobně lze doplnit i zbylá pravidla. Pro vypsání výsledné hodnoty každého výrazu ze vstupu slouží následující zápis pravidla pro neterminál `line`:

```
line:   exp NEWLINE
    { System.out.println(String.format("%.2f", $exp.val)); } ;
```

Tím, že jsou všechny tyto akce napsané v cílovém jazyce, nemůže být gramatika znovu použita pro jiný cílový jazyk. Proto ANTLR, narozdíl od GNU Bison, nabízí možnost úplně oddělit gramatiku od sémantických akcí. K tomuto lze přistupovat dvěma způsoby. Jedná se o přístup typu posluchač (angl. *listener*) nebo návštěvník (angl. *visitor*). Implicitně ANTLR k problematice přistupuje strategií posluchače a spolu s lexikálním a syntaktickým analyzátorem vygeneruje také soubory ExpressionsBaseListener.java a ExpressionsListener.java. Soubor ExpressionsBaseListener.java umožňuje implementovat akce na základě událostí v analyzátoru. Defaultně vygeneruje metody pro jednotlivé neterminály v následující podobě:

```
void enterInput(ExpressionsParser.InputContext ctx);
void exitInput(ExpressionsParser.InputContext ctx);
```

V souboru s gramatikou je však možné jednotlivé alternativy rozkladu těchto neterminálů označit pomocí znaku mřížka ('#') například následovně:

```
line    :   NEWLINE         #emptyLine
        |   exp NEWLINE     #lineWithExpression
        ;
```

V takovém případě už budou vygenerovány navíc i následující metody:

```
void enterEmptyLine(ExpressionsParser.EmptyLineContext ctx);
void exitEmptyLine(ExpressionsParser.EmptyLineContext ctx);
void enterLineWithExpression(ExpressionsParser.LineWithExpressionContext ctx);
void exitLineWithExpression(ExpressionsParser.LineWithExpressionContext ctx);
```

V těchto metodách je možné přistupovat k jednotlivým prvkům pravidla pomocí parametru `ctx`.

Zhodnocení implementace

Už tento jednoduchý příklad vypovídá o zásadních výhodách a nevýhodách těchto nástrojů. Při návrhu a implementaci překladače je rozhodně více uživatelsky přívětivý ANTLR. Výhody EBNF se projevují jak při zápisu gramatik, tak při generování abstraktního syntaktického stromu. Vestavěný lexikální analyzátor a to, že nabízí úplnou separaci kódu od gramatiky, zlepšuje přehlednost a znovupoužitelnost. Výchozí hlášení chyb je mnohem kvalitnější a nemusí se kvůli něj měnit gramatika. Na druhou stranu je přes všechny tyto výhody znatelně pomalejší a zrovna rychlost může být mnohdy při implementaci překladače klíčová.

5.4 Podmnožina jazyka C

Pro lepší porovnání schopností těchto generátorů a jejich přístupu k syntaktické analýze byl s jejich pomocí zpracováván smyšlený programovací jazyk s pracovním názvem SubC, který je podmnožinou jazyka C. Pro autentičnost s reálným využitím těchto nástrojů byla vybrána taková podmnožina, která je výpočetně úplná.

Obecné vlastnosti

Identifikátor je definován jako neprázdná posloupnost číslic, písmen a znaku podtržítka začínající písmenem nebo podtržítkem. Jazyk obsahuje následující klíčová slova, která nesmějí být použity jako identifikátory, neboť mají speciální význam:

double, else, for, if, int, return

U identifikátorů a klíčových slov záleží na velikosti písmen (tzv. *case-sensitive*).

Jazyk SubC využívá celočíselné a desetinné literály. Celočíselný literál je tvořen posloupností číslic a vyjadřuje hodnotu celého nezáporného čísla v desítkové soustavě. Desetinný literál je tvořen celou a desetinnou částí a rovněž vyjadřuje hodnotu nezáporného čísla v desítkové soustavě. Obě jeho části jsou tvořeny neprázdnou posloupností číslic a jsou odděleny tečkou. V těchto literálech se nesmí vyskytovat žádný jiný znak včetně znaků bílých.

Datové typy sloužící pro uchovávání literálů jsou `int` a `double`. Proměnné mohou nabývat i záporných hodnot.

Jazyk SubC podporuje řádkové i blokové komentáře. Řádkový komentář začíná dvojicí lomítek ('//') a je ukončen odřádkováním. Blokový komentář začíná symboly '/*' a je ukončen symboly '*/'. Vše, co se nachází uvnitř je ignorováno.

Sémantika se v tomto příkladě nezahrnuje, a proto nebude kontrolována typová správnost, existence proměnných, přítomnost hlavní řídicí funkce a podobně. Validní programy pro tento jazyk tedy nemusí být validními pro jazyk C.

Struktura jazyka

Korektně zapsaný zdrojový soubor jazyka SubC se skládá z libovolného počtu deklarací a definic uživatelských funkcí. V těle definice takové funkce se může nacházet libovolný počet příkazů. Jednoduché příkazy a deklarace funkcí jsou ukončeny středníkem (znak ';'). Mezi jednotlivými lexémy jazyka se může vyskytovat libovolný, avšak nenulový počet bílých znaků.

Příklad programu pro výpočet faktoriálu v jazyce SubC:

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main() {
    int num = 5;
    int res = factorial(num);
    return 0;
}
```

Všechny povolené programové konstrukce se zapisují ve stejném tvaru jako v jazyce C. Jazyk SubC podporuje deklarace a definice uživatelských funkcí, deklarace a definice proměnných, příkaz přiřazení, podmíněné příkazy, cyklus `for`, volání uživatelem definovaných funkcí a příkaz návratu z funkce. Podrobný popis syntaktických konstrukcí jazyka SubC se nachází v příloze C.

Výrazy

Výrazy se zapisují stejně jako aritmetické výrazy (viz sekce 5.3). Skládají se z celých a desetinných čísel, proměnných, závorek a operátorů (aritmetických, logických a relačních). Priorita a asociativita operátorů je popsána v tabulce 5.5.

Priorita	Operátor	Popis	Asociativita
1	-	unární mínus	pravá
2	* /	násobení a dělení	levá
3	+ -	sčítání a odčítání	levá
4	< <= > >=	porovnávání hodnot	levá
5	== !=	rovnost a nerovnost	levá
6	&&	logická konjunkce	levá
7	‖	logická disjunkce	levá
8	=	přiřazení	pravá

Tabulka 5.5: **Priorita a asociativita operátorů jazyka SubC**

Prioritu lze měnit dosazením závorek. Operátor přiřazení se nesmí vyskytovat ve výrazech.

Implementace

Při více složitých gramatikách, jako je tato, se již projevují výhody lepších vyjadřovacích schopností EBNF, které poskytuje ANTLR. Zápis pravidla pro deklaraci funkce v GNU Bison vypadá následovně:

```
function_declaration
    : type ID '(' parameter_list ')' ';'
    ;

parameter_list
    :  %empty
    | parameter parameter_next
    ;

parameter_next
    : %empty
    | ',' parameter parameter_next
    ;

parameter
    : type ID
    ;
```

Za seznam parametrů (neterminál `parameter_list`) lze dosadit libovolný počet dvojic *typ id* oddělených čárkami. Neterminál `parameter_list` tedy musí obsahovat pravidlo `%empty` pro případ nepřítomnosti parametrů. Dále je zde potřeba neterminálu `parameter_next` pro rekurzivní generování následujících parametrů funkce. Stejnou funkcionalitu lze v ANTLR zapsat takto:

```
functionDeclaration
    :   type ID '(' parameterList? ')' ';'
    ;

parameterList
    :   parameter (',' parameter)*
    ;

parameter
    :   type ID
    ;
```

Pro stanovení toho, že je seznam parametrů libovolný, stačí za neterminál `parameterList` zapsat znak otazníku ('?') přímo v pravidle `functionDeclaration`. Pomocí znaku hvězda ('*') lze zase zapsat, že se za parametrem může nacházet nula nebo více dalších parametrů, kterým předchází čárka. Celkově je tak gramatika v této části o tři pravidla kratší a mnohem přehlednější. Podobné rozdíly v zápisu pak nastaly například u definování libovolného počtu konstrukcí `else if`, nebo u seznamu výrazů při volání uživatelem vestavěných funkcí.

Některá místa v použité gramatice mohou být pro jednotlivé generátory a jejich přístupy k syntaktické analýze problematická. To se projeví při běhu překladače a může to ovlivňovat čas, či spotřebu paměti.

Pro LR metody je to například pravá rekurze. Pokud má operátor pravou asociativitu, je pro jeho zápis logicky použita pravá rekurze, aby gramatika správně popisovala jazyk. To vyústí v rostoucí zásobník. V navržené gramatice je pravá rekurze použita při vícenásobném přiřazení.

```
a = b = 1;
```

U LL metod nastává problém při načítání definice a deklarace funkce, protože je jejich zápis stejný až do terminálu ';' nebo '{'.

```
int foo (int a, int b...);

int foo (int a, int b...) {
        ...
}
```

LL analyzátor nemůže pracovat se dvěma pravidly současně, a tak si musí vybrat jedno z nich. S rostoucím počtem argumentů funkce se bude zvětšovat *lookahead*, který je teoreticky sice nekonečný, ale prakticky ne. LR analyzátor se nemusí dívat dopředu, protože dokáže pracovat s více pravidly najednou.

Testování vstupu a rychlost

Následující tabulky obsahují rychlost a využití paměti při generování a používání syntaktického analyzátoru jazyka SubC s pomocí jednotlivých nástrojů. Všechny výsledky jsou průměrem deseti spuštění měření.

Rychlost generování analyzátoru a využití paměti je zaznamenáno v tabulce 5.6.

Nástroj	Čas	Paměť
GNU Bison	0,058 s	3 514 kB
Flex	0,010 s	3 274 kB
ANTLR	0,581 s	121 621 kB

Tabulka 5.6: Využití zdrojů při generování syntaktického analyzátoru jazyka SubC

Překlad vygenerovaných souborů popisuje tabulka 5.7

Nástroj	Čas	Paměť
GCC (GNU Bison / Flex)	0,118 s	25 788 kB
JACAC (ANTLR)	1,242	133 418 kB

Tabulka 5.7: Využití zdrojů při překladu vygenerovaných souborů

Doba generování byla u obou nástrojů nepatrně vyšší, než při generování v sekci 5.3. Využití paměti se procentuálně zvýšilo více u ANTLR. Využití zdrojů při překladu vygenerovaných souborů se zvýšilo u ANTLR, zatímco u GNU Bison zůstalo téměř stejné.

Využití zdrojů ve výchozích cílových jazycích

Zpracování zdrojového souboru o 30 řádcích pomocí jednotlivých nástrojů vyobrazuje tabulka 5.8.

Zpracování velkého souboru, který obsahuje přes 10 000 řádků, vyobrazuje tabulka 5.9.

GNU Bison využíval pro zpracování jazyka SubC téměř stejně paměti, jako pro zpracování výrazů v sekci 5.3, nezávisle na velikosti souboru. Nástroji ANTLR se zvyšovala

Nástroj	Čas	Procesor	Paměť
GNU Bison	0,003 s	90 %	1 361 kB
ANTLR	0,123 s	165 %	46 102 kB

Tabulka 5.8: Využití zdrojů pro vstupní soubor o 30 řádcích

Nástroj	Čas	Procesor	Paměť
GNU Bison	0,011 s	90 %	1 394 kB
ANTLR	0,542	338 %	156 216 kB

Tabulka 5.9: Využití zdrojů pro vstupní soubor o 10 000 řádcích

spotřeba paměti s velikostí souboru, stejně jako v předchozím příkladě. Při analýze souboru o 30 řádcích využil o 3 387 % víc paměti než GNU Bison. Analýzou souboru o 10 000 řádcích spotřeboval o 11 106 % více paměti. Zde se projevuje velká výhoda LR analýzy. S většími soubory se u nástroje ANTLR rovněž zvyšovala výpočetní náročnost, zatímco u GNU Bison ne.

Rychlost analýzy souboru o 30 řádcích byla s GNU Bison víc jak 40krát větší a skoro 50krát větší při analýze souboru o 10 000 řádcích. Poměr rychlosti se tedy nezvětšoval s velikostí souboru, tak jako poměr využité paměti, ale stále znatelně ano.

Detekování chyby

U této gramatiky se ANTLR opět implicitně stará o zotavování z chyb. Gramatika GNU Bison však musí být upravena podobně jako v předchozím případě (sekce 5.3). Tím, že obsahuje složitější programové konstrukce se rovněž tato úprava stává náročnější.

Čistě teoreticky by bylo například vhodné obnovit analýzu po každém řádku kódu, který je zapsán chybně. Tato gramatika, stejně jako větší část gramatiky jazyka C, však umožňuje odřádkování ze zdrojových kódů téměř úplně vynechat, nebo naopak vkládat jej mezi výrazy a podobně. Je tedy nutné zvolit jiné znaky, které po chybě obnoví syntaktickou analýzu.

V tomto případě se například nabízí znak středníku (';') nebo levé složené závorky ('{'), neboť se tyto znaky nachází na konci každého příkazu. U dobře strukturovaných zdrojových kódů končí příkaz odřádkováním, a tak by bylo splněno zmíněné žádoucí chování, tedy obnova analýzy po něm.

Například pravidla pro deklaraci a definici funkce budou po úpravě vypadat takto:

```
function_declaration
    : type ID '(' parameter_list ')' ';'
    | error ';'
    ;

function_definition
    : type ID '(' parameter_list ')' compound_statement
    | error compound_statement
    ;
```

U deklarace se při chybě zahazuje následující vstup až do znaku středníku a následně je analýza obnovena. Stejné chybové pravidlo je použito pro jednoduché příkazy končící středníkem jako je přiřazení, volání uživatelských funkcí a podobně. U definice funkce je analýza

obnovena při načtení složeného příkazu, tedy levé složené závorky. Pokud však bude stejné pravidlo použito i u podmíněných konstrukcí a cyklu, nastanou *shift/reduce* a *reduce/reduce* konflikty. Toto se dá řešit buďto nastavením GLR analýzy s uvedeným počtem očekávaných konfliktů, nebo úpravou gramatiky. V tomto případě je možné gramatiku vcelku jednoduše upravit tak, že bude klíčovému slovu `error` předcházet symbol, který pravidla odliší, a analyzátor tak může stále používat metodu LALR(1). Pro podmíněné příkazy vypadají chybová pravidla po úpravě takto:

```
if_statement
    :   IF '(' expression ')' compound_statement
    |   IF error compound_statement
    ;

else_if_statement
    :   ELSE IF '(' expression ')' compound_statement
    |   ELSE IF error compound_statement
    ;
```

Takto zapsaná gramatika je již schopná se lépe vypořádávat s chybovými vstupy. Je důležité správně zvolit, kde do gramatiky zotavení z chyb přidat, aby bylo při používání analyzátoru efektivní.

Zhodnocení implementace

Při návrhu a implementaci složitější gramatiky se ještě znatelněji projevily výhody práce s nástrojem ANTLR. Ve složitějších programových konstrukcích gramatiky je přehlednost důležitá a způsob zápisu zdrojových souborů GNU Bison nemůže konkurovat tomu od ANTLR. Také se v tomto příkladě ale zvětšily výkonnostní rozdíly obou nástrojů a projevila se síla LR překladačů jak z hlediska spotřeby paměti, tak rychlosti.

Kapitola 6

Závěr

Cílem této práce byl průzkum existujících nástrojů pro generování syntaktických analyzátorů založených na pokročilých metodách syntaktické analýzy a jejich porovnání z hlediska schopností a omezení. Zadání práce bylo splněno a její výsledky mohou pomoci při výběru nástrojů pro implementaci syntaktického analyzátoru.

V práci byly teoreticky popsány a porovnány nejrozšířenější přístupy k řešení syntaktické analýzy. Dále byly popsány existující nástroje pro generování syntaktických analyzátorů, které tyto přístupy používají.

V implementační části práce byl podrobně porovnáván nástroj GNU Bison s nástrojem ANTLR tak, že byl s jejich pomocí implementován syntaktický analyzátor pro smyšlený programovací jazyk, který je podmnožinou jazyka C. Byla srovnávána složitost přepisu jazyka do jejich gramatik, řešení lexikální analýzy, možnost zápisu sémantických akcí a zotavování se z chyb ve vstupních souborech. Měřena byla paměťová a časová náročnost vygenerovaných analyzátorů, ale také jejich generování. Na dostatečném počtu testů byla ověřena požadovaná funkčnost vygenerovaných analyzátorů.

Díky měření rychlosti analýzy různě velkých souborů bylo zjištěno, že je GNU Bison řádově rychlejší než ANTLR. Při jednoduché gramatice byl až 30krát rychlejší a při složitější gramatice až 50krát. GNU Bison také při všech testech využíval téměř konstantní množství paměti, které nepřekročilo 1 400 kB. Spotřeba paměti u ANTLR se zvyšovala s velikostí vstupního souboru a při velkých souborech složité gramatiky byla až o 11 106 % větší. Na druhou stranu byla implementace pomocí GNU Bison o dost náročnější a nepřehlednější. ANTLR zahrnuje lexikální analyzátor, úplnou separaci gramatiky a kódu, přehlednější popis jazyka díky EBNF, mnohem kvalitnější výchozí hlášení chyb a mnoho dalších výhod, které se vztahují k implementaci analyzátoru.

Využití by v dnešní době našel GNU Bison například u vestavěných systémů, kde je důležité šetrné nakládání se zdroji. Počítače však dnes mají dostatek paměti a pro aplikace, jež syntaktický analyzátor potřebují, je jednodušší využít ANLTR. Rychlost je nicméně důležitým faktorem, a proto může být i dnes nástroj jako GNU Bison vhodnější.

Jelikož jsem ve druhém ročníku programoval syntaktický analyzátor ručně, oceňuji schopnosti těchto nástrojů usnadnit implementaci překladačů a v budoucnu bych jejich pomoc určitě využil.

V práci bych chtěl pokračovat měřením výkonu těchto nástrojů při stejném cílovém jazyce, což by pomohlo získat objektivnější pohled na efektivitu jednotlivých metod syntaktické analýzy.

Literatura

[1] Aho, A., Lam, M., Sethi, R. a Ullman, J. *Compilers: Principles, Techniques, & Tools*. Pearson/Addison Wesley, 2007. ISBN 0321486811.

[2] Economopoulos, G. R. *Generalised LR parsing algorithms*. 2006. Disertační práce. Department of Computer Science Royal Holloway, University of London.

[3] Ford, B. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. Massachusetts Institute of Technology. 2004.

[4] Grune, D. a Jacobs, C. *Parsing Techniques: A Practical Guide*. Springer Science Business Media, 2008.

[5] Haberman, J. LL and LR Parsing Demystified. 2013. Dostupné z: https://blog.reverberate.org/2013/07/ll-and-lr-parsing-demystified.html.

[6] Johnson, M. *Bottom-Up Parsing*. Standford University, Department of Computer Science, Červenec 2012. Dostupné z: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/100%20Bottom-Up%20Parsing.pdf.

[7] Johnson, M. *LALR Parsing*. Standford University, Department of Computer Science, Červenec 2012. Dostupné z: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/140%20LALR%20Parsing.pdf.

[8] Johnson, M. *Top-Down Parsing*. Standford University, Department of Computer Science, Červenec 2012. Dostupné z: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/090%20Top-Down%20Parsing.pdf.

[9] Parr, T. a Fisher, S. LL(*): The Foundation of the ANTLR Parser Generator. University of San Francisco and ATT Labs Research. 2011.

[10] Tomassetti, G. Why you should not use (f)lex, yacc and bison. Březen 2020. Dostupné z: https://tomassetti.me/why-you-should-not-use-flex-yacc-and-bison/.

[11] Tomita, M. *LR parsers for natural languages*. Stanford, California, USA: Association for Computational Linguistics, 1984.

Příloha A

Obsah přiloženého paměťového média

- **antlr** – syntaktické analyzátory vytvořené pomocí ANTLR
  - **expressions** – pro analýzu aritmetických výrazů
  - **expressions-semantic** – pro výpočet aritmetických výrazů
  - **sub-c** – pro jazyk SubC
  - **antlr-4.9.2-complete.jar** – knihovna potřebná pro běh
- **gnu-bison** – syntaktické analyzátory vytvořené pomocí GNU Bison
  - **expressions** – pro analýzu aritmetických výrazů
  - **expressions-flex** – pro analýzu aritmetických výrazů s využitím Flex
  - **expressions-semantic** – pro výpočet aritmetických výrazů
  - **sub-c** – pro jazyk SubC
- **samples** – příklady vstupních souborů
  - **expressions** – aritmetické výrazy
  - **sub-c** – jazyk SubC
- **tex** – zdrojové soubory textu této práce
- **text.pdf** – text této práce
- **readme.txt** – manuál pro práci s přiloženým médiem

Každý adresář pro syntaktický analyzátor obsahuje gramatiku pro daný nástroj, Makefile, adresář **src**, ve které jsou vygenerované zdrojové soubory syntaktického analyzátoru, a adresář **bin**, ve které je spustitelný syntaktický analyzátor pro systém Linux.

Příloha B

Manuál

Implementace byla provedena na systému Linux, pro který je také určen tento manuál.

Příprava nástroje ANTLR

1. Nainstalujte Javu (verze 1.7 nebo vyšší)
2. Exportujte ANTLR runtime knihovnu (antlr/antlr-4.9.2-complete.jar) do systémových proměnných
3. Nainstalujte GNU Make

Dodatečné informace lze nalézt zde:
https://github.com/antlr/antlr4/blob/master/doc/getting-started.md

Příprava nástroje GNU Bison

1. Nainstalujte překladač GNU Compiler Collection (GCC)
2. Nainstalujte GNU Bison
3. Nainstalujte Flex
4. Nainstalujte GNU Make

Překlad a spuštění

V každém adresáři se nachází soubor Makefile, který umožňuje použití těchto příkazů:

- `make compile` – vygeneruje a přeloží syntaktický analyzátor
- `make run` – vygeneruje syntaktický analyzátor, přeloží jej a spustí
- `make clean` – smaže vygenerované a přeložené soubory

Příklady spuštění

Syntaktický analyzátor pro aritmetické výrazy vygenerovaný nástrojem ANTLR lze spustit následujícím způsobem:

```
$ cd antlr/expressions/bin
$ java ExpressionMain <../../../samples/expressions/test.expr
```

nebo

```
$ cd antlr/expressions
$ make run <../../samples/expressions/test.expr
```

Syntaktický analyzátor pro jazyk SubC vygenerovaný nástrojem GNU Bison lze spustit následujícím způsobem:

```
$ cd gnu-bison/sub-c
$ ./bin/sub_c <../../samples/sub-c/test.c
```

nebo

```
$ cd gnu-bison/sub-c
$ make run <../../samples/sub-c/test.c
```

Pro běh syntaktických analyzátorů vygenerovaných nástrojem ANTLR je nutné exportovat ANTLR runtime knihovnu do systémových proměnných.

```
$ cp antlr/antlr-4.9.2-complete.jar /usr/local/lib
$ export CLASSPATH=".:/usr/local/lib/antlr-4.9.2-complete.jar:$CLASSPATH"
```

Příloha C

Syntaktická pravidla jazyka SubC

1. Deklarace uživatelské funkce:

   typ id `(` *seznam_parametrů* `)` `;`

   Seznam parametrů je libovolná sekvence dvojic *typ id* oddělených čárkami.

2. Definice uživatelské funkce:

   typ id `(` *seznam_parametrů* `)` *složený_příkaz*

3. Deklarace proměnné:

   typ seznam_id `;`

   Identifikátory jsou odděleny čárkami.

4. Definice proměnné:

   typ seznam_přiřazení `;`

   Jednotlivá přiřazení jsou odděleny čárkami. Definice a deklarace proměnných se mohou kombinovat, tak jako v jazyce C.

5. Příkaz přiřazení:

   id `=` *výraz_nebo_přiřazení* `;`

   Přiřazení má pravou asociativitu.

6. Složený příkaz:

   `{` *seznam_příkazů* `}`

7. Podmíněný příkaz:

   `if (` *výraz* `)` *složený_příkaz* `else if (` *výraz* `)` *složený_příkaz* `else` *složený_příkaz*

   Za první podmínkou `if` se může nacházet libovolný počet podmínek `else if`. Na konci této sekvence může a nemusí být konstrukce `else`.

8. Cyklus:

   `for (` *příkaz_přiřazení* `;` *výraz* `;` *příkaz_přiřazení* `)` *složený_příkaz*

9. Volání uživatelem definované funkce:

   id `(` *seznam_výrazů* `)`

10. Příkaz návratu z funkce:
    `return` *výraz* `;`