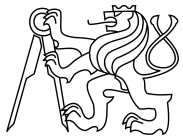
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Aplikace přepisovacích pravidel v teorii čísel
Student:	Bc. David Oppl
Vedoucí:	Ing. Daniel Dombek, Ph.D.
Studijní program:	Informatika
Studijní obor:	Systémové programování
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce zimního semestru 2018/19

Pokyny pro vypracování

O algebraickém číselném tělese řekneme, že je DUG (Distinct Unit Generated), pokud každý prvek z okruhu celých čísel v tělese lze vyjádřit konečnou sumou různých jednotek. To lze v některých případech ekvivalentně formulovat pomocí přepisování řetězců celých čísel na řetězce nad omezenou abecedou.

Ve zkoumané třídě těles existuje několik stále neověřených příkladů a není dopředu zřejmé, zda metoda přepisování řetězců povede k cíli. Tato diplomová práce má za cíl navázat na bakalářskou práci Davida Oppla ve snaze o jejich vyřešení.

Shrňte potřebnou teorii a známé výsledky na dané téma. Navrhněte program v jazyku C++, který funkčnost stávajícího nástroje rozšíří o práci s řetězci komplexních cifer a především o možnost parametrizace vstupu načtením sady přepisovacích pravidel z externího souboru. Pro potřeby Vaší implementace nastudujte potřebné pojmy z oblastí formálních jazyků, překladačů a interpretů. Analyzujte výpočetní náročnost tohoto postupu (ve smyslu asymptotického odhadu).

Seznam odborné literatury

Oppl, David. *Přepisování číselných řetězců a jeho aplikace v teorii čísel.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 12. června 2017

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky

Diplomová práce

Aplikace přepisovacích pravidel v teorii čísel

Bc. David Oppl

Vedoucí práce: Ing. Daniel Dombek, Ph.D.

8. května 2018

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Danielu Dombkovi, PhD. za jeho ochotu, přínosné rady, připomínky a odbornou pomoc při tvorbě této diplomové práce. Dále děkuji své rodině za jejich podporu po celou dobu mého studia. Také děkuji svým rukám, že jsou vždy po mém boku a prstům, že s nimi mohu kdykoli počítat. Dále děkuji své páteři, že mi vždy kryje záda.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 8. května 2018 .....

Odkaz na tuto práci

Oppl, David. *Aplikace přepisovacích pravidel v teorii čísel.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Tato diplomová práce se zabývá jazykem přepisovacích pravidel, speciálních zobrazení na množině nekonečných slov. Práce obsahuje přehled základních pojmů z teorie formálních jazyků a teorie čísel. Následně je čtenáři uvedena do souvislosti DUG vlastnost algebraických číselných těles s přepisovacími pravidly. Následuje popis činnosti překladačů. Výstupem implementační části je program pro přepisování číselných řetězců s možností parametrizace vstupu pomocí vlastního interpretu přepisovacích pravidel.

Klíčová slova přepisovací pravidlo, teorie čísel, DUG vlastnost, formální jazyky, překladač, interpret, parser, C++

Abstract

This master's thesis deals with the language of the so-called rewriting rules, special maps over the set of infinite words. This thesis contains summary on the topic of formal languages and number theory. Furthermore, the reader is apprised with the DUG property of algebraic number fields and its connection with rewriting rules. The reader is given brief overview of the compilation process. The output of the implementation part is application for rewriting digit string with input parametrized by the custom interpreter of user-defined rewriting rules.

Keywords rewriting rule, number theory, DUG property, formal languages, compiler, interpreter, parser, C++

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V této diplomové práci se budeme věnovat jazyku tzv. přepisovacích pravidel, speciálních zobrazení na množině nekonečných slov. Dalším předmětem práce jsou formální překladače a interprety, s jejichž využitím navrhneme a implementujeme užitečný program, který bude pomocí přepisovacích pravidel přepisovat číselné řetězce.

V první části se seznámíme se základními pojmy z kombinatoriky na slovech, formálních jazyků, teorie čísel, reprezentace v bázi. Znalost těchto pojmů je důležitá pro pochopení dalších částí této práce.

Ve druhé části se budeme zabývat algebraickými číselnými tělesy, v nichž lze každé celé číslo zapsat jako konečný součet různých algebraických jednotek. Tato tělesa se nazývají DUG (anglicky distinct unit generated). V této části shrneme dosavadní výsledky na téma klasifikace DUG těles a ukážeme si, jaké předpoklady musí být splněny, pokud se budeme pokoušet ověřovat DUG vlastnost u dosud neověřených případů pomocí kombinatorického přístupu.

Třetí část je věnována překladačům. Nejdříve si popíšeme jednotlivé typy překladačů a následně se zaměříme na činnost jednotlivých částí frontendu překladače.

V dalších dvou částech shrneme aktuální stav programu pro přepisování číselných řetězců a stanovíme si požadavky na implementovaný program. Také se zaměříme na návrh implementovaného programu, jenž bude obsahovat interpret jazyka přepisovacích pravidel. Za využití poznatků o překladačích ze třetí části této práce navrhneme, jak lze takový program implementovat.

V poslední části popíšeme implementovaný program, jenž je výstupem této práce. Nejdříve shrneme použité třídy a funkce, následuje popis ovládání programu a na závěr odvodíme asymptotickou časovou složitost tohoto programu.

Kapitola 1

Základní pojmy

Tato kapitola obsahuje definice základních pojmů, které jsou důležité pro tuto práci. Většina definic z teorie čísel byla převzata ze skript Teorie čísel [1]. Definice týkající se formálních jazyků byly sepsány na základě skript Jazyky a překlady [2] a knihy Combinatorics on words [3]. Teorie týkající se pozičních reprezentací čísel je citována z Algebraic combinatorics on words [4] a případné další zdroje jsou uvedeny explicitně.

1.1 Kombinatorika na slovech

Množinu $\mathcal{A}$ nazveme **abecedou** a její prvky symboly. Pro zjednodušení uvažujme $\mathcal{A} \subseteq \mathbb{Z}$. Posloupnost symbolů abecedy nazveme **řetězcem** nad danou abecedou. **Prázdným řetězcem** rozumíme prázdnou posloupnost symbolů a značíme písmenem ε. Množinu všech konečných řetězců označíme $\mathcal{A}^*$. Množinu všech konečných neprázdných řetězců nad abecedou $\mathcal{A}$ označíme $\mathcal{A}^+$.

Délkou řetězce $w = w_1 w_2 \dots w_n$ rozumíme počet symbolů abecedy $\mathcal{A}$, tedy

$$|w| = |w_1 w_2 \dots w_n| = n, \qquad \text{kde } w_1, w_2, \dots, w_n \neq \varepsilon.$$

Délka prázdného řetězce je 0.

Na množině řetězců $\mathcal{A}^*$ je definována operace **zřetězení** následovně: Nechť $w_1 = a_1 a_2 \dots a_n \in \mathcal{A}^*$ a $w_2 = b_1 b_2 \dots b_m \in \mathcal{A}^*$. Řetězec w nazveme jejich zřetězením, pokud

$$w = w_1 w_2 = a_1 a_2 \dots a_n b_1 b_2 \dots b_m.$$

Prázdný řetězec $\varepsilon \in \mathcal{A}^*$ je neutrálním prvkem k operaci zřetězení, pro každý řetězec $w \in \mathcal{A}^*$ tedy platí

$$\varepsilon w = w\varepsilon = w.$$

Faktor řetězce $w = w_1 w_2 \dots w_n$ je řetězec $\hat{w} = w_i w_{i+1} \dots w_{m+i}$, kde $i \geq 0$ a $m + i \leq n$. Pokud $i = 1$, řetězec $\hat{w}$ nazveme **prefixem** řetězce w. Je-li $m + i = n$, řetězec $\hat{w}$ je **sufixem** řetězce w. Jestliže se prefixy a sufixy nerovnají původnímu řetězci, nazýváme je vlastní.

Množinou $\mathcal{A}^{\mathbb{N}}$ označíme všechny **nekonečné řetězce** nad abecedou $\mathcal{A}$, tedy řetězce ve tvaru $w = w_1 w_2 w_3 \cdots$. U nekonečných řetězců můžeme určit faktor, prefix a sufix obdobně jako u konečných slov, avšak nedefinují se konečný sufix, nekonečný prefix a nekonečný faktor, který není zároveň sufixem.

Pro porovnání dvou nekonečných řetězců nad uspořádanou množinou symbolů A využijeme **lexikografické uspořádání** $\preceq_{lex}$, které definujeme takto: Nechť $w = w_1 w_2 \dots, v = v_1 v_2 \cdots \in \mathcal{A}^{\mathbb{N}}$. Řetězec w je lexikograficky menší nebo roven v, značeno $w \preceq_{lex} v$, právě tehdy, když platí

$$w = v \quad \text{nebo} \quad w_k < v_k \text{ pro } k = min\{i \geq 1 | w_i \neq v_i\}.$$

1.2 Formální jazyky

Tato kapitola objasňuje a formalizuje základní pojmy z teorie formálních jazyků. Tato problematika je zásadní pro pochopení fungování překladačů a interpretů programovacích jazyků. Definice a pojmy vysvětlené v této kapitole pochází především z [2].

Jazykem nad abecedou $\mathcal{A}$ značíme libovolnou množinu konečných řetězců nad $\mathcal{A}$. Tato podmnožina $\mathcal{A}^*$ může být konečná i nekonečná. Prázdná množina je jazykem nad jakoukoli abecedou. Příkladem nekonečného jazyka nad abecedou $\{0, 1, \dots, 9\}$ je množina všech nezáporných celých čísel. Příkladem konečného jazyka nad stejnou abecedou je množina všech prvočísel menších než 100. Řetězce z daného jazyka někdy nazýváme **slova**.

Jazyky lze kromě množinového zápisu reprezentovat několika způsoby. Konečné a ne příliš obsáhlé jazyky lze zapsat pomocí výčtu všech slov. Pro popis nekonečných a obsáhlých jazyků lze využít regulární výrazy, automaty a gramatiky. Pro potřeby této práce je postačující reprezentace jazyků pomocí automatů (konečných a zásobníkových) a gramatik.

1.2.1 Operace nad jazyky

S formálními jazyky lze provádět množinové operace sjednocení, průnik a rozdíl. Dále jazyk L_2 nad $\mathcal{A}$ nazveme **doplňkem** jazyka L_1 nad $\mathcal{A}$, pokud platí $L_1 \cup L_2 = \mathcal{A}^*$ a $L_1 \cap L_2 \neq \emptyset$.

Kromě množinových operací jsou definovány operace zřetězení, mocnina a iterace, které mají smysl pouze pro jazyky. **Zřetězení** jazyků L_1 a L_2 nad abecedou $\mathcal{A}$ je jazyk $L = L_1.L_2 = \{xy : x \in L_1, y \in L_2\}$. **N-tá mocnina** jazyka L je definována rekurzivním vztahem $L^n = L.L^{n-1}$ s počáteční podmínkou $L^0 = \{\epsilon\}$. **Iterace** L^* jazyka L je definována jako $L^* = \bigcup_{n=0}^{\infty} L^n$ a **pozitivní iterace** jako $L^* = \bigcup_{n=1}^{\infty} L^n$.

1.2.2 Automaty

Automat lze chápat jako systém, který pro každé vstupní slovo rozhodne, zda do daného jazyka patří (přijetí) nebo ne (nepřijetí). Základními typy automatů jsou konečný automat, zásobníkový automaty, Turingův stroj a lineárně omezený Turingův stroj.

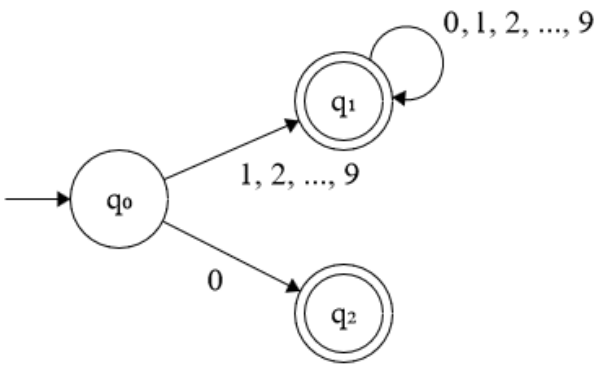
Konečný automat obsahuje čtecí hlavu, vstupní pásku se zadaným vstupem a řídící jednotku, která umožňuje systému přecházet mezi jednotlivými stavy, kterých je konečný počet. Konečný automat začne svůj výpočet v počátečním stavu a postupným načítáním vstupu mění svůj stav. Pokud je celý vstup načten a automat se nachází v konečném stavu, je slovo přijato, v opačném případě slovo není přijato.

Formálně je konečný automat uspořádaná pětice $(Q, \Sigma, \delta, q_0, F)$, kde Q je konečná množina stavů automatu, Σ konečná vstupní abeceda, δ **přechodová funkce**, $q_0 \in Q$ počáteční stav a $F \subseteq Q$ množina koncových stavů. U **deterministického konečného automatu** je přechodová funkce definována jako zobrazení z $Q \times \Sigma$ do Q. V každém stavu a daném symbolu na vstupu je přesně určeno, do jakého stavu se automat načtením symbolu dostane. Přechodová funkce nemusí být totální zobrazení. Pokud není funkce pro daný stav a daný symbol na vstupu definována, dojde k signalizaci chyby. U **nedeterministického konečného automatu** je přechodová funkce δ definována jako zobrazení z $Q \times \Sigma$ do množiny všech podmnožin Q (značeno 2^Q). Stejně jako u deterministického konečného automatu přechodová funkce nemusí být totální zobrazení. U nedeterministického automatu nemusí být vždy jednoznačné, do kterého konkrétního stavu se automat načtením vstupu dostane. Nastane-li během výpočtu situace, že přechod do dalšího stavu není jednoznačný, automat přejde současně do všech stavů zadaných přechodovou funkcí a každé takto vzniklé výpočetní větvi pokračuje čtením dalšího vstupu. Pokud dojde alespoň v jedné větvi výpočtu k přijetí slova, je slovo přijato. Každý nedeterministický konečný automat lze převést na ekvivalentní deterministický konečný automat, jak je popsáno v [2]. Výpočetní síla deterministických a nedeterministických konečných automatů je tak stejná. Přijímá-li automat všechna slova z daného jazyka a ostatní slova odmítne, říkáme, že automat přijímá daný jazyk. Konečné automaty přijímají regulární jazyky.

Příklad 1.1. *Mějme jazyk L nad abecedou $\mathcal{A} = \{0, 1, 2, \ldots, 9\}$, který obsahuje všechna nezáporná celá čísla. Konečný automat přijímající tento jazyk lze vytvořit jako pětici $K = (\{q0, q1, q2\}, \mathcal{A}, \delta, q_0, \{q_1, q_2\})$, kde δ je definováno následovně:*

$$\delta(q_0, 0) = q_2,$$
$$\delta(q_0, 1) = q_1, \delta(q_0, 2) = q_1, \cdots, \delta(q_0, 9) = q_1,$$
$$\delta(q_1, 0) = q_1, \delta(q_1, 1) = q_1, \cdots, \delta(q_1, 9) = q_1$$

Tento automat lze popsat i graficky, jako je uvedeno na obrázku 1.1.



Obrázek 1.1: Automat přijímající všechna nezáporná celá čísla nad abecedou $\mathcal{A} = \{0, 1, \ldots, 9\}$

V grafickém zápisu automatu jsou přechody reprezentovány pomocí šipek s přidruženými vstupními symboly. Stavy automatu jsou označeny kolečky. Dvojité kolečko reprezentuje koncový stav. Počáteční stav je označen šipkou směřující z volného prostoru k počátečnímu stavu.

Existují však jazyky, které nelze přijmout konečným automatem. Příkladem může být jazyk nad abecedou $\{0, 1\}$ obsahující slova se stejným počtem 0 a 1. Pro takové případy existuje **zásobníkový automat**. Ten, stejně jako konečný automat, obsahuje čtecí hlavu, vstupní pásku se vstupním slovem a řídící jednotku, navíc však využívá **zásobník** – přídavnou paměť typu LIFO. K přechodům mezi stavy dochází nejen v závislosti symbolu na vstupu, ale i podle aktuálního stavu zásobníku. K přijetí slova dojde, pokud se automat nachází v koncovém stavu nebo pokud je na konci zásobník prázdný. Oba způsoby přijetí jsou na sebe navzájem převeditelné a výpočetně jsou tedy ekvivalentní.

Formálně je zásobníkový automat uspořádaná sedmice $(Q, \Sigma, G, \delta, q_0, Z_0, F)$, kde Q je konečná množina vnitřních stavů, Σ konečná vstupní abeceda, G konečná abeceda zásobníku, δ zobrazení z konečné podmnožiny $Q \times (\Sigma \cup \{\epsilon\}) \times G^*$ do množiny konečných podmnožin $Q \times G^*$, $q_0 \in$ počáteční stav, $Z_0 \in G$ počáteční symbol na zásobníku a $F \subseteq Q$. Variantou zásobníkového automatu je **deterministický zásobníkový automat**. U takového automatu existuje v každé situaci maximálně jedna operace, kterou může provést. Formálně jde o zásobníkový automat $R = (Q, \Sigma, G, \delta, q_0, Z_0, F)$, pro který dále platí:

1. $|\delta(q, a, \gamma)| \leq 1, \forall q \in Q, \forall a \in (\Sigma \cup \{\epsilon\}), \forall \gamma \in G^*$
2. Pokud $\delta(q, a, \alpha) \neq \emptyset, \delta(q, a, \beta) \neq \emptyset$ pro $\alpha \neq \beta$, pak α není předponou β a β není předponou α (tzn. $\forall \gamma \in G^* : \alpha\gamma \neq \beta, \alpha \neq \beta\gamma$).

3. Pokud $\delta(q, a, \alpha) \neq \emptyset, \delta(q, \epsilon, \beta) \neq \emptyset$, pak α není předponou β a β není předponou α (tzn. $\forall\gamma \in G^* : \alpha\gamma \neq \beta, \alpha \neq \beta\gamma$).

Zásobníkové automaty přijímají bezkontextové jazyky. Na rozdíl od konečných automatů však nelze každý zásobníkový automat převést na ekvivalentní deterministický. Protože navíc deterministický zásobníkový automat již z definice splňuje definici zásobníkového automatu, je výpočetně slabší.

Existují jazyky, na které ani zásobníkové automaty nestačí, ty však pro potřeby této práce nejsou nezbytné.

1.2.3 Gramatiky

Druhým možným způsobem, jak popsat jazyk je pomocí gramatik. **Gramatika** je podle [2] čtveřice $G = (N, T, P, S)$, kde N je konečná množina **neterminálních symbolů**, T je konečná množina **terminálních symbolů** ($T \cap N = \emptyset$), P je konečná podmnožina $(N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$, $S \in N$ je **počáteční symbol** gramatiky. Element (α, β) z P se zapisuje $\alpha \to \beta$ a nazývá se **pravidlo**. Pokud gramatika obsahuje pravidla tvaru $\alpha \to \beta_1, \alpha \to \beta_2, \ldots, \alpha \to \beta_n$, lze tato pravidla se stejnou levou stranou zapisovat zkráceně jako $\alpha \to \beta_1|\beta_2|\ldots|\beta_n$. Terminální symboly budeme označovat malými písmeny a neterminální symboly velkými písmeny.

Pokud nebude uvedeno jinak, pomocí písmen malé řecké abecedy budeme zapisovat posloupnost terminálních a neterminálních symbolů.

Příklad 1.2. *Mějme jazyk L nad abecedou $\mathcal{A} = \{0, 1, \ldots, 9\}$ obsahující všechna nezáporná celá čísla. Gramatiku G popisující tento jazyk lze zapsat jako čtveřici $G = (\{S, A\}, \mathcal{A}, P, S)$, kde P obsahuje pravidla:*

$$S \to 0$$
$$S \to 1A|2A|3A|4A|5A|6A|7A|8A|9A$$
$$A \to 0A|1A|2A|3A|4A|5A|6A|7A|8A|9A|\epsilon$$

Každou posloupnost terminálů a neterminálů, kterou lze vytvořit postupným používáním pravidel na startovní neterminál nazýváme větnou formou. Posloupnost použitých pravidel nazýváme derivace a značíme pomocí symbolu $\Rightarrow^*$

Podle tvaru přepisovacích pravidel rozlišil Noam Chomský [5] čtyři základní typy gramatik a jazyků, které generují. Nechť $G = (N, T, P, S)$ je gramatika, potom řekneme, že G je:

1. **Neomezená** (typu 0), pokud splňuje definici gramatiky

2. **Kontextová** (typu 1), pokud každé pravidlo z P má tvar $\gamma A\delta \to \gamma\alpha\delta$, kde $\alpha, \gamma, \delta \in (N \cup T)^*, A \in N$

3. **Bezkontextová** (typu 2), pokud každé pravidlo z P má tvar $A \to \alpha$, kde $\alpha \in (N \cup T)^*, A \in N$

4. **Regulární** (typu 3), pokud každé pravidlo z P má tvar $A \to aB$ nebo $A \to a$, kde $A, B \in N, a \in T$. Výjimku tvoří pravidlo $S \to \epsilon$ v případě, že se S neobjeví na pravé straně žádného pravidla.

Zároveň platí, že regulární gramatiky jsou podmnožinou bezkontextových, bezkontextové podmnožinou kontextových a kontextové podmnožinou neomezených.

1.3 Teorie čísel

Množina G s asociativní binární operací $\circ : G \times G \to G$ se nazývá **grupa**, pokud:

- v G existuje neutrální prvek e, pro který platí $x \circ e = e \circ x = x$ pro každé $x \in G$,
- ke každému $x \in G$ existuje inverzní prvek $x \in G$, pro který platí $x \circ y = y \circ x = e$.

Pokud je operace $\circ$ komutativní, nazýváme G komutativní grupou.

Množina R se dvěma asociativními binárními operacemi $+, \times : R \times R \to R$ se nazývá **okruh**, pokud:

- R s operací $+$ je komutativní grupa s neutrálním prvkem 0,
- operace $\times$ je distributivní vůči $+$, tedy platí $x \times (y+z) = (x \times y) + (x \times z)$ a $(y + z) \times x = (y \times x) + (z \times x)$ pro všechna $x, y \in R$.

Okruh R s operacemi $+$ a $\times$ se nazývá **těleso**, pokud $R \setminus \{0\}$ s operací $\times$ je grupa. Je-li operace $\times$ navíc komutativní, nazveme R komutativním tělesem.

Nechť R je okruh. **Polynomem** nad R nazýváme výraz:

$$f(x) = a_0 + a_1 X + \cdots + a_{n-1} X^{n-1} + a_n X^n, \qquad a_0, ...a_n \in R$$

Polynom f je stupně $n \geq 0$, je-li n nejvyšší index, pro který platí $a_n \neq 0$. Pokud $a_i = 0$ pro všechna $i \geq 0$, f je nulový polynom a jeho stupeň je -1. Polynom f stupně n nazveme **monický**, pokud koeficient $a_n = 1$.

Nechť K je komutativní okruh. **Okruh polynomů** nad K označíme $K[X]$, kde X značí formální proměnnou, a definujeme jej následovně:

$$K[X] = \left\{ \sum_{i=0}^{m} c_i X^i \quad \middle| \quad m \geq 0, c_i \in K \right\}$$

Nechť L a K jsou komutativní tělesa a $K \subseteq L$. Je-li prvek $x \in L$ kořenem nějakého polynomu $f \in K[X]$, prvek x je **algebraický** nad K. Pro každé

takové x existuje monický polynom $f_0 \in K[X]$ nejmenšího stupně m, který nazýváme **minimálním polynomem** x nad K. **Stupeň** x nad K je definován jako stupeň polynomu f_0, tedy m. Jsou-li všechny prvky L algebraické nad K, L je nadtělesem K a K je podtělesem L. Na L lze nahlížet jako na vektorový prostor nad tělesem K. Dimenzi vektorového prostoru L nad K značíme $[L : K]$.

Číslo $\alpha \in \mathbb{C}$ je **algebraické číslo**, pokud existuje monický polynom $f \in \mathbb{Q}[X]$ takový, že $f(\alpha) = 0$.

Algebraické číslo $\alpha \in \mathbb{C}$ je **algebraické celé číslo**, pokud jeho minimální polynom f_0 je prvkem okruhu polynomů $\mathbb{Z}[X]$.

Pro libovolné $\alpha \in \mathbb{C}$ definujeme **algebraické číselné těleso** $\mathbb{Q}(\alpha)$ jako minimální podtěleso(ve smyslu inkluze) tělesa $\mathbb{C}$, které obsahuje $\mathbb{Q}$ a α. Je-li α algebraické nad $\mathbb{Q}$ konečného stupně n, lze $\mathbb{Q}(\alpha)$ zapsat následovně:

$$\mathbb{Q}(\alpha) = \{c_0 + c_1\alpha + c_2\alpha^2 + \cdots + c_{n-1}\alpha^{n-1} |\ c_i \in \mathbb{Q}\}$$

Stupeň generujícího prvku α nazýváme **stupněm tělesa** $\mathbb{Q}(\alpha)$.

Nechť K je těleso. Množinu O_K všech algebraických celých čísel obsažených v K nazýváme **okruh celých čísel** v K.

V každém algebraickém tělese $K = \mathbb{Q}(\alpha)$ stupně n s okruhem algebraických celých čísel O_K existuje uspořádaná množina $\{\beta_1, \ldots, \beta_n\}$, kterou nazýváme **integrální báze**. Její prvky jsou algebraická celá čísla a pro všechna $\beta \in O_K$ platí

$$\beta = \sum_{i=1}^{n} a_i\beta_i \quad a_i \in \mathbb{Z}.$$

Algebraické celé číslo $y \in O_K$ **dělí** $x \in O_K$, pokud existuje $z \in O_K$ takové, že $x = yz$. Pokud x dělí 1, čísla x a $1/x$ jsou **algebraické jednotky**. Množina všech algebraických jednotek tvoří multiplikativní grupu a značíme ji U_K.

Číslo $\zeta \in \mathbb{C}$ nazveme ***n*-tým kořenem jednotky**, pokud platí $\zeta^n = 1$. Pokud n je nejmenší celé číslo ≥ 1, pro které platí $\zeta^n = 1$, nazveme ζ **primitivním** n-tým kořenem jednotky.

Věta 1.3. (Dirichletova) Nechť $K = \mathbb{Q}(\alpha)$ je těleso a minimální polynom algebraického čísla α má s reálných a $2t$ nereálných (v komplexně sdružených párech) kořenů. $r = s + t - 1$ nazveme **unit rank**. Existuje primitivní kořen $\zeta \in K$ a jednotky $\eta_1, \ldots, \eta_r$ takové, že platí

$$U_k = \{\zeta^{j_0}\eta_1^{j_1} \cdots \eta_r^{j_r} \mid j_0, \ldots j_n \in \mathbb{Z}\}.$$

Množinu $\{\eta_1, \ldots, \eta_r\}$ nazýváme **fundamentální systém** jednotek tělesa K.

Příklad 1.4. *Nejjednodušším příkladem algebraického číselného tělesa je těleso racionálních čísel* $\mathbb{Q}$*. Okruh algebraických celých čísel v něm pak tvoří celá*

čísla. Platí tedy

$$O_{\mathbb{Q}} = \mathbb{Z} \qquad\qquad U_{\mathbb{Q}} = \{1, -1\}.$$

Dalším zajímavým příkladem je rozšíření racionálních čísel o větší z kořenů polynomu $f(X) = X^2 - X - 1$. *Kořeny tohoto polynomu jsou tzv. zlatý řez* $\tau = \frac{1+\sqrt{5}}{2}$ *a* $\tau' = \frac{1-\sqrt{5}}{2} = -\frac{1}{\tau}$. *Těleso* $K = \mathbb{Q}(\tau)$ *obsahuje všechna algebraická čísla ve tvaru* $a + b\frac{1+\sqrt{5}}{2}$, *kde* $a, b \in \mathbb{Q}$. *Okruh algebraických celých čísel* O_K *lze vyjádřit následovně:*

$$O_K = O_{\mathbb{Q}(\tau)} = \left\{ a + b\frac{1+\sqrt{5}}{2} \;\middle|\; a, b \in \mathbb{Z} \right\}$$

Pro grupu jednotek U_K *platí*

$$U_K = U_{\mathbb{Q}(\tau)} = \left\{ \pm \left(\frac{1+\sqrt{5}}{2} \right)^k \middle| k \in \mathbb{Z} \right\}$$

1.4 Reprezentace čísel v bázi

Každé číslo $x \in \mathbb{R}$ může být vyjádřeno v číselné soustavě o celočíselném základu $b > 1$ následovně:

$$x = \pm \left(d_k b^k + \cdots + d_1 b + d_0 + \frac{d_{-1}}{b} + \frac{d_{-2}}{b^2} + \dots \right), \text{ kde } d_i \in \{0, 1, \dots, b-1\}$$

Nekonečnou posloupnost symbolů $d_k d_{k-1} \cdots$ zapíšeme jako nekonečné slovo $d(x) \in \mathcal{A}^{\mathbb{N}}$ takto

$$d(x) = \begin{cases} d_k \cdots d_0 \bullet d_{-1} \cdots & \text{pokud } k \geq 0, \\ 0 \bullet 0^{-k-1} d_k d_{k-1} \cdots & \text{pokud } k < 0, \end{cases}$$

kde zlomková tečka $\bullet$ odděluje koeficienty u nezáporných a záporných mocnin b, tzv. **celou** a **zlomkovou část** reprezentace $d(x)$. Řetězec $d(x)$ nazýváme **reprezentací** čísla x v číselné soustavě o základu b. Má-li řetězec $d(x)$ nekonečný sufix 0^ω, nepíšeme jej (celou část reprezentace $d(x)$ však zapíšeme celou) a řekneme, že x má **konečnou reprezentaci** $d(x)$. Mezi všemi reprezentacemi $x \in \mathbb{R} \setminus \{0\}$ v celočíselné bázi $b \geq 2$ existuje vždy jedna reprezentace bez sufixu $(b-1)^\omega$, tzv. **přípustná**.

Poznámka 1.5. *Poznamenejme, že se v této práci odchylujeme od tradičního zápisu konečných i nekonečných slov s indexací od jedničky* $(x_1 x_2 x_3 \cdots)$, *případně od nuly. Pro znázornění vztahu slov nad abecedou a reprezentací čísel v bázi bude vhodnější často použít sestupnou indexaci* $x_k \cdots x_0 \bullet x_{-1} \cdots$. *Pokud nebude nutné použít v zápisu zlomkovou tečku* $\bullet$, *vynecháme ji.*

Poznámka 1.6. *Vztah* $d(x) = d_k \cdots d_0 \bullet d_{-1} \cdots$ *lze zapisovat jako* $x = d_k \cdots d_0 \bullet_b d_{-1} \cdots$*, případně* $d(x) = d_k \cdots d_0\, b$*, pokud je zlomková část* $d(x)$ *nulová.*

Tento způsob zápisu lze rozšířit i pro soustavy s neceločíselným základem $\beta > 1$. Řetězec d se v takovém případě označuje jako β-reprezentace. Pro každé reálné číslo existuje alespoň jedna β-reprezentace. V soustavách se základem $b \in \mathbb{Z}$ nejsou některé nekonečné reprezentace přípustné, ale všechny konečné reprezentace již přípustné jsou. Naproti tomu v soustavách s obecným základem $\beta > 1$ je situace složitější a konečnost reprezentace vůbec nezaručuje její přípustnost nebo jednoznačnost.

Příklad 1.7. *Uvažujme číselnou soustavu o základu* $\beta = \tau = \frac{1+\sqrt{5}}{2}$*. Pro čísla* $x = 11_\tau$ *a* $y = 100_\tau$ *snadno ukážeme, že se rovnají*

$$x = 11_\tau = \frac{1+\sqrt{5}}{2} + 1 = \frac{3+\sqrt{5}}{2}$$
$$y = 100_\tau = \left(\frac{1+\sqrt{5}}{2}\right)^2 = \frac{3+\sqrt{5}}{2}$$

I přes nejednoznačnost zápisu některých čísel v soustavách o neceločíselném základu lze podle [4] pro každé číslo $x \in \mathbb{R}_0^+$ nalézt tzv. hladovým algoritmem jeho unikátní, lexikograficky největší β-reprezentaci $d = d_k d_{k-1} \cdots$. Algoritmus najde největší k splňující podmínku $\beta^k \leq x$ a následně největší d_k splňující $d_k \beta^k \leq x$. V každém dalším kroku j je nalezeno největší d_j, které splňuje $d_j \beta^j + \sum_{i=j+1}^{k} d_i \beta^i \leq x$. Je tedy patrné, že výstupem je nekonečné slovo - lexikograficky největší β-reprezentace čísla x, tzv. **β-rozvoj** čísla x. β-rozvoj je prvkem $\mathcal{A}^{\mathbb{N}}$, kde $\mathcal{A} = \{0, 1, \ldots, \lceil \beta \rceil - 1\}$ je **kanonická abeceda**. β-rozvoj záporného čísla x se definuje jako β-rozvoj $|x|$, jehož nejlevější nenulový symbol je doplněn o znaménko $-$.

Algoritmus 1.1 Nalezení β-rozvoje čísla $x \geq 0$

Vstup: x,
1: základ číselné soustavy β
Výstup: $d = d_k d_{k-1} \cdots d_0 \bullet d_{-1} \cdots$
2: $k \leftarrow \lfloor log_\beta x \rfloor$
3: $d_k \leftarrow \lfloor x/\beta^k \rfloor$
4: $r_k \leftarrow x/\beta^k - d_k$
5: **for** $j = k - 1$ to $-\infty$ **do**
6: $\quad d_j \leftarrow \lfloor \beta r_{j+1} \rfloor$
7: $\quad r_j \leftarrow \beta r_{j+1} - d_j$
8: **end for**
9: **return** d

β-rozvoj reálných čísel lze ekvivalentně definovat pomocí iterací jisté transformace jednotkového intervalu. Tento způsob pochází z práce A. Rényiho [6]:

Věta 1.8. *Nechť je zobrazení* $T_\beta : [0,1) \to [0,1)$ *definováno jako*

$$T_\beta : x \to \beta x - \lfloor \beta x \rfloor,$$

kde $\lfloor y \rfloor$ *značí dolní celou část* y.

Každému $x \in [0,1)$ *lze přiřadit řetězec* $d_\beta(x) = x_1 x_2 \cdots$ *předpisem*

$$x_i = \lfloor \beta T^{i-1}(x) \rfloor, \quad \text{pro každé } i \geq 1.$$

Řetězec $d_\beta(x)$ *pak splňuje* $x = \sum_{i \geq 1} \frac{x_i}{\beta^i}$.

S využitím předchozí věty lze přirozeně rozšířit definici řetězců $d_\beta(x)$ na všechna reálná čísla. Pro každé kladné $x \in \mathbb{R}$ zvolíme nejmenší index k, pro který platí

$$\frac{1}{\beta^k} x \in [0,1).$$

Dle Věty 1.8 existuje řetězec $d_\beta\left(\frac{x}{\beta^k}\right)$. Označíme-li cifry $d_\beta\left(\frac{x}{\beta^k}\right)$ následovně

$$d_\beta\left(\frac{x}{\beta^k}\right) = d_{k-1} d_{k-2} \cdots ,$$

platí $\frac{x}{\beta^k} = \sum_{i \geq 1} d_{k-i} \beta^i$, tedy $x = \sum_{j \leq k-1} d_j \beta^j$ a **β-rozvojem** čísla x nazveme řetězec

$$x = x_\beta = \begin{cases} 0 \bullet d_\beta(x) & \text{pokud } k = 0, \\ d_{k-1} d_{k-2} \cdots d_0 \bullet d_{-1} \cdots & \text{pokud } k > 0, \end{cases}$$

přičemž používáme zkrácený zápis jako v Poznámce 1.6.

V této práci budeme pracovat s podobným, ale obecnějším typem reprezentací, s obecně komplexním základem a různými volbami pro abecedu použitých cifer. Existenci takových reprezentací zaručuje následující věta z práce W. P. Thurstona [7]:

Věta 1.9. *Nechť* $\alpha \in \mathbb{C}$ *a* $|\alpha| > 1$. *Pokud daná abeceda* $\mathcal{A} \subseteq \mathbb{C}$ *a množina* $V \subseteq \mathbb{C}$ *splňují*

$$\alpha V \subseteq \bigcup_{\alpha \in \mathcal{A}} (V + \alpha), \tag{1.1}$$

potom pro všechna $z \in V$ *existuje takový řetězec* $a_1 a_2 \cdots \in \mathcal{A}^{\mathbb{N}}$, *že*

$$z = \frac{a_1}{\alpha} + \frac{a_2}{\alpha^2} + \ldots .$$

Pokud 0 *je prvkem vnitřku množiny* V, *každé* $z \in \mathbb{C}$ *lze zapsat ve tvaru*

$$z = d_k \alpha^k + \cdots + d_1 \alpha + d_0 + \frac{d_{-1}}{\alpha} + \frac{d_{-2}}{\alpha^2} + \ldots$$

kde $k \in \mathbb{Z}$, $d_i \in \mathcal{A}$ *a* $d_k \neq 0$.

Řetězce $d_k d_{k-1} \cdots \in \mathcal{A}^{\mathbb{N}}$ z druhé části Věty 1.9 nazveme **α-reprezentacemi nad abecedou $\mathcal{A}$** a budeme je zapisovat podobně jako výše,

$$x = \begin{cases} d_k \cdots d_0 \bullet d_{-1} \cdots & \text{pokud } k \geq 0, \\ 0 \bullet 0^{-k-1} d_k d_{k-1} \cdots & \text{pokud } k < 0. \end{cases}$$

Poznamenejme, že podmínku 1.1 z Věty 1.9 nebude třeba v konkrétních případech ověřovat. Pro ilustraci lze uvést, že definice Rényiho β-rozvojů spadá pod speciální případ věty 1.9, zvolíme-li bázi $\alpha = \beta - 1$, jednotkový interval $V = [0, 1)$ a kanonickou abecedu $\{0, 1, \ldots, \lceil \beta \rceil - 1\}$.

Zdůrazněme, že u obecnějších reprezentací nebudeme používat termín rozvoj, neboť tam je problematika jednoznačnosti poněkud komplikovanější.

V rámci této práce se také budeme setkávat s pojmem přepisovací pravidlo.

Definice 1.10. *Nechť $\alpha \in \mathbb{C}, |\alpha| > 1$, je kořenem polynomu $p \in \mathbb{Z}[X]$, ne nutně minimálního,*

$$p(x) = a_m X^m + a_{m-1} X^{m-1} + \cdots + a_1 X + a_0.$$

*Konečné slovo $a_m a_{m-1} \cdots a_0 \in \mathbb{Z}^*$ nazveme **přepisovacím pravidlem** v bázi α.*

*Nechť $x_k x_{k-1} \cdots x_0 \bullet x_{-1} \cdots$ je α-reprezentace nějakého $x \in \mathbb{C}$. Uvažujme, že $x_{k+1} = x_{k+2} = \cdots = 0$. **Aplikací přepisovacího pravidla** $a_m a_{m-1} \cdots a_0$ na pozici $i \in \mathbb{Z}$ rozumíme přiřazení $x_k x_{k-1} \cdots \to z_l z_{l-1} \cdots$ kde*

$$z_j = \begin{cases} x_j + a_{j-i} & \text{pokud } j \in \{i, i+1, \ldots, i+m\}, \\ x_j & \text{jinak.} \end{cases}$$

Aplikaci přepisovacího pravidla budeme znázorňovat takto:

			x_k	x_{k-1}	$\cdots$	x_i	$\cdots$	x_1	x_0	$\bullet$	x_{-1}	$\cdots$
$+$	a_m	a_{m-1}			$\cdots$	a_0						
	z_l	z_{l-1}			$\cdots$	z_i	$\cdots$	z_1	z_0	$\bullet$	z_{-1}	$\cdots$

Z definice přepisovacího pravidla v bázi α je zřejmé, že aplikace přepisovacího pravidla pouze převádí reprezentace čísel v bázi α na jiné, přičemž se může změnit abeceda použitých cifer.

Z důvodu přehlednosti zapisujeme záporné cifry pomocí symbolu $\bar{}$ nad danou cifrou.

Příklad 1.11. *Přepisovací pravidla lze využít například při normalizaci po sčítání. Uvažujme čísla $x = 6185$ a $y = 98665$ zapsaná v desítkové soustavě (tedy báze $\beta = 10$). Nechť $z = x + y$.*

Součet znázorníme následovně:

$x =$		6	1	8	5	$\bullet$
$y =$	9	8	6	6	5	$\bullet$
$z =$	9	14	7	14	10	$\bullet$

Všechna z_i, která nejsou součástí abecedy $\{0,\ldots,10\}$, musíme upravit. Úpravu provedeme tak, že několikrát aplikujeme přepisovací pravidlo $\pm(1\bar{\beta})$, tj.$\pm(1(\overline{10}))$ na pozici i tak, aby symbol $z_{i+1}+10$ (resp.$z_{i+1}-10$) náležel abecedě $\{0,1,\ldots,9\}$.

	$z=$		9	14	7	14	10	$\bullet$
$+$	$0=$	1	$\overline{10}$					
$+$	$0=$		1	$\overline{10}$				
$+$	$0=$				1	$\overline{10}$		
$+$	$0=$					1	$\overline{10}$	$\bullet$
	$z=$	1	0	4	8	5	0	$\bullet$

Výsledkem sčítání je $z=104850\bullet\in\{0,1,\ldots,9\}^*$.

Kapitola 2

DUG vlastnost algebraických těles

Některá tělesa mají vlastnost, že v nich lze každé algebraické celé číslo zapsat jako konečný součet různých jednotek. Tato tělesa nazýváme **DUG** (anglicky **distinct unit generated**). Této vlastnosti si všiml B. Jacobson [8] a dokázal ji u těles $\mathbb{Q}(\sqrt{2})$ a $\mathbb{Q}(\sqrt{5})$. Domníval se, že žádná další kvadratická tělesa tuto vlastnost nesplňují. Jeho hypotézu pak potvrdil J.Śliwa [9] a dokázal, že žádné čistě kubické těleso, tedy těleso $\mathbb{Q}(\sqrt[3]{n})$, pro $n \in \mathbb{Z}$ a $\sqrt[3]{n} \notin \mathbb{Z}$, není DUG. Další výsledky rozšiřující klasifikaci DUG vlastností u těles nižších stupňů lze nalézt v [10] [11] [12].

Následující definice pochází z práce J. Thuswaldnera a V. Zieglera [12] a určuje, jak vzdálené je dané těleso DUG vlastnosti.

Definice 2.1. *Nechť O_K je okruh celých čísel v tělese K a $\alpha \in O_K$. Předpokládejme, že α lze zapsat jako lineární kombinaci jednotek $\epsilon_1, \ldots, \epsilon_l \in U_K$*

$$\alpha = a_1\epsilon_1 + \cdots + a_\ell\epsilon_\ell,$$

*kde $a_1 \geq \cdots \geq a_\ell \geq 1$ jsou celá čísla. Zvolíme-li reprezentaci s nejmenším možným a_1, potom $\omega(\alpha) = a_1$ je tzv. **jednotková výška** (anglicky **unit sum height**) α. Jestliže $\alpha = 0$, pak $\omega(\alpha) = \omega(0) = 0$. Pokud α nelze zapsat jako lineární kombinaci jednotek, $\omega(\alpha) = \infty$. Jednotkovou výšku tělesa K definujeme předpisem*

$$\omega(K) = max\,\{\omega(\alpha) \,|\, \alpha \in O_K\}$$

Je-li $\omega(K) = 1$, těleso K je DUG.

Nejaktuálnější výsledek na téma klasifikace DUG těles pochází z prací [13] [14]. Ten je ve formě seznamu obsahujícího všechna DUG tělesa čtvrtého stupně, která jsou současně tzv. **plně komplexní**, anglicky **totally complex** (to jsou taková tělesa $\mathbb{Q}(\gamma)$, kde minimální polynom čísla γ nemá žádné reálné kořeny). Doposud však nebylo dokázáno, že všechna tělesa z daného seznamu DUG vlastnost opravdu splňují.

Věta 2.2. *[13] Každé plně komplexní těleso čtvrtého stupně s DUG vlastností je nutně jedno z těles v následujícím seznamu (ζ_n značí primitivní n-tý kořen z jednotky).*

- $\mathbb{Q}(\zeta_\mu)$, *kde* $\mu = 5, 8, 12$,
- $\mathbb{Q}(\gamma)$, *kde* γ *je kořenem jednoho z polynomů* $X^4 - X + 1$, $X^4 + 2X^2 - 2X + 1^{\triangle}$, $X^4 + X^2 - X + 1$, $X^4 - X^3 + X + 1^{\triangledown}$, $X^4 - X^3 + X^2 + X + 1^{\triangledown}$, $X^4 - X^3 + 2X^2 - X + 2^{\triangle}$,
- $\mathbb{Q}(\sqrt{a + ib})$ *pro* $(a, b) = (1, 1), (1, 2), (1, 4), (7, 4)^{\triangle}$,
- $\mathbb{Q}(\sqrt{a + \zeta_3 b})$ *pro* $(a, b) = (2, 1)$, $(4, 1)$, $(8, 1)$, $(3, 2)$, $(4, 3)$, $(7, 3)$, $(11, 3)$, $(5, 4)$, $(9, 4)$, $(13, 4)$, $(12, 5)$, $(11, 7)$, $(9, 8)$, $(15, 11)$, $(19, 11)^{\triangle}$, $(17, 12)^{\triangle}$, $(17, 16)^{\triangle}$,
- $\mathbb{Q}(\zeta_3, \sqrt{d})$ *pro* $d = 5, 6, 21$,
- $\mathbb{Q}(\zeta_4, \sqrt{5})$,
- $\mathbb{Q}\left(\sqrt{-1 - \sqrt{2}}\right)$ *nebo* $\mathbb{Q}\left(\sqrt{-\frac{1+\sqrt{5}}{2}}\right)$.

Věta 2.3. *[14] Pokud je K plně komplexní těleso čtvrtého stupně ze seznamu ve Větě 2.2, váha tohoto tělesa splňuje $\omega(K) \leq 3$. Tělesa označená symbolem $\triangle$ navíc splňují $\omega(K) \leq 2$. Jednotková výška těles označených $\triangledown$ splňuje pouze podmínku $\omega(K) \leq 3$. Všechna ostatní tělesa jsou DUG, tedy $\omega(K) = 1$.*

Součástí této práce je rozšíření programu implementovaného v [15], který přepisuje reprezentace čísel v bázi γ, která je kořenem polynomu $X^4 - X + 1$. Daný program tak reflektuje důkaz DUG vlastnosti u tělesa $\mathbb{Q}(\gamma)$ uvedený v [16]. Jde o první těleso, u kterého byla DUG vlastnost dokázána kombinatoricky.

Nechť $K = \mathbb{Q}(\gamma)$, kde γ je libovolným kořenem polynomu $X^4 - X + 1$. Kořeny tohoto polynomu jsou čtyři nereálná čísla ve dvou komplexně sdružených párech. Označíme je $\gamma_1, \bar{\gamma}_1$ a $\gamma_2, \bar{\gamma}_2$, přičemž platí $|\gamma_1| > 1 > |\gamma_2|$. Zvolíme-li $\gamma \in \{\gamma_1, \bar{\gamma}_1\}$, má smysl uvažovat obecné reprezentace čísel v komplexní bázi γ. Pokud zvolíme $\gamma \in \{\gamma_2, \bar{\gamma}_2\}$ v absolutní hodnotě menší než jedna, můžeme využít faktu uvedeného v [1], že pro každé algebraické číslo $\gamma \neq 0$ platí $\mathbb{Q}(\gamma) = \mathbb{Q}\left(\frac{1}{\gamma}\right)$, a zvolit $\gamma \in \left\{\frac{1}{\gamma_2}, \frac{1}{\bar{\gamma}_2}\right\}$ v absolutní hodnotě větší než jedna. Bez újmy na obecnosti tedy můžeme dále předpokládat, že $|\gamma| > 1$.

Množina $\{1, \gamma, \gamma^2, \gamma^3\}$ tvoří integrální bázi okruhu celých čísel

$$O_K = \left\{ \sum_{n=-\infty}^{\infty} u_n \gamma^n \right\},$$

pro $u_n \in \mathbb{Z}$ a $u_n \neq 0$ pro maximálně konečně mnoho indexů n. Ve smyslu Věty 1.9 můžeme zapsat všechna celá čísla $\alpha \in O_K$ jako γ-reprezentace ve tvaru

$$\alpha = \cdots u_1 u_0 \bullet u_{-1} u_{-2} \cdots ,$$

nad abecedou $\mathbb{Z}$. Bez újmy na obecnosti však můžeme uvažovat pouze konečné γ-reprezentace, tedy reprezentace ve tvaru $v_k \cdots v_0 \bullet v_{-1} \cdots v_l$, kde $l, k \in \mathbb{Z}$.

Jelikož γ je fundamentální jednotkou tělesa K(viz [14]) a zároveň K neobsahuje žádný nereálný kořen z jednotky(pouze ± 1), grupu jednotek U_K lze zapsat takto

$$U_K = \left\{ \pm \gamma^k | k \in \mathbb{Z} \right\}$$

Má-li tedy být každé celé číslo $\alpha \in O_K$ vyjádřeno konečným součtem různých jednotek (tedy má-li platit DUG vlastnost), musí pro každé α existovat γ-reprezentace

$$v_k \cdots v_0 \bullet v_{-1} \cdots v_l, \quad \text{kde } v_i \in \{\bar{1}, 0, 1\} \text{ a } l, k \in \mathbb{Z}$$

Existence takového přepisu pro každý prvek okruhu O_K je uvedena v práci trojice Dombek, Masáková, Ziegler [14] a [16] a využívá přepisovacího pravidla $100\bar{1}1$, které odpovídá minimálnímu polynomu $X^4 - X + 1$. U jiných těles může být situace složitější. Nejzajímavější z těles, u kterých nebyla DUG vlastnost ověřena, jsou tělesa z druhé a třetí částí Věty 2.2, neboť z nereálných kořenů jednotky obsahují nejvýše $\pm i$. Oproti tomu tělesa ze čtvrté části Věty 2.2 obsahují například číslo $\sqrt[3]{1}$, což vede na složitější abecedu. Zdrojem následujícího shrnutí bylo ústní sdělení Daniela Dombka.

Uvažujme těleso $K = \mathbb{Q}(\gamma)$, kde γ je libovolným kořenem polynomu $X^4 + 2X^2 - 2X + 1$. K neobsahuje žádný nereálný kořen z jednotky(pouze ± 1). V daném tělese jsou fundamentálními jednotkami γ a $\frac{1}{\gamma}$. Na základě fundamentální jednotky γ je přepisovacím pravidlem $102\bar{2}1$, které odpovídá minimálnímu polynomu $X^4 + 2X^2 - 2X + 1$. Fundamentální jednotka $\frac{1}{\gamma}$ naopak zajišťuje existenci přepisovacího pravidla $1\bar{2}201$ v tělese $\mathbb{Q}(\frac{1}{\gamma})$. Podle Věty 2.2 lze každé algebraické celé číslo v K zapsat jako γ-reprezentaci ve tvaru

$$\alpha = \cdots u_1 u_0 \bullet u_{-1} u_{-2} \cdots$$

nad abecedou $\{-2, -1, 0, 1, 2\}$. DUG vlastnost a tedy existence konečné γ-reprezentace nad abecedou $\{-1, 0, 1\}$ pro každé algebraické celé číslo zatím dokázána nebyla.

Dalším tělesem je $K = \mathbb{Q}(\gamma)$, kde γ je libovolným kořenem polynomu $X^4 - X^3 + 2X^2 - X + 2$. U tohoto tělesa je také dokázána pro všechna algebraická celá čísla v K existence γ-reprezentace nad abecedou $\{-2, -1, 0, 1, 2\}$. K opět neobsahuje žádný nereálný kořen z jednotky (pouze ± 1). Fundamentální jednotky jsou $\pm(1 + \gamma^2), \pm\frac{1}{1+\gamma^2}$. Přepisovacími pravidly jsou pak $1\bar{1}301$, 11301 (v případě fundamentální jednotky $\pm(1+\gamma^2)$), resp. $103\bar{1}1$, 10311 (v případě fundamentální jednotky $\pm\frac{1}{1+\gamma^2}$).

Tabulka 2.1: Shrnutí vybraných těles podezřelých z DUG vlastnosti

Číselné těleso	Přepisovací pravidla	Známá abeceda	Cílová abeceda
$\mathbb{Q}(\gamma)$, kde γ je kořenem X^4+2X^2-2X+1	$102\bar{2}1$	$\{-2,-1,0,1,2\}$	$\{-1,0,1\}$
$\mathbb{Q}(\gamma)$, kde γ je kořenem $X^4-X^3+2X^2-X+2$	$1\bar{1}301$, 11301	$\{-2,-1,0,1,2\}$	$\{-1,0,1\}$
$\mathbb{Q}(\gamma)$, kde γ je kořenem X^4-X^3+X+1	$1\bar{1}011$	$\{-3,-2,-1,0,1,2,3\}$	$\{-1,0,1\}$
$\mathbb{Q}(\gamma)$, kde γ je kořenem $X^4-X^3+X^2+X+1$	$1\bar{1}111$	$\{-3,-2,-1,0,1,2,3\}$	$\{-1,0,1\}$
$\mathbb{Q}(\sqrt{7+4i})$	$1\bar{1}341$, $1(-i-2)\bar{1}$, $(2-i)\bar{5}(i-2)$	$\{0,\pm1,\pm i,\pm(1+i),$ $\pm(1-i),\ \pm2,\ \pm2i,$ $\pm(1+2i),\ \pm(1-2i),$ $\pm(2+i),\ \pm(2-i),$ $\pm(2+2i),\ \pm(2-2i)\}$	$\{0,\pm1,\pm i,$ $\pm(1+i),$ $\pm(1-i)\}$

Jedním z dalších těles je $K=\mathbb{Q}(\gamma)$, kde γ je libovolným kořenem polynomu X^4-X^3+X+1. V tomto tělese je dokázána existence γ-reprezentace algebraických celých čísel pouze nad abecedou $\{-3,-2,-1,0,1,2,3\}$. K neobsahuje žádný nereálný kořen z jednotky(pouze ±1) a fundamentální jednotky jsou γ a $\frac{1}{\gamma}$. To určuje přepisovací pravidla $1\bar{1}011$, resp. $110\bar{1}1$.

Následujícím tělesem je $K=\mathbb{Q}(\gamma)$, kde γ je libovolným kořenem polynomu $X^4-X^3+X^2+X+1$. V tomto tělese je také dokázána existence γ-reprezentace algebraických celých čísel pouze nad abecedou $\{-3,-2,-1,0,1,2,3\}$. K neobsahuje žádný nereálný kořen z jednotky(pouze ±1) a fundamentální jednotky jsou γ a $\frac{1}{\gamma}$. Přepisovacím pravidlem je tak $1\bar{1}111$, které odpovídá minimálnímu polynomu $X^4-X^3+X^2+X+1$. Přepisovacím pravidlem v tělese $\mathbb{Q}(\frac{1}{\gamma})$ je pak $111\bar{1}1$.

U tělesa $K=\mathbb{Q}(\sqrt{7+4i})$ je situace komplikovanější. Kromě reálných kořenů jednotky ±1 jsou v tělese K obsaženy i nereálné kořeny jednotky $\pm i$. Fundamentálními jednotkami ϵ jsou kořeny polynomů $X^4-2X^3+7X^2-2X+1$ a $X^4-4X^3+3X^2+4X+1$. Kvůli nereálným kořenům jednotky obsažených v K patří do integrální báze tělesa K také $\alpha=\epsilon i$. To vede na přepisovací pravidla, která obsahují nereálné cifry. Přepisovací pravidla v tělese K jsou například $1\bar{1}341$, $(2-i)\bar{5}(i-2)$ a $1(-i-2)\bar{1}$. Pro všechna algebraická celá čísla obsažená v K existuje γ-reprezentace nad abecedou $\{0,\pm1,\pm i,\pm(1+i),\pm(1-i),\pm2,\pm2i,\pm(1+2i),\pm(1-2i),\pm(2+i),\pm(2-i),\pm(2+2i),\pm(2-2i)\}$. Ověření DUG vlastnosti by znamenalo omezit abecedu na $\{0,\pm1,\pm i,\pm(1+i),\pm(1-i)\}$.

Všechny tyto uvedené poznatky jsou zobrazeny v tabulce 2.1.

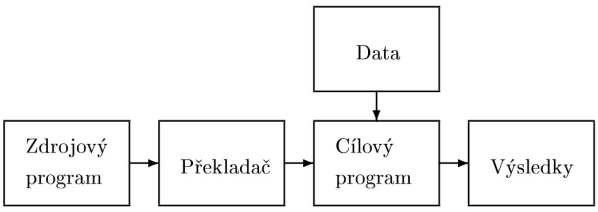
Kapitola 3

Překladače

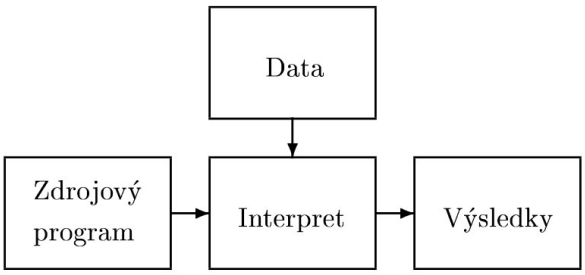
Pro potřeby implementace je důležité pochopit, jak funguje strojový překlad. Právě touto problematikou se zabývá následující kapitola. Podle [17] je **překladač** program, který k libovolnému zdrojovému(překládanému) programu P_z v jazyce J_z vytvoří cílový(přeložený) program P_c v jazyce J_c se stejným významem. Překladač tedy zpracuje text ve zdrojovém jazyce a převede na sémanticky ekvivalentní text cílového jazyka.

3.1 Hlavní části překladače

Překlad se skládá z několika částí. První částí je lexikální analýza, která transformuje kód zdrojového programu do tvaru, který je lépe zpracovatelný v dalších fázích překladače. Poté následuje syntaktická analýza, která vytváří derivační strom celého programu a určuje tak jeho strukturu. Poté sémantická analýza přiřadí jednotlivým symbolům ze syntaktické analýzy jejich význam. Následně se generuje mezikód, který může být optimalizován, a nakonec se vygeneruje program v cílovém jazyce. Lexikální, syntaktickou a sémantickou analýzu nazýváme souhrnně **přední částí překladače** (frontend) a je nezávislá na cílovém systému. Oproti tomu **zadní část překladače** (backend) zahrnuje generátor mezikódu, optimalizaci kódu a generátor kódu. Zadní část překladače je závislá na cílovém systému. Pro jeden programovací jazyk můžeme mít stejnou přední část překladače a podle cílového systému (Windows, MacOS, Linux) stačí změnit zadní část. Oproti tomu pro více programovacích jazyků na jednom systému lze použít stejnou zadní část překladače a změnit přední. Příkladem takového překladače je GCC. V rámci této práce se budeme zabývat především frontendem, který je specifický pro náš jazyk přepisovacích pravidel. O backend se postará překladač programovacího jazyka C++.



Obrázek 3.1: Schéma činnosti kompilačního překladače



Obrázek 3.2: Schéma činnosti interpretačního překladače

3.2 Typy překladačů

Pro program napsaný v některém z vyšších programovacích jazyků existuje více možností, jak takový program spustit. První variantou je převod do ekvivalentního strojového kódu počítače. Takový typ překladače se nazývá **kompilátor** nebo kompilační překladač. Druhou možností je **interpret** nebo interpretační překladač, který interpretuje příkazy zdrojového jazyka a přímo provádí požadované akce. Schéma obou překladačů je na obrázcích 3.1 a 3.2.

V praxi se často používají kombinace obou zmíněných přístupů, protože každý způsob překladu má své výhody i nevýhody. Podle [18] je nevýhodou kompilátorů obtížné hledání chyb ve zdrojovém programu, jelikož jsou k dispozici informace o chybách pouze v pojmech strojového jazyka, jako je například obsah paměti nebo adresy jednotlivých instrukcí. Moderní překladače tuto slabinu řeší vytvářením pomocných souborů, které umožňují provádět ladění programu přímo na úrovni strojového kódu. Mezi výhody kompilačních překladačů lze zařadit to, že k překladu dochází pouze jednou (byť jde o časově náročný proces) a dále se spouští již přeložený program.

Interpretace je daleko pomalejší, protože dochází k analyzování zdrojových příkazů pokaždé, kdy na ně program narazí. Poměr rychlostí zkompilovaného a interpretovaného programu je uváděn v rozmezí 1:10 a 1:100. Výhodou interpretačních překladačů jsou především přesné informace o chybách ve zdrojovém programu. Další výhodou je možnost spouštět programy, ve kterých

se typy objektů dynamicky mění. Interprety jsou také strojově nezávislé, výsledný program je tak přenositelný mezi různými platformami.

Vavrečková [17] dodává, že další výhodou kompilátoru je možnost náročných optimalizací a kontrol a nižší využití pamětí při běhu programu. Oproti tomu výhodami interpretů je doba spuštění programu, menší velikost interpretovaného programu na disku. Vytvořit interpret také považuje za jednodušší než vytvořit kompilátor.

3.3 Lexikální analýza

První nedílnou součástí překladače jazyka je lexikální analyzátor. Ten přečte zdrojový kód ve vstupním jazyce a převede jej na posloupnost lexikálních elementů – tokenů. Tokenem rozumíme terminální symboly bezkontextové gramatiky, která popisuje syntaxi vstupního jazyka. Lexikální analyzátor tak rozpozná identifikátory, klíčová slova, literály, operátory a speciální symboly (jednoznakové i víceznakové). Naproti tomu lexikální analyzátor odstraňuje mezery, oddělovače řádků a komentáře. V případě potřeby reaguje na direktivy překladače. Každý token lze přijmout konečným automatem. Jazyk lexikálních elementů je tak regulární, lze jej popsat regulární gramatikou, a proto lze lexikální analyzátor realizovat pomocí konečného automatu. V případě přečtení identifikátoru uloží lexikální analyzátor název tohoto identifikátoru do tabulky symbolů a v případě čísla jeho hodnotu. Tyto uložené údaje pak využije syntaktický analyzátor.

Další, volitelnou funkcí lexikálního analyzátoru je zajištění konzistence chybových hlášení syntaktického analyzátoru a zdrojového textu. Pokud si lexikální analyzátor ke každému tokenu zapamatuje i číslo řádku, na kterém se nachází, může syntaktický analyzátor v případe chyby přímo určit, na kterém řádku zdrojového kódu se chyba vyskytla.

I když lexikální analyzátor obecně nekontroluje přímo strukturu jazyka, dokáže odhalit některé chyby. Pokud lexikální analyzátor objeví neznámý znak, vrátí chybu. Stejně tak skončí chybou, pokud nenajde konec komentáře, či konec nějakého jiného řetězce. Naproti tomu překlep v klíčovém slovu nedetekuje, neboť si myslí, že se jedná o identifikátor. Tuto chybu detekuje tedy až syntaktický analyzátor.

Klíčová slova obvykle nemají v automatu lexikálního analyzátoru vlastní přechody. Nejčastěji se tak přečte celý identifikátor a následně se podle tabulky vyhledá, zda-li identifikátor není klíčovým slovem. Pokud ano, pak lexikální analyzátor vrátí token příslušného klíčového slova.

3.4 Syntaktická analýza

Syntaktický analyzátor na základě posloupnosti tokenů v programu, které získá z lexikální analýzy, rozpozná syntaktické konstrukce jazyka a sestaví

syntaktickou strukturu programu pomocí tzv. derivačního stromu. V případě chybné jazykové konstrukce se většinou pokusí o zotavení, následně pokračuje v analýze a teprve potom ohlásí chybu. To umožňuje odhalit více chyb zároveň.

Některé jazykové struktury nelze popsat regulární gramatikou, typickým případem můžou být závorkové struktury. Párovost závorek nelze popsat regulární gramatikou, a proto takový jazyk nelze přijímat konečným automatem. Pro syntaktickou analýzu se tak často využívají zásobníkové automaty, přestože existují jazyky, na které nestačí ani zásobníkové automaty.

Existují dva základní typy syntaktické analýzy, které se liší v postupu vytváření syntaktického stromu. První je syntaktická analýza shora-dolů. Při této analýze se postupně expandují neterminály nejvíce vlevo ve snaze dospět k řetězci terminálu, který se shoduje s posloupnosti tokenů na vstupu. Vzniká tak levý rozklad. Tento typ analyzátorů se označuje jako LL-analyzátory (Left to right, Leftmost derivation). Syntaktickou analýzu shora dolů nelze provádět pro libovolnou gramatiku, ale pouze pro bezkontextovou LL gramatiku.

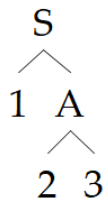
Druhým typem je syntaktická analýza zdola-nahoru. Takový analyzátor upravuje výslednou posloupnost tokenů a snaží se ji zredukovat na počáteční neterminál gramatiky. Během této analýzy se tak načítají vstupní tokeny zleva doprava a vzniká pravý rozklad. Proto se tento typ nazývá LR-analyzátor (Left to right, Rightmost derivation). LR-analyzátory jsou silnější než LL-analyzátory, umožňují tedy analyzovat i bezkontextové jazyky, které nejsou LL.

3.4.1 Derivační strom

Derivační strom, jakožto výstup syntaktické analýzy, popisuje syntaktickou strukturu programu. Formální definice podle [2] uvádí, že derivační strom je orientovaný acyklický graf, který má jediný kořen a do každého uzlu vstupuje právě jedna hrana. Uzly, z nichž žádná hrana nevystupuje, jsou listy.

Pro derivační strom v dané gramatice $G = (N, T, P, S)$ platí:

1. Uzly derivačního stromu jsou ohodnoceny terminálními a neterminálními symboly.
2. Kořen stromu je ohodnocen počátečním symbolem.
3. Pokud má uzel alespoň jednoho následovníka, je ohodnocen neterminálním symbolem.
4. Jestliže $n_1, n_2, \ldots, n_k$ jsou bezprostřední následovníci uzlu n, který je ohodnocen symbolem A, a tyto uzly jsou zleva doprava ohodnoceny symboly $A_1, A_2, \ldots, A_k$, pak $A \to A_1 A_2 \ldots A_k$ je pravidlo v P
5. Koncové uzly derivačního stromu tvoří zleva doprava větnou formu nebo větu v gramatice G, která je výsledkem derivačního stromu



Obrázek 3.3: Derivační strom pro slovo 123 v gramatice z příkladu 3.1

Při kreslení derivačních stromů se dodržují tyto konvence:

- Kořen stromu je vždy nejvýše a všechny hrany jsou orientovány směrem dolů, proto lze vynechat šipky označující orientaci hran.
- Uzel se reprezentuje pouze jeho ohodnocením.

Příklad 3.1. *Uvažujme gramatiku G z příkladu 1.2, který popisuje jazyk všech nezáporných celých čísel. Platí $G = (\{S, A\}, \mathcal{A}, P, S)$, kde P obsahuje pravidla:*

$$S \to 0|1A|2A|3A|4A|5A|6A|7A|8A|9A$$
$$A \to 0A|1A|2A|3A|4A|5A|6A|7A|8A|9A|\epsilon$$

Slovo 123 *lze generovat derivací $S \Rightarrow 1A \Rightarrow 12A \Rightarrow 123$. Této derivaci odpovídá derivační strom na obrázku 3.3.*

3.4.2 Analýza shora dolů

Při syntaktické analýze shora dolů se na základě vstupní posloupnosti tokenů vytváří derivační strom od kořene (počátečního symbolu gramatiky) a postupným přidáváním následníků vznikne celý derivační strom. Výstupem je tedy derivační strom, který popisuje syntaktickou strukturu vstupní věty.

Při této konstrukci se nahrazuje vždy nejlevější neterminální symbol, to vede ke vzniku levého rozkladu. Při samotném procesu se používají 2 operace – expanze a srovnání. Expanze se využije v případě, že je na vrcholu zásobníku neterminál, a dojde k nahrazení tohoto neterminálu, který se nachází na levé straně nějakého pravidla, pravou stranou stejného pravidla. K srovnání dochází, pokud je na vrcholu zásobníku terminální symbol. Při této operaci se tento terminální symbol odebere zároveň z vrcholu zásobníku a ze vstupní pásky. Pro obecnou bezkontextovou gramatiku nemusí být vždy jasné, jaké pravidlo k nahrazení neterminálu použít. K tomu dojte, pokud existuje více pravidel se stejným neterminálem na levé straně a na vrcholu zásobníku se vyskytne onen neterminál. První možností, jak se nejednoznačností vypořádat, je vybrat jedno z více pravidel a v případě neúspěchu se vrátit zpět a vybrat jiné. Tomuto postupu se říká syntaktická analýza s návratem. Jedná se o časově

náročné řešení a proto se v praxi pro běžné jazykové konstrukce nepoužívá. Existuje však i deterministická varianta, tedy bez návratu.

3.4.2.1 FIRST a FOLLOW

Pro nejběžnější deterministickou syntaktickou analýzu jsou zásadní množiny $FIRST$ a $FOLLOW$. Podle [2] je množina $FIRST$ definována následovně:

Definice 3.2. Nechť $G = (N, T, P, S)$ je bezkontextová gramatika a $\alpha \in (N \cup T)^*$. Pro každé $\alpha \in (N \cup T)^*$ definujeme $FIRST(\alpha) = \{a : a \in T, \alpha \Rightarrow^* a\beta, \beta \in (N \cup T)^*\} \cup \{\epsilon : \alpha \Rightarrow^* \epsilon\}$.

Jedná se tak o množinu všech terminálních symbolů, jimiž začínají řetězce derivované z větné formy α. Pokud lze prázdný řetězec získat derivací z α, pak i prázdný řetězec patří do množiny $FIRST$.

Pro množinu $FOLLOW$ je v [2] uvedena tato definice:

Definice 3.3. Nechť $G = (N, T, P, S)$ je bezkontextová gramatika. Pro každý neterminální symbol $X \in N$ definujeme $FOLLOW(X) = \{a : a \in T, S \Rightarrow^* \alpha X \beta, \beta \Rightarrow^* a\gamma, \gamma \in (N \cup T)^*\} \cup \{\epsilon : S \Rightarrow^* \alpha X\}$.

Neformálně jde o množinu všech terminálních symbolů, které se mohou nacházet bezprostředně vpravo od X v nějaké větné formě. Pokud je symbol X posledním symbolem větné formy v nějaké derivaci, pak množina $FOLLOW$ obsahuje prázdný řetězec.

Množiny $FIRST$ a $FOLLOW$ jsou postačující pro vytvoření LL-překladače pouze pro takové gramatiky, kde při rozhodování, které pravidlo gramatiky použít, stačí přečíst nejvýše jeden symbol na vstupu. Tyto gramatiky označujeme jako $LL(1)$. Gramatiky, kde je potřeba přečíst nejvýše k symbolů ze vstupu, aby šlo jednoznačně rozhodnout, podle kterého pravidla gramatiky provést expanzi, nazýváme $LL(k)$. V případě $LL(k)$ gramatik je potřeba definovat množiny $FIRST_k$ a $FOLLOW_k$.

Definice 3.4. Nechť $G = (N, T, P, S)$ je bezkontextová gramatika, $\alpha \in (N \cup T)^*$ a $X \in N$.

$FIRST_k(\alpha) = \{x : x \in T^*, \alpha \Rightarrow^* xy \text{ a } |x| = k, y \in T^*\} \cup \{x : x \in T^*, \alpha \Rightarrow^* x \text{ a } |x| < k\}$.

$FOLLOW_k(X) = \{x : x \in T^*, S \Rightarrow^* wXxy \text{ a } |x| = k, y \in T^*\} \cup \{x : x \in T^*, S \Rightarrow^* wXx \text{ a } |x| < k, \}$.

$FIRST_k(\alpha)$ je tak množinou všech řetězců délky k, kterými může začínat nějaká větná forma derivovaná z α. Pokud lze z α derivovat kratší slovo, pak i toto slovo patří do množiny $FIRST_k(\alpha)$. Analogicky z $FOLLOW(X)$ je odvozena $FOLLOW_k(X)$. Je to množina všech k-tic terminálů, které se mohou v nějaké větné formě vyskytovat bezprostředně vpravo od symbolu X.

Množina $FOLLOW_k(X)$ může obsahovat i řetězce kratší než k, a to tehdy, pokud takovými řetězci končí nějaká větná forma derivovaná z X.

Pro účely práce budou postačující algoritmy 3.1 a 3.2 pro výpočet $FIRST_1$ a $FOLLOW_1$, uvedené v [17], lze je však drobnými úpravami zobecnit pro libovolné množiny $FIRST_k$ a $FOLLOW_k$. U výpočtu $FIRST$ si lze povšimnout, že je třeba odlišit neterminály N obsahující epsilonové pravidlo $N \to \epsilon$. Pokud by tato pravidla nebyla brána v potaz, mohli bychom do množiny $FIRST$ nesprávně přidat ϵ (v algoritmu 3.1 použitím řádku 8) nebo nesprávně nepřidat ϵ (v algoritmu 3.1 použitím řádku 11).

Z uvedených algoritmů si lze povšimnout, že výpočet $FOLLOW$ je náročnější než výpočet $FIRST$.

Algoritmus 3.1 Konstrukce množiny $FIRST$

Vstup: Gramatika $G = (N, T, P, S)$, $\alpha \in (N \cup T)^*$, množina neterminálů s epsilonovým pravidlem $N_1 \subset N$
Výstup: $FIRST(\alpha)$

1: **if** $\alpha = \epsilon$ **then**
2: **return** ϵ
3: **end if**
4: **if** $\alpha = a\beta$ pro $a \in T, \beta \in (N \cup T)^*$ **then**
5: **return** a
6: **end if**
7: **if** $\alpha = A\gamma$ pro $A \in N \setminus N_1, \gamma \in (N \cup T)^*$ **and** $A \to \beta_1|\beta_2|\ldots|\beta_n \in P$ **then**
8: **return** $\bigcup_{i=1}^{n} FIRST(\beta_i)$
9: **end if**
10: **if** $\alpha = A\gamma$ pro $A \in N_1, \gamma \in (N \cup T)^*$ **and** $A \to \beta_1|\beta_2|\ldots|\beta_n \in P$ **then**
11: **return** $(\bigcup_{i=1}^{n} FIRST(\beta_i) - \{\epsilon\}) \bigcup FIRST(\gamma)$
12: **end if**

3.4.2.2 Rozkladová tabulka

Pro každou $LL(k)$ gramatiku lze sestrojit tzv. rozkladovou tabulku, která určuje, podle kterých pravidel se má při syntaktické analýze shora dolů provádět expanze. Pro $LL(k)$ gramatiku $G = (N, T, P, S)$ má rozkladová tabulka M rozměry $|N| \times |T^{*k}|$, kde $T^{*k} = \{x : x \in T^*, |x| \leq k\}$. Rozkladová tabulka je tak vlastně zobrazení $M(A, a), A \in N, a \in T^{*k} \cup \epsilon$, jehož hodnotou je číslo pravidla, které se má použít při expanzi neterminálu A a k-tice a na vstupní pásce, nebo symbol chyby. Algoritmus 3.3 publikovaný v [2] popisuje konstrukci rozkladové tabulky.

V případě, že se během vytváření rozkladové tabulky pro danou gramatiku G a dané k přepisuje již existující hodnota (číslo pravidla pro expanzi) v tabulce, dochází k nejednoznačnosti při expanzi a daná gramatika

Algoritmus 3.2 Konstrukce množiny $FOLLOW$

Vstup: Gramatika $G = (N, T, P, S), X \in N$
Výstup: M = $FOLLOW(X)$

1: **if** $X = S$ **then**
2: M.add(ϵ)
3: **end if**
4: **for all** $X \to \alpha A \beta \in P$, kde $\alpha, \beta \in (N \cup T)^*, B \in N$ **do**
5: M.add($FIRST(\beta) - \{\epsilon\}$)
6: **end for**
7: **for all** $A \to \alpha X \beta \in P$, kde $\alpha, \beta \in (N \cup T)^*, B \in N, \epsilon \in FIRST(\beta)$ **do**
8: M.add($FOLLOW(A)$)
9: **end for**
10: **for all** $A \to \alpha X \in P$, kde $\alpha \in (N \cup T)^*, B \in N$ **do**
11: M.add($FOLLOW(A)$)
12: **end for**
13: **return** M

Algoritmus 3.3 Konstrukce rozkladové tabulky pro $LL(k)$ gramatiku.

Vstup: $LL(k)$ gramatika $G = (N, T, P, S)$
Výstup: Rozkladová tabulka M pro gramatiku G o rozměrech $N \times T^{*k}$

1: **for** $i = 1$ to $|P|$ **do**
2: **if** $A \to \alpha$ je i-té pravidlo v P **then**
3: $M(A, x) = i$ pro všechna $x \in FIRST_k(\alpha)$ taková, že $|x| = k$
4: **end if**
5: **if** $A \to \alpha$ je i-té pravidlo v P **and** $y \in FIRST_k(\alpha)$ takové, že $|y| < k$ **then**
6: $M(A, z) = i$ pro všechna $z \in FIRST_k(yFOLLOW_k(\alpha))$
7: **end if**
8: **end for**
9: $M(Y, x) \leftarrow$ chyba, pro všechny ostatní případy.
10: **return** M

není $LL(k)$. Tyto konflikty mohou být dvojího typu – $FIRST - FIRST$ a $FIRST - FOLLOW$. K prvnímu uvedenému dochází, pokud existuje více pravidel pro stejný neterminál se stejnou hodnotou $FIRST$. Ke konfliktu $FIRST - FOLLOW$ dojde u pravidel typu $A \to \alpha_1 | \ldots | \alpha_n$, pokud existuje pravidlo $\alpha_i \Rightarrow^* \epsilon$ a pro nějaké $i \neq j$ platí $FIRST(\alpha_j) \cap FOLLOW(A) \neq \emptyset$.

Rozkladová tabulka pro $LL(1)$ gramatiku je pouze konkrétním případem $LL(k)$.

3.4.2.3 Transformace gramatiky na $LL(1)$

Existují jazyky, které nelze popsat $LL(k)$ gramatikou. Pro gramatiky všech ostatních jazyků existují pravidla, kterými lze danou gramatiku transformovat na $LL(1)$ gramatiku. Tato pravidla jsou detailněji popsána v [2]. Aplikování těchto pravidel na gramatiku není deterministické a nemusí vždy vést k cíli.

Mezi tato pravidla patří:

- **Odstranění levé rekurze**

  Při transformaci dochází k tomuto nahrazení pravidel

  $$A \to A\alpha_1 | \ldots | A\alpha_n | \beta_1 | \ldots | \beta_m$$
  $$\downarrow$$
  $$A \to \beta_1 A' | \ldots | \beta_m A'$$
  $$A' \to \alpha_1 A' | \ldots | \alpha_n A' | \epsilon.$$

- **Levá faktorizace**

  Levá faktorizace sjednocuje pravidla se stejným prefixem na pravé straně, které by vedly v rozkladové tabulce na $FIRST - FIRST$ konflikt. Stejně jako při odstranění levé rekurze se přidává nový neterminál. Dochází k této transformaci:

  $$A \to \alpha\alpha_1 | \ldots | \alpha\alpha_n$$
  $$\downarrow$$
  $$A \to \alpha A'$$
  $$A' \to \alpha_1 | \ldots | \alpha_n$$

- **Rohová substituce**

  Rohovou substituci je možné použít v případě $FIRST - FIRST$ konfliktu, který je způsoben pravidlem, jehož pravá strana začíná neterminálem. Expanzí tohoto neterminálu může dojít k odstranění konfliktu.

$$B \to \beta_1 | \ldots | \beta_n$$
$$A \to B\alpha$$
$$\downarrow$$
$$A \to \beta_1\alpha | \ldots | \beta_m\alpha$$

- **Pohlcení terminálu**

  Pohlcení terminálu může odstranit $FIRST - FOLLOW$ konflikt. Teto transformace lze využít, pokud se pohlcovaný terminál vyskytuje přímo za problematickým neterminálem. Příkladem je transformace níže:

$$A \to \alpha B\beta$$
$$B \to \beta\gamma | \epsilon$$
$$\downarrow$$
$$A \to \alpha B'$$
$$B' \to \beta | \beta\gamma\beta$$

  Došlo sice ke vzniku $FIRST-FIRST$ konfliktu, ten je ale možné pomocí levé faktorizace odstranit.

- **Extrakce pravého kontextu** V případě, že nelze využít pravidlo pohlcení terminálu je možné opakovaně substituovat za neterminál bezprostředně vpravo od problematického neterminálu a získat tak konflikt, na který je možné aplikovat pravidlo pro pohlcení terminálu.

$$A \to \alpha BB\beta$$
$$B \to \beta\gamma | \epsilon$$
$$\downarrow$$
$$A \to \alpha B\beta\gamma\beta | \alpha B\beta$$
$$B \to \beta\gamma | \epsilon$$

  Nyní je možné pohltit terminál (případně řetězec terminálů) a následnou levou faktorizací vytvořit $LL(1)$ gramatiku.

3.4.2.4 Syntaktická analýza shora dolů pomocí zásobníkového automatu

Mějme rozkladovou tabulku M pro $LL(1)$ gramatiku $G = (N, T, P, S)$ a vstupní řetězec $w \in T^*$. Konfigurací rozumějme trojici (x, α, π), kde $x \in T^*$ je doposud nepřečtená část vstupu, $\alpha \in (N \cup T)^*$ je obsah zásobníku a π dosud vytvořená část levého rozkladu (posloupnost pravidel při provedených expanzích). Levý rozklad získáme pomocí přechodů z konfigurace (w, S, ϵ) do konfigurace

$(\epsilon, \epsilon, \pi)$, přičemž π je získaný levý rozklad. Přechod mezi konfiguracemi označujeme symbolem $\vdash$. V případě, že je na vrcholu zásobníku neterminál, dojde k jeho expanzi podle rozkladové tabulky M, tedy nahrazení neterminálu na zásobníku pravou stranou pravidla a připsání čísla daného pravidla k již vytvořené části levého rozkladu. Pokud se nachází na vrcholu zásobníku a na vstupu stejný terminální symbol, dojde ke srovnání a ze vstupu i z vrcholu zásobníku se daný terminál odebere. Pokud není možné použít expanzi ani srovnání a automat se nenachází v konfiguraci $(\epsilon, \epsilon, \pi)$, dojde k chybě a analýza skončí neúspěchem.

Příklad 3.5 popisuje, jak může konstrukce takového syntaktického analyzátoru vypadat.

Příklad 3.5. *Uvažujme jazyk L, který generuje jednoduché matematické výrazy se sčítáním a násobením čísel. Příkladem gramatiky generující jazyk L může být gramatika $G_1 = (\{S, A, B\}, \{+, *, num\}, P_1, S)$, kde P_1 jsou následující pravidla:*

$$S \to A + A | A * A$$
$$A \to S | num$$

Z algoritmu 3.3 lze vidět, že pro konstrukci rozkladové tabulky je nutné nalézt množiny $FIRST$ pro pravé strany všech pravidel. Tyto požadované množiny $FIRST$ jsou:

$$FIRST(A + A) = FIRST(A * A) = \{num\}$$
$$FIRST(S) = \{num\}$$

Nyní lze vidět, že zvolená gramatika obsahuje konflikty $FIRST - FIRST$, a proto není $LL(1)$. Tuto gramatiku lze však transformovat na $LL(1)$. Takovým Příkladem může být gramatika $G_2 = (\{S, A, B\}, \{+, *, num\}, P_2, S)$ s pravidly P_2:

$$S \to AB \tag{3.1}$$
$$A \to num \tag{3.2}$$
$$B \to +S \tag{3.3}$$
$$B \to *S \tag{3.4}$$
$$B \to \epsilon \tag{3.5}$$

Pro množiny $FIRST$ a $FOLLOW$ platí:

$$FIRST(AB) = FIRST(num) = \{num\}$$
$$FIRST(+S) = \{+\}$$
$$FIRST(*S) = \{*\}$$
$$FOLLOW(S) = FOLLOW(B) = \{\epsilon\}$$
$$FOLLOW(A) = \{+, *, \epsilon\}$$

Na základě množin $FIRST$ a $FOLLOW$ lze sestrojit rozkladovou tabulku 3.1.

Tabulka 3.1: Rozkladová tabulka pro gramatiku z příkladu 3.5

	num	$+$	$*$	ϵ
S	1			
A	2			
B		3	4	5

K nalezení levé derivace pro slovo $num + num * num$ dojde následujícími přechody zásobníkového automatu:

$$\begin{aligned}(num + num * num, S, \epsilon) &\vdash (num + num * num, AB, 1)\\ &\vdash (num + num * num, numB, 12)\\ &\vdash (+num * num, B, 12) \vdash (+num * num, +S, 123)\\ &\vdash (num * num, S, 123) \vdash (num * num, AB, 1231)\\ &\vdash (num * num, numB, 12312) \vdash (*num, B, 12312)\\ &\vdash (*num, *S, 123124) \vdash (num, S, 123124)\\ &\vdash (num, AB, 1231241) \vdash (num, numB, 12312412)\\ &\vdash (\epsilon, B, 12312412) \vdash (\epsilon, \epsilon, 123124125)\end{aligned}$$

Slovo $num + num * num$ je tak součástí jazyka L a levou derivací je posloupnost pravidel 123124125*.*

3.4.3 Analýza zdola nahoru

Zdrojem poznatků k této kapitole byla skripta [18] od trojice Češka, Hruška, Beneš. U syntaktické analýzy zdola nahoru je situace poněkud komplexnější. Jelikož je pro tuto práci důležitější syntaktická analýza shora dolů, je problematika syntaktické analýzy zdola nahoru zjednodušena.

Během této analýzy dochází k tvorbě derivačního stromu ze vstupního řetězce od listů ke kořenu. K tomu se využívají operace přesun a redukce. Při přesunu dochází k přesunutí symbolu ze vstupu na zásobník. Redukce probíhá na vrcholu zásobníku a nahrazuje se při ní část řetězce, která se nachází na pravé straně nějakého pravidla, symbolem na levé straně téhož pravidla. Obdobně jako u syntaktické analýzy shora dolů lze sestrojit tabulku, podle které se syntaktický analyzátor rozhodne, jakou operaci provede v závislosti na kontextu již proběhlé analýzy a symbolu (či více symbolů) na vstupu. Konstrukce takových rozkladových tabulek je však složitější než u LL analýzy a podle typu gramatiky existují 3 různé metody, jak rozkladovou tabulku vytvořit, které se liší v síle a složitosti implementace. Pro účely této práce však není konstrukce rozkladových tabulek pro LR analýzu podstatná a proto ji zde neuvádíme. Ukázka průběhu syntaktické analýzy zdola nahoru je uvedena v příkladu 3.6

Příklad 3.6. *Uvažujme jazyk L, který generuje jednoduché matematické výrazy se sčítáním a násobením čísel. Příkladem gramatiky generující jazyk L může být LR(1) gramatika* $G = (\{S, A, B\}, \{+, *, num\}, P, S)$*, kde P jsou následující pravidla:*

$$S \to S + A \tag{3.1}$$
$$S \to A \tag{3.2}$$
$$A \to A * B \tag{3.3}$$
$$A \to B \tag{3.4}$$
$$B \to num \tag{3.5}$$

Rozkladová tabulka pro danou gramatiku je zobrazena v tabulce 3.2

Tabulka 3.2: Rozkladová tabulka pro gramatiku z příkladu 3.6

stav	num	+	*	ϵ	S	A	B
0	s5				1	2	3
1		s6		acc			
2		r2	s7	r2			
3		r4	r4	r4			
4	s5				8	2	3
5		r5	r5	r5			
6	s5					9	3
7	s5						10
8		s6					
9		r1	s7	r1			
10		r3	r3	r3			

V rozkladové tabulce si značí stav přesun a přidání stavu i na vrchol zásobníku, ri znamená redukci podle i-tého pravidla, acc značí přijetí a prázdné pole značí chybu. Samotná čísla značí přechod, při kterém nedochází ani k přesunu ani k redukci.

Uvažujme vstupní větu $num * num + num$*. Analýzu této věty lze vidět v tabulce 3.3*

Tabulka 3.3: Syntaktická analýza řetězce $num * num + num$ z příkladu 3.6

	zásobník	vstup	akce
(1)	0	$num * num + num$	přesun
(2)	0 num 5	$*num + num$	redukce podle $B \to num$ (5)
(3)	0 B 3	$*num + num$	redukce podle $A \to B$ (4)
(4)	0 A 2	$*num + num$	přesun
(5)	0 A 2 * 7	$num + num$	přesun
(6)	0 A 2 * 7 num 5	$+num$	redukce podle $B \to num$ (5)
(7)	0 A 2 * 7 B 10	$+num$	redukce podle $A \to A * B$ (3)
(8)	0 A 2	$+num$	redukce podle $S \to A$ (2)
(9)	0 S 1	$+num$	přesun
(10)	0 S 1 + 6	num	přesun
(11)	0 S 1 + 6 num 5		redukce podle $B \to num$ (5)
(12)	0 S 1 + 6 B 3		redukce podle $A \to B$ (4)
(13)	0 S 1 + 6 B 9		redukce podle $S \to S + A$ (1)
(14)	0 S 1		přijetí

Analýza řetězce $num * num + num$ *začíná ve stavu* 0 *(který je na vrcholu zásobníku) a symbolem na vstupu* num*. Rozkladová tabulka má v řádku* 0 *a sloupci num hodnotu* $s5$*, což značí, že se provede přesun terminálu ze vstupu na zásobník a dojde k přesunu do stavu* 5*, který se také uloží na zásobník. Tato operace je znázorněna v tabulce 3.3 na řádku* 1*. Nyní je na vstupu symbol* $*$ *a syntaktický analyzátor se nachází ve stavu* 5*. Podle rozkladové tabulky se provede operace s označením* $r5$*, což značí redukci podle pátého pravidla, tedy pravidla* $B \to num$*. Ze zásobníku tak dojde k odebrání symbolu num spolu se symbolem aktuálního stavu* 5 *a nahradí se neterminálem* B*. Na zásobníku tak zůstane* 0 B*. Podle rozkladové tabulky ze stavu* 0 *a symbolu* B *dojde k přesunu do stavu* 3*, a tak ho přidáme na zásobník. Aktuální stav zásobníku tak lze nyní vidět na řádku* 3 *tabulky 3.3. Tímto způsobem můžeme pokračovat v syntaktické dále, až bude přečten celý vstup a na zásobníku zůstane* 0 S 1*. Podle rozkladové tabulky je tak možná jediná operace - přijetí vstupu.*

Pravou derivací je pak posloupnost čísel pravidel, podle kterých postupně probíhaly redukce, avšak v opačném pořadí. Pro slovo $num * num + num$ *je pravou derivací posloupnost* 14523545*.*

Kapitola 4

Analýza

Hlavní součástí této práce je rozšíření programu pro přepisování číselných řetězců. Tato kapitola shrnuje stávající situaci a vymezuje požadavky na přepisovací program.

4.1 Aktuální stav

Stávající program pro přepisování číselných řetězců byl součástí práce [15]. Program slouží k přepisování reprezentací čísel v bázi γ, která je kořenem polynomu $X^4 - X + 1$, na reprezentace nad abecedou $\{-1, 0, 1\}$. Jedná se tedy o program, který každé algebraické celé číslo $x \in O_K$ v tělese $K = \mathbb{Q}(\gamma)$ vyjádří jako součet různých jednotek. Program nejdříve využije 8 konkrétních pravidel, který vstupní reprezentaci přepíše na reprezentaci ve formě řídkého řetězce nad abecedou $\{-2, -1, 0, 1, 2\}$. V druhé fázi dojde v přepsání této nové reprezentace na výslednou reprezentaci nad abecedou $\{-1, 0, 1\}$.

Načítání vstupu je možné ze standardního vstupu nebo ze souboru. Výsledek přepisování je možné uložit do souboru nebo zobrazit na standardní výstup. Tento program je implementován v jazyce C++ a je určen primárně pro operační systémy Windows. Aplikace je pouze konzolová a nemá tedy grafické rozhraní.

Přepisovací pravidla, pomocí kterých dochází k přepisování, jsou natvrdo implementována přímo ve zdrojovém kódu. Jelikož je celý program implementován v jazyce, který je kompilovaný, a aplikování přepisovacích pravidel je podmíněné, nelze pouze přesunout tato pravidla do externího souboru, ze kterého budou načítána. Je navíc nezbytné implementovat změny v podmíněném vyhodnocování pravidel za běhu programu. Jelikož má být program použitelný samostatně bez dalších prerekvizit, není možné kompilovat program s novými pravidly (či jeho část) před každým spuštěním nebo při každé změně pravidel.

Naskýtá se tak otázka, zda využít již existující lexikální a syntaktické analyzátory jako Flex, yacc a bison. Jelikož jazyk přepisovacích pravidel je poměrně jednoduchý a je možné jej realizovat jednoprůchodově pomocí re-

kurzivního sestupu, lze očekávat, že již existující parsery pro složitější jazyky (např. popsanými LALR gramatikou) budou pomalejší. Další nespornou výhodou vlastní implementace lexikální a syntaktické analýzy je vlastní error management, který je možné upravit podle potřeby, na rozdíl od již existujících parserů.

Podle [16] lze každé algebraické celé číslo napsat jako γ-reprezentaci nad abecedou $\mathbb{Z}$. V kapitole 2 je ukázáno, že pro některé báze nestačí abeceda celých čísel, ale je nezbytná abeceda komplexních celých čísel. Stávající implementace využívá třídu *NumString*, která slouží k uchovávání číselných řetězců nad abecedou $\mathbb{Z}$. Vlastní řetězec je uchováván ve formě dynamicky alokovaného pole a indexu, na kterém se nachází zlomková tečka.

Pro nahrazení abecedy komplexními celými čísly je nutné především zvolit v dané třídě jiné datové struktury, upravit načítání a výpis číselných řetězců. Kromě třídy *NumString* budou podobné úpravy nutné i ve vlastním programu.

4.2 Požadavky

4.2.1 Funkční požadavky

- **Načtení přepisovacích pravidel**
  Program za běhu načte ze souboru přepisovací pravidla, s jejichž pomocí bude přepisovat řetězec zadaný uživatelem.

- **Načtení vstupního řetězce**
  Vstupní řetězec je možné načíst ze standardního vstupu nebo ze zvoleného souboru. Vstupní řetězec je ve tvaru posloupnosti komplexních celých čísel a vyznačenou zlomkovou tečkou.

- **Výstup do souboru**
  Výsledek výpočtu je možné zobrazit na standardní výstup nebo vypsat do souboru. Výstupní reprezentace čísla v dané bázi je formě posloupnosti komplexních celých čísel s vyznačenou zlomkovou tečkou.

- **Omezení doby běhu**
  Uživatel může nastavit maximální dobu běhu, po které se výpočet zastaví, nehledě na to, zda byl nalezen výsledný řetězec.

- **Přepsání vstupní reprezentace podle zadaných pravidel**
  Program bude přepisovat vstupní řetězec podle zadaných pravidel, dokud nedojde k překročení povolené doby běhu nebo nebude nalezena reprezentace, ve které se nevyskytují velké cifry. Maximální povolenou velikost i dobu běhu volí uživatel.

4.2.2 Nefunkční požadavky

- **Rozšíření stávající aplikace**
  Implementace rozšiřuje stávající program pro přepisování řetězců.
- **Běh na Windows**
  Program běží na systému Windows (7 a vyšší).
- **Samostatný běh programu**
  Program je možné provozovat na běžném osobním počítači, bez nutnosti instalace speciálního softwaru nebo hardwaru. Program běží na lokálním systému, komunikace či ovládání přes internet není vyžadováno.
- **Textové uživatelské rozhraní - CLI**
  Program funguje jako konzolová aplikace. Grafické rozhraní není vyžadováno, program je však přehledný a jednoduchý na ovládání.

Kapitola 5

Návrh

Tato kapitola je věnována návrhu programu pro přepisování číselných řetězců a detailně popisuje jeho části.

5.1 Interpret

Implementovaný program musí být schopný přepisovat číselné řetězce pomocí přepisovacích pravidel zadaných uživatelem. Tato pravidla nejsou známa v době překlad programu a vyhodnocují se podmíněně. To vyžaduje takovou reprezentaci pravidel, které bude daný program rozumět a na jejíž základě bude schopen za běhu programu pravidla vyhodnocovat.

Existují dva způsoby, jak tento problém vyřešit. První možností je vytvořit z přepisovacího pravidla zdrojový kód, ten následně zkompilovat jako dynamickou knihovnu a načíst do paměti programu. Samotnou kompilaci by však musel řešit kompilátor na uživatelském počítači. Bylo by však velmi nepraktické vyžadovat po uživateli, aby měl nainstalovaný nějaký kompilátor pro C++, neboť takový uživatel většinou ani neví, co to kompilátor je. Druhou možností je přepisovací pravidlo interpretovat v paměti programu obdobně, jako se interpretují běžné programy - pomocí abstraktního syntaktického stromu, který se bude za běhu programu vyhodnocovat. Tento způsob realizace přepisovacích pravidel tak nevyžaduje žádný další software instalovaný na počítači uživatele. Interpret je také nezávislý na platformě a program tak bude snadno přenositelný i na jiný operační systém. Nevýhodou je , že program bude pomalejší než v případě kompilované knihovny. Obecně uváděné poměry mezi rychlostmi interpretovaného a kompilovaného programu se uvádí 10:1 až 100:1. Také se očekávají vyšší paměťové nároky.

Z důvodu uživatelské přívětivosti jsem se rozhodl pro druhou variantu. Součástí programu tedy bude interpret přepisovacích pravidel. Ze všech možných typů interpretů jsem zvolil interpretaci pomocí abstraktního syntaktického stromu.

Uživatelem zadaná pravidla mohou obsahovat standardní matematické operace, jako je sčítání, odečítání, násobení, dělení a zbytek po dělení. Podmínku, kdy se má dané pravidlo aplikovat, může obsahovat obvyklé relační operátory - logický and, logický or, negace a operátory porovnání. Samozřejmostí je i podpora kulatých závorek určujících prioritu vykonávaných operací. Důležité jsou i hranaté závorky, v nichž je možné indexovat jednotlivé cifry přepisovacích pravidel obdobně jako se indexují prvky v poli v různých programovacích jazycích.

Strukturu přepisovacího pravidla můžeme popsat následující gramatikou zapsanou v Backus-Naurově formě :

⟨*rule*⟩ ::= ‘`if`’ ⟨*expression*⟩ ‘`then`’ ‘`apply`’ ‘`rule`’ ‘`[`’ ⟨*vector*⟩ ‘`]`’ ‘`at`’ ‘`position`’ ⟨*number*⟩ ‘`decreasing`’ ⟨*ident*⟩

⟨*expression*⟩ ::= ⟨*simple-expression*⟩ ⟨*relOp*⟩ ⟨*simple-expression*⟩ | ‘`(`’ ⟨*expression*⟩ ‘`)`’ ⟨*relOp*⟩ ‘`(`’ ⟨*expression*⟩ ‘`)`’

⟨*simple-expression*⟩ ::= ⟨*simple-expression*⟩ ⟨*addOp*⟩ ⟨*term*⟩ | ⟨*term*⟩

⟨*term*⟩ ::= ⟨*term*⟩ ⟨*mulOp*⟩ ⟨*factor*⟩ | ⟨*factor*⟩

⟨*factor*⟩ ::= ‘`abs (`’ ⟨*simple-expression*⟩ ‘`)`’ | ‘`(`’ ⟨*simple-expression*⟩ ‘`)`’ | ⟨*ident*⟩ | ⟨*number*⟩

⟨*relOp*⟩ ::= ‘`=`’ | ‘`<>`’ | ‘`<`’ | ‘`>`’ | ‘`<=`’ | ‘`>=`’ | ‘`and`’ | ‘`or`’

⟨*adOp*⟩ ::= ‘`+`’ | ‘`-`’

⟨*mulOp*⟩ ::= ‘`*`’ | ‘`/`’ | ‘`mod`’

⟨*vector*⟩ ::= ⟨*number*⟩ ⟨*vector*⟩ | ⟨*number*⟩ | ⟨*complex-number*⟩

⟨*number*⟩ ::= ‘`0`’ | [‘`-`’] ⟨*natural-number*⟩

⟨*natural-number*⟩ ::= ⟨*non-zero-cipher*⟩ {⟨*cipher*⟩}

⟨*complex-number*⟩ ::= ⟨*number*⟩ ‘`i`’ | ‘`(`’⟨*number*⟩ ⟨*adOp*⟩ ⟨*natural-number*⟩ ‘`i`’ ‘`)`’

⟨*non-zero-cipher*⟩ ::= ‘`1`’ | ‘`2`’ | ‘`3`’ | ‘`4`’ | ‘`5`’ | ‘`6`’ | ‘`7`’ | ‘`8`’ | ‘`9`’

⟨*cipher*⟩ ::= ‘`0`’ |⟨*non-zero-cipher*⟩

⟨*ident*⟩ ::= ⟨*letter*⟩ {⟨*cipher*⟩ | ⟨*letter*⟩} ‘`[`’ ⟨*number*⟩ ‘`]`’

⟨*letter*⟩ ::= [a-zA-Z]

Příkladem přepisovacího pravidla zapsaného pomocí následující gramatiky je např.

```
If abs(a[8]) >= 3 then apply rule [ 1 0 0 0 0 0 0 3 0
0 0 0 0 1 ] at position 1 decreasing a[8].
```

Nejdříve se zkontroluje podmínka, zda-li je na osmé pozici řetězce cifra v absolutní hodnotě větší nebo rovna 3. Pokud ano, tak se aplikuje pravidlo

1 0 0 0 0 0 0 3 0 0 0 0 0 0 1 na první pozici řetězce tak, aby se snížila hodnota cifry na osmé pozici řetězce (tj. pravidlo přičte nebo odečte).

Lze si povšimnout, že při syntaktické analýze shora-dolů daná gramatika zohledňuje priority operací.

5.1.1 Lexikální analýza

Samotný lexikální analyzátor jakožto konečný automat vždy načte jeden znak ze vstupu, následně se přesune do dalšího stavu a pokud je to nutné, uloží načtený znak do paměti. Přechodem do koncového stavu je vždy načten celý token, ten se uloží do seznamu tokenů a lexikální analyzátor se vrátí do svého počátečního stavu. Může se však stát, že jeden token je prefixem druhého. V tomto případě se nejdříve načte další znak ze vstupu a poté se rozhodne, zda-li se první token uloží nebo se bude pokračovat v načítání druhého.

Programová realizace lexikálního analyzátoru může mít více podob. Konečný automat lze pomocí programovacího jazyku reprezentovat dvěma způsoby. První možností je realizovat přechody automatu pomocí dvourozměrného pole a aktuální stav uložit do proměnné. V poli reprezentujícím automat jeden rozměr odpovídá aktuálnímu stavu, druhý rozměr načtenému symbolu na vstupu a výsledná hodnota v poli označuje následující stav. Tato implementace je pro složité jazyky poměrně časově náročná, ale zdrojový kód je kratší než u druhé možnosti, jak implementovat automat. Tou je realizovat aktuální stav pomocí pozice v programu (zdrojovém kódu) a jednotlivé přechody uskutečnit pomocí skoků v programu. Takto implementovaný automat má sice delší kód, ale je paměťové úspornější.

Jelikož reprezentace pravidel je poměrně jednoduchá, nejsou mezi možnostmi implementace lexikálního analyzátoru prakticky žádné rozdíly. Jelikož se mi realizace stavu lexikálního analyzátoru pomocí pozice programu jeví jako přehlednější, rozhodl jsem se pro tuto možnost.

Možný automat realizující lexikální analyzátor pro přepisovací pravidla je na obrázku 5.1. V přepisovacích pravidlech se mohou vyskytovat běžné relační operátory – $=, <>, <, >, <=, >=, and, or$, kde $<>$ značí nerovnost. Tyto operátory jsou přijímány ve stavech q_i na obrázku 5.1. Matematické operátory, které je možné v pravidlech využít jsou $+, -, *, /, mod, abs()$. Při načítání identifikátoru řetězce je často žádoucí odkázat na konkrétní prvek řetězce. To lze zapsat pomocí operátoru []. Z povahy přepisovacích pravidel nemá smysl odkazovat na pozici řetězce složeným výrazem nebo jiným identifikátorem. Z tohoto důvodu lexikální analyzátor načte nejen identifikátor, ale i celočíselný odkaz na konkrétní pozici v řetězci. To lehce zjednoduší syntaktický analyzátor. Po načtení identifikátoru je nutné rozhodnout, zda-li se nejedná o klíčové slovo. Klíčová slova na základě gramatiky 5.1 jsou if, *apply*, *rule*, *at*, *position*, *decreasing*, *abs*, *mod*, *and*, *or*. Mezi klíčová slova je nutno zahrnout i relační a matematické operátory, které jsou složené z písmen, neboť jinak by je lexikální analyzátor rozpoznal jako identifikátory.

Dalším zjednodušením syntaktického analyzátoru je načítání posloupnosti komplexních celých čísel pravidla přímo v lexikálním analyzátoru. Toto načítání realizují stavy s_i na obrázku 5.1. Pro snazší identifikaci bude tato posloupnost uzavřena do závorek []. Pokud by nebyly závorky použity, nešlo by pomocí konečného automatu rozpoznat, zda se načítá nějaký výraz nebo konkrétní pravidlo (posloupnost komplexních celých čísel).

Také si lze povšimnout, že pokud se výraz s komplexním číslem (např. $(-2 + 3i)$) objeví v samotné podmínce pravidla, lexikální analyzátor rozdělí tento výraz na symboly $(, -, 2, +, 3, i,)$ pomocí přechodů mezi stavy s označením q_i. Pokud se stejný výraz objeví v samotném pravidle (tj. bude se vyskytovat mezi závorkami []), lexikální analyzátor ho rozpozná jako komplexní číslo s hodnotou $-2 + 3i$, které je součástí pravidla. Důvod tohoto řešení je prostý – v podmínce nelze toto načítání použít, protože bez syntaktické analýzy nelze zjistit, jestli operátory $+, -$ jsou součástí komplexního čísla, nebo jde o operátory mezi složitějšími výrazy. Naproti tomu v samotném pravidle žádné složité výrazy nejsou a proto je vždy jasné, k čemu přiřadit operátory $+, -$.

5.1.2 Syntaktická analýza

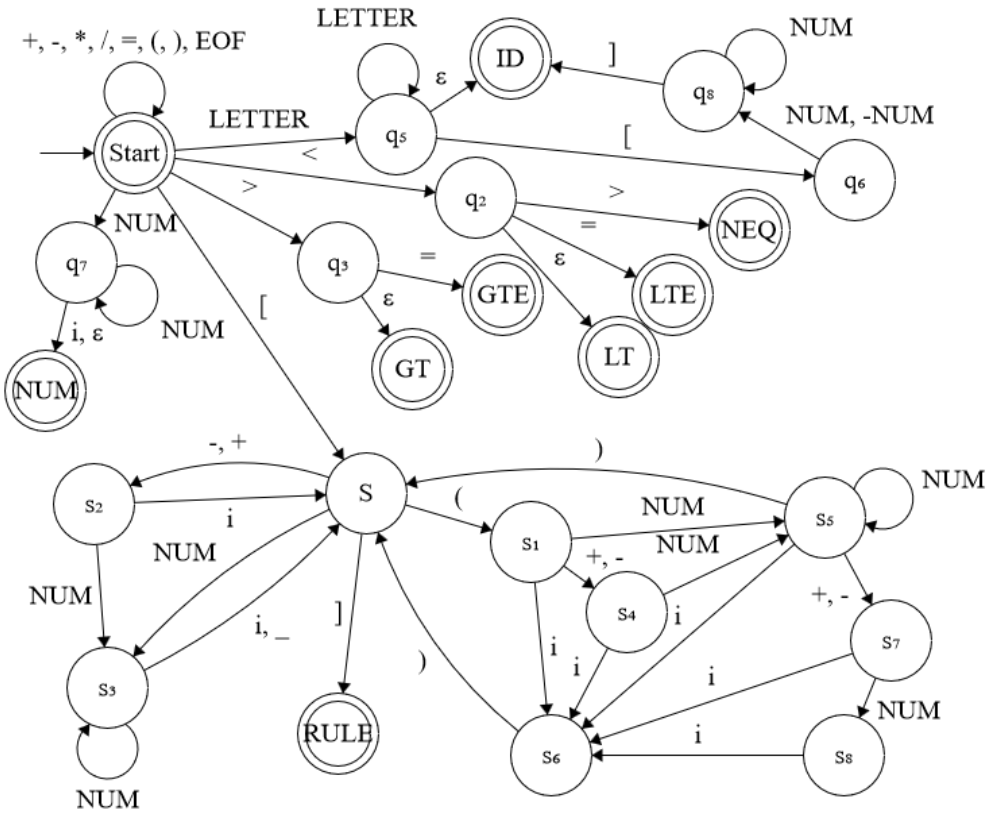
Syntaktickou analýzu je možné provést LR-analyzátorem a LL-analyzátorem. Přestože LR-analýza je silnější, pro potřeby naší implementace je LL-analýza postačující a proto bude syntaktický analyzátor implementován metodou shora dolů.

Jednou z možností, jak implementovat LL-analýzu je pomocí rekurzivního sestupu. V této metodě se využívá několika funkcí. První funkcí je srovnání, která porovná symbol na vstupu s očekávaným symbolem. Pokud se symboly shodují, odebere se ze zásobníku a ze vstupu tento symbol. V opačném případě se signalizuje chyba. Další funkce se implementují pro jednotlivé neterminály gramatiky. Každá z těchto funkcí analyzuje jeden neterminál a může volat další funkce syntaktické analýzy. Právě rekurzivní volání (přímé nebo nepřímé) těchto funkcí simuluje zásobník zásobníkového automatu.

Uvažujme gramatiku $G = (N, T, P, S)$ a neterminál $A \in N$ s jedním pravidlem $A \to aBb$, kde $a \in T, B \in N$. Funkce pro analýzu symbolu A může vypadat následovně:

```
void A (void) {
compare(a)
B()
compare(b)
}
```

Funkce *compare(sym)* je funkce pro srovnání symbolu *sym* a B je funkce pro analýzu neterminálu B. V případě, že gramatika G obsahuje více pravidel



Obrázek 5.1: Automat realizující lexikální analyzátor přepisovacích pravidel

pro neterminál A, tedy pravidla ve tvaru $A \to \alpha_1|\alpha_2|\cdots|\alpha_k$, je potřeba rozlišit, podle kterého pravidla se má analýza provést. Analýza neterminálu A pak může být implementována jako

```
void A (void) {
if  input_symbol in Φ(A, α_i)  then
   /* Implementace analýzy řetězce α_i */
end if
}
```

kde *input_symbol* je aktuální symbol (token) na vstupu a funkce $\Phi(A, \alpha_i)$ je definována jako

$$\Phi(A, \alpha_i) = \begin{cases} FIRST(\alpha) & \text{pokud } \epsilon \notin FIRST(\alpha), \\ FOLLOW(A) \cup (FIRST(\alpha) \setminus \{\epsilon\}) & \text{pokud} \epsilon \in FIRST(\alpha). \end{cases}$$

Funkce $\Phi(A, \alpha)$ definuje množinu symbolů, které se mohou vyskytovat na vstupu v případě expanze neterminálu A na řetězec α. Pokud však lze

A expandovat na prázdný řetězec, je potřeba na vstupu očekávat i symboly z $FOLLOW(A)$. Funkce $\Phi(A, \alpha_i)$ tak odpovídá záznamu v rozkladové tabulce na řádku A a sloupci $FIRST(\alpha_i)$.

Součástí implementovaného programu je syntaktická analýza pro jazyk přepisovacích pravidel. $LL(1)$ gramatika $G = (N, T, P, S)$ pro zadaný jazyk může být následující:

$$N = \{start, expression, relop, logop, expression, expression2, relexp, relexp2, simple-expression, simple-expression2, sign, addop, term, term2, mulop, factor, factor2\}$$

$$T = \{IDENT, NUMB, PLUS, MINUS, TIMES, DIVIDE, MOD, AND, OR, NOT, kwTRUE, kwFALSE, EQ, NEQ, LT, GT, LTE, GTE, LPAR, RPAR, kwIF, kwTHEN, kwAPPLY, kwRULE, kwAT, kwPOSITION, kwDECREASING, kwVALUE, kwOF, kwABS, I, RULE\}$$

Přechodová funkce P obsahuje tato pravidla:

1 $start \rightarrow$ *kwIF expression kwTHEN kwAPPLY kwRULE RULE kwAT kwPOSITION sign NUMB kwDECREASING IDENT*
2 $relop \rightarrow$ *EQ*
3 $relop \rightarrow$ *NEQ*
4 $relop \rightarrow$ *LT*
5 $relop \rightarrow$ *GT*
6 $relop \rightarrow$ *LTE*
7 $relop \rightarrow$ *GTE*
8 $logop \rightarrow$ *AND*
9 $logop \rightarrow$ *OR*
10 $expression \rightarrow relexp\ expression2$
11 $expression2 \rightarrow logop\ relexp\ expression2$
12 $expression2 \rightarrow \epsilon$
13 $relexp \rightarrow simple\text{-}expression\ relexp2$
14 $relexp2 \rightarrow relop\ simple\text{-}expression\ relexp2$
15 $relexp2 \rightarrow \epsilon$
16 $simple\text{-}expression \rightarrow sign\ term\ simple\text{-}expression2$
17 $simple\text{-}expression2 \rightarrow addop\ sign\ term\ simple\text{-}expression2$
18 $simple\text{-}expression2 \rightarrow \epsilon$
19 $term \rightarrow factor\ term2$
20 $term2 \rightarrow mulop\ factor\ term2$
21 $term2 \rightarrow \epsilon$
22 $factor \rightarrow$ *IDENT*

23 $factor \rightarrow kwTRUE$
24 $factor \rightarrow kwFALSE$
25 $factor \rightarrow LPAR\ expression\ RPAR$
26 $factor \rightarrow kwABS\ LPAR\ expression\ RPAR$
27 $factor \rightarrow NOT\ factor$
28 $factor \rightarrow NUMB\ factor2$
29 $factor \rightarrow I$
30 $factor2 \rightarrow I$
31 $factor2 \rightarrow \epsilon$
32 $addop \rightarrow PLUS$
33 $addop \rightarrow MINUS$
34 $mulop \rightarrow TIMES$
35 $mulop \rightarrow DIVIDE$
36 $mulop \rightarrow MOD$
37 $sign \rightarrow PLUS$
38 $sign \rightarrow MINUS$
39 $sign \rightarrow \epsilon$

Sestavením množin $FIRST$ pro pravé strany pravidel dostaneme tyto množiny (pro přehlednost $FIRST(i)$ je množina odpovídající $FIRST$ pravé strany i-tého pravidla gramatiky G):

$FIRST(1) = \{kwIF\}$
$FIRST(2) = \{EQ\}$
$FIRST(3) = \{NEQ\}$
$FIRST(4) = \{LT\}$
$FIRST(5) = \{GT\}$
$FIRST(6) = \{LTE\}$
$FIRST(7) = \{\}$
$FIRST(8) = \{AND\}$
$FIRST(9) = \{OR\}$
$FIRST(10) = FIRST(13) = FIRST(16) = \{PLUS, MINUS, IDENT,$
$kwTRUE, kwFALSE, LPAR, kwABS, NOT, NUMB, I\}$
$FIRST(11) = \{AND, OR\}$
$FIRST(12) = FIRST(15) = FIRST(18) = FIRST(21) = FIRST(31) =$
$= FIRST(39) = \{\epsilon\}$
$FIRST(14) = \{EQ, NEQ, LT, GT, LTE, GTE\}$
$FIRST(17) = \{PLUS, MINUS\}$
$FIRST(19) = \{IDENT, kwTRUE, kwFALSE, LPAR, kwABS, NOT,$
$NUMB, I\}$
$FIRST(20) = \{TIMES, DIVIDE, MOD\}$
$FIRST(22) = \{IDENT\}$
$FIRST(23) = \{kwTRUE\}$
$FIRST(24) = \{kwFALSE\}$
$FIRST(25) = \{LPAR\}$

$FIRST(26) = \{kwABS\}$
$FIRST(27) = \{NOT\}$
$FIRST(28) = \{NUMB\}$
$FIRST(29) = FIRST(30) = \{I\}$
$FIRST(32) = FIRST(37) = \{PLUS\}$
$FIRST(33) = FIRST(38) = \{MINUS\}$
$FIRST(34) = \{TIMES\}$
$FIRST(35) = \{DIVIDE\}$
$FIRST(36) = \{MOD\}$

Množiny $FOLLOW$ pro všechny neterminály jsou následující:

$FOLLOW(start) = \{\epsilon\}$
$FOLLOW(relop) = FOLLOW(logop) = \{PLUS, MINUS, IDENT,$
$kwTRUE, kwFALSE, LPAR, kwABS, NOT, NUMB, I\}$
$FOLLOW(expression) = FOLLOW(expression2) = \{kwTHEN, RPAR\}$
$FOLLOW(relexp) = FOLLOW(relexp2) = \{kwTHEN, RPAR, AND, OR\}$
$FOLLOW(simple - expression) = FOLLOW(simple - expression2) =$
$= \{kwTHEN, RPAR, EQ, NEQ, LT, GT, LTE, GTE, AND, OR\}$
$FOLLOW(term) = FOLLOW(term2) = \{kwTHEN, RPAR, PLUS,$
$MINUS, EQ, NEQ, LT, GT, LTE, GTE, AND, OR\}$
$FOLLOW(factor) = FOLLOW(factor2) = \{kwTHEN, RPAR, TIMES,$
$DIVIDE, MOD, PLUS, MINUS, EQ, NEQ, LT, GT, LTE, GTE, AND, OR\}$
$FOLLOW(addop) = \{PLUS, MINUS, IDENT, kwTRUE, kwFALSE,$
$LPAR, kwABS, NOT, NUMB, I\},$
$FOLLOW(mulop) = \{IDENT, kwTRUE, kwFALSE, LPAR, kwABS,$
$NOT, NUMB, I\}$
$FOLLOW(sign) = \{NUMB, IDENT, kwTRUE, kwFALSE, LPAR,$
$kwABS, NOT, I\}$

Na základě množin $FIRST$ a $FOLLOW$ sestrojíme rozkladovou tabulku 5.3, 5.4, 5.5, 5.6, a 5.7. Pokud v nějaké buňce tabulky není uvedena hodnota, a syntaktický analyzátor ji bude vyžadovat, dojde k signalizaci chyby.

Tabulka 5.3: Rozkladová tabulka pro gramatiku popisující jazyk přepisovacích pravidel (1. část)

	IDENT	NUMB	PLUS	MINUS	TIMES	DIVIDE
start						
relop						
logop						
expression	10	10	10	10		
expression2						
relexp	13	13	13	13		
relexp2						
simple-expression	16	16	16	16		
simple-expression2			17	17		
sign	39	39	37	38		
addop			32	33		
term	19	19				
term2			21	21	20	20
mulop					34	35
factor	22	28				
factor2			31	31	31	31

Tabulka 5.4: Rozkladová tabulka pro gramatiku popisující jazyk přepisovacích pravidel (2. část)

	MOD	AND	OR	NOT	kwTRUE	kwFALSE	EQ
start							
relop							2
logop		8	9				
expression				10	10	10	
expression2		11	11				
relexp				13	13	13	
relexp2		15	15				14
simple-expression				16	16	16	
simple-expression2		18	18				18
sign				39	39	39	
addop							
term				19	19	19	
term2	20	21	21				21
mulop	36						
factor				27	23	24	
factor2	31	31	31				31

Tabulka 5.5: Rozkladová tabulka pro gramatiku popisující jazyk přepisovacích pravidel (3. část)

	NEQ	LT	GT	LTE	GTE	LPAR	RPAR	kwIF
start								1
relop	3	4	5	6	7			
logop								
expression						10		
expression2							12	
relexp						13		
relexp2	14	14	14	14	14		15	
simple-expression						16		
simple-expression2	18	18	18	18	18		18	
sign						39		
addop								
term						19		
term2	21	21	21	21	21		21	
mulop								
factor						25		
factor2	31	31	31	31	31		31	

Tabulka 5.6: Rozkladová tabulka pro gramatiku popisující jazyk přepisovacích pravidel (4. část)

	kwTHEN	kwAPPLY	kwRULE	kwAT	kwPOSITION
start					
relop					
logop					
expression					
expression2	12				
relexp					
relexp2	15				
simple-expression					
simple-expression2	18				
sign					
addop					
term					
term2	21				
mulop					
factor					
factor2	31				

Tabulka 5.7: Rozkladová tabulka pro gramatiku popisující jazyk přepisovacích pravidel (5. část)

	kwDECREASING	kwABS	I	RULE	ϵ
start					
relop					
logop					
expression		10	10		
expression2					
relexp		13	13		
relexp2					
simple-expression		16	16		
simple-expression2					
sign		39	39		
addop					
term		19	19		
term2					
mulop					
factor		26	29		
factor2			30		

5.1.3 Konstrukce derivačního stromu

Postup pro syntaktickou analýzy shora dolů z kapitoly 5.1.2 zjistí, jestli vstup patří do daného jazyka, nevytvoří však strukturu, se kterou je možné dále pracovat. K tomu je nutné přidat k syntaktické analýze výstup. To lze popsat **překladovou gramatikou**. Překladová gramatika PG je uspořádaná pětice $PG = (N, T, D, R, S)$, kde N je konečná množina neterminálních symbolů, T je konečná množina vstupních symbolů, D je konečná množina výstupních symbolů, R je konečná množina pravidel ve tvaru $A \rightarrow \alpha$, kde $A \in N, \alpha \in (N \cup T \cup D)^*$ a S je počáteční symbol gramatiky. Překladová gramatika tak pouze rozšiřuje bezkontextovou gramatiku o nějaký výstup. Jde o nejjednodušší popis formálního překladu (binární relace $Z \subseteq L \times V$, která vstupnímu řetězci z jazyka L nad abecedou T přiřadí výstupní řetězec z jazyka V nad abecedou D).

Pro potřeby této práce je nezbytné popsat formální překlad, který nelze popsat běžnou překladovou gramatiku. Z tohoto důvodu překladovou gramatiku rozšíříme o atributy. Ty si lze představit jako proměnné. V našem případě atributy využijeme i pro uložení výstupu. **Atributová překladová gramatika** je čtveřice $APG = (PG, A, V, F)$, kde $PG = (N, T, D, R, S)$ je překladová gramatika, A je množina atributů, která se dělí na dědičné atributy I a syntetizované atributy S, přičemž $I \cap S = \emptyset$. V je zobrazení, které každému symbolu $X \in N \cup T \cup D$ přiřazuje množinu atributů $A(X) \in A$. Vstupním symbolů jsou přiřazeny pouze syntetizované atributy a všem výstupním symbolům jsou

přiřazeny pouze dědičné atributy. F je množina sémantických pravidel, kterými jsou definovány hodnoty atributů. Přiřazování hodnot atributů nějakého symbolu v pravidle z R je možné pouze na základě hodnot atributů ostatních symbolů stejného pravidla.

Pokud je vypočítáván atribut symbolu na levé straně pravidla, jedná se o syntetizovaný atribut. Při výpočtu atributu symbolu na pravé straně pravidla jde o dědičný atribut.

Pomocí atributovaných překladových gramatik lze podle [2] přijímat všechny rekurzivně spočetné jazyky.

Příklad 5.1. *Uvažujme jazyk L, který generuje jednoduché matematické výrazy se sčítáním a násobením čísel. Pokud chceme vytvořit formální překlad, který každému takovému výrazu přiřadí výslednou hodnotu val, můžeme využít atributovou překladovou gramatiku $APG = (PG, A, V, F)$.*

Překladová gramatika může být například $PG = (\{S, At, B, C\}, \{+, *, num\}, \{v\}, P, S)$ s pravidly P:

$$\begin{aligned} S &\to Av \\ A &\to A + B \\ A &\to B \\ B &\to B * C \\ B &\to C \\ C &\to num \end{aligned}$$

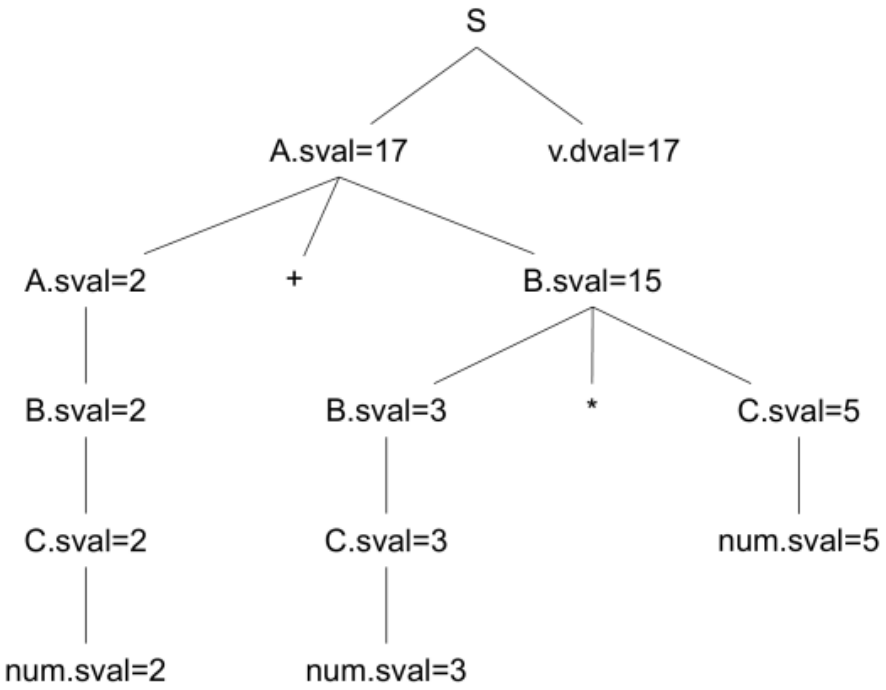
Množinu atributů lze definovat jako $At = \{v.dval, A.sval, B.sval, C.sval, num.sval\}$. Přiřazení atributů symbolům V je pak určeno:

symbol	dědičné atributy	syntetizované atributy
num		sval
v	dval	
A		sval
B		sval
C		sval

Sémantická pravidla F pak lze vytvořit takto:

syntaktické pravidlo	sémantické pravidlo
$S \to Av$	$v.dval = A.sval$
$A \to A + B$	$A^0.sval = A^1.sval + B.sval$
$A \to B$	$A.sval = B.sval$
$B \to B * C$	$B^0.sval = B^1.sval * C.sval$
$B \to C$	$B.sval = C.sval$
$C \to num$	$C.sval = num.sval$

Atributový derivační (překladový) strom pro výraz $2+3*5$ *pak může vypadat jako na obrázku 5.2:*



Obrázek 5.2: Překladový strom pro výraz $2 + 3 * 5$ z příkladu 5.1

Jelikož výpočet atributů probíhá od listů, dochází k zohlednění priorit operací a výsledná hodnota je 17 *a nikoli* 25*, jak by tomu bylo, kdyby uzly pro sčítání a násobení byly v překladovém stromu na stejné úrovni.*

Obecně můžou být atributy na sobě závislé tak, že tvoří cyklus a nelze tak hodnoty všech atributů vypočítat během jednoho průchodu. V případě jazyka přepisovacích je však popisující gramatika $LL(1)$ se syntaktickými pravidly ve tvaru $X_0 \to X_1 X_2 \ldots X_n$ a splňuje tyto vlastnosti:

- Každý dědičný atribut symbolu X_k pro $1 \leq k \leq n$ závisí pouze na dědičných atributech symbolu X_0 a na atributech symbolů X_i, kde $1 \leq i < k$
- Každý syntetizovaný atribut $X_0.s$ závisí pouze na dědičných atributech symbolu X_0 a na atributech symbolů pravé strany pravidla

Taková gramatika se označuje jako **L-atributová** a každý libovolný překladový strom pak lze vyhodnotit jedním průchodem. Speciálním typem L-atributové gramatiky je gramatika, která obsahuje pouze syntetizované atributy a nazývá se **S-atributová** gramatika.

Samotné rozšíření syntaktické analýzy z kapitoly 5.1.2 o atributy je poměrně jednoduché. Dědičné atributy neterminálů jsou vstupními parametry funkcí. Syntetizované atributy neterminálů jsou výstupní parametry funkcí. V případě syntetizovaných atributů vstupních terminálů je nutné rozšířit funkci srovnání o výstupní parametry. Pokud výstupní terminály mají přidruženy dědičné atributy, je potřeba upravit funkci srovnání o vstupní parametry. Do jednotlivých funkcí se pak na příslušná místa přidá vlastní výpočet atributů podle sémantických pravidel.

Pro konstrukci syntaktického stromu lze využít atributy. Každému neterminálu se přiřadí syntetizovaný atribut, který reprezentuje uzel syntaktického stromu ohodnoceného daným neterminálem. Pro zaručení jednoprůchodové sémantické analýzy se v případě binárních operátorů pak využijí dědičné atributy pro pravé operandy.

Příklad 5.2 popisuje konstrukci syntaktické a sémantické analýzy rekurzivním sestupem.

Příklad 5.2. *Uvažujme jazyk L z příkladu 5.1, který generuje jednoduché matematické výrazy se sčítáním a násobením čísel. Povšimněme si, že gramatika z příkladu 5.1 není LL(1). Transformací získáme* $LL(1)$ *gramatiku* $APG = (PG, At, V, F)$*, kde překladová gramatika* $PG = (\{S, A, A', B, B', C\}, \{+, *, num\}, \emptyset, P, S)$ *má tyto atributy:*

symbol	dědičné atributy	syntetizované atributy
num		sval
A		sval
A’	dval	sval
B		sval
B’	dval	sval
C		sval

Syntaktická a sémantická pravidla gramatiky jsou zadána následovně:

syntaktické pravidlo	sémantické pravidlo
$A \to BA'$	$A'.dval = B.sval$ $A.sval = A'.sval$
$A' \to +BA'$	$A'^1.dval = A^0.sval + B.sval$ $A'^0.sval = A'^1.sval$
$A' \to \epsilon$	$A'.sval = A'.dval$
$B \to CB'$	$B'.dval = C.sval$ $B.sval = B'.sval$
$B' \to *CB'$	$B'^1.dval = B^0.sval + C.sval$ $B'^0.sval = B'^1.sval$
$B' \to \epsilon$	$B'.sval = B'.dval$
$C \to num$	$C.sval = num.sval$

Překladová tabulka je znázorněna v tabulce 5.8:

Tabulka 5.8: Rozkladová tabulka pro jazyk z příkladu 5.2

	num	*	+	ϵ
A	$A \to BA'$			
A'			$A' \to +BA'$	$A' \to \epsilon$
B	$B \to CB'$			
B'		$B' \to *CB'$	$B' \to \epsilon$	$B' \to \epsilon$
C	$C \to num$			

Pseudokód funkcí pro rekurzivní sestup pak může vypadat následovně:

```
int A (void) {
if input_symbol = num then
  int b ← B()
  int a ← A'(b)
  return a
else
  return ERROR
end if
}
```

```
int A' (int in) {
if input_symbol = + then
  compare(+)
  int b ← B()
  int a ← A'(in + b)
  return a
else if input_symbol = EOF then
  return in
else
  return ERROR
end if
}

int B (void) {
if input_symbol = num then
  int c ← C()
  int b ← B'(c)
  return b
else
  return ERROR
end if
}

int B' (int in) {
if input_symbol = * then
  compare(*)
  int c ← C()
  int b ← B'(in * c)
  return b
else if input_symbol = EOF or + then
  return in
else
  return ERROR
end if
}

int C (void) {
if input_symbol = num then
  int val ← compare(num)
  return val
else
  return ERROR
end if
}
```

Kapitola 6

Implementace

6.1 Popis tříd

Tato kapitola popisuje jednotlivé části implementace.

6.1.1 Číselné řetězce

NumString je třída reprezentující číselný řetězec, která byla implementována v [15]. V této práci bylo implementováno rozšíření o práci s komplexními celými čísly oproti obyčejným celým číslům. Všechny symboly (komplexní celá čísla), jsou ukládány v dynamicky alokovaném poli *numbers* délky *max_length*. Počet symbolů v poli udává proměnná *length*. Pozici zlomkové tečky značí proměnná *frac_point*. Třída obsahuje následující metody:

- ***int readString(istream & input)*** je metoda, která načte ze vstupního streamu *input* čísla a uloží je do pole *numbers*.
- ***void debug(ostream & output)*** je metoda, která slouží pro výpis číselného řetězce do výstupního streamu *output*.
- ***int getMaxHeight(void);*** je metoda, která vrací jednotkovou výšku daného řetězce
- ***bool isMaxHeight(int number)*** je metoda, která vrací *true*, pokud je jednotková výška číselného řetězce nejvýše *number*. Je-li jednotková výška řetězce vyšší, metoda vrátí *false*.
- ***int getWeight(void)*** je metoda, která vrací váhu řetězce.
- ***int getLength(void)*** je metoda, která vrací počet platných číslic v textovém řetězci.
- ***int* getNumbers(void)*** je metoda, která vrací ukazatel na pole cifer *numbers*.

- ***void add(complex <int> number)*** je metoda, která přidá do pole *numbers* komplexní celé číslo *number*. V případě potřeby dojde z rozšíření pole *numbers*.
- ***void addVector(int offset, string vector, int op)*** je metoda, která za předpokladu *op=ADD* (1) přičte k číselnému řetezci řetězec *vector* na pozici *offset*. Je-li *op=SUBSTRACT* (2), dojde k odečtení řetězce *vector* na pozici *offset*.
- ***void resize(int direction)*** je metoda, která rozšíří pole *numbers* doleva(*direction=LEFT*(1)) nebo doprava(*direction=RIGHT*(2))
- ***int getFracPoint()*** je metoda, která vrátí pozici zlomkové tečky.
- ***void setFracPoint(int n)*** je metoda, která nastaví pozici zlomkové tečky *frac_point* na hodnotu *n*.
- ***int getLength()*** je metoda, která vrátí velikost pole *numbers*.
- ***complex<int> *getNumbers()*** je metoda, která vrátí pole *numbers*.

6.1.2 Syntaktický strom

Syntaktický strom, který popisuje strukturu pravidla a je tvořen syntaktickým analyzátorem, je realizován pomocí abstraktní třídy *Node*. Tato třída obsahuje následující metody:

- ***virtual void debug (ostream & out)*** vypíše informace o daném uzlu na výstupní stream *out*. Podle typu uzlu se výpis liší (např. výpis identifikátoru vypadá jinak než výpis binárního operátoru).
- ***virtual complex<int> evaluate (complex<int> * array , int len, int pos)*** vrací hodnotu daného uzlu. V případě výrazu vrátí hodnotu výrazu, v případě podmínky vyhodnotí podmínku a vrátí informaci zda byla podmínka provedena. Některé metody dědící z *Expr* využívají ukazatel na řetězec komplexních celých čísel *array* a informace o jeho délce *len* a případném indexu *pos* konkrétní pozice v daném řetězci.

Implementace v jazyce C++ využívá dědičnosti a dělí uzly do několika typů. Z této třídy tak dědí další virtuální třída *Expr*, která reprezentuje libovolný výraz.

První třídou, která dědí z *Expr* je třída *Var*, která popisuje identifikátor a uchovává si adresu do tabulky symbolů, na které se daný identifikátor nachází. Metoda *evaluate* v této třídě vypisuje konkrétní hodnotu identifikátoru. Metoda pro výpis vypíše jméno tohoto identifikátoru a případný ofset, pokud se jedná o ukazatel (např. výraz $a[5]$ pro pátou pozici v řetězci a).

Následující třídou, která dědí z třídy *Expr*, je třída *Numb*. Tato třída uchovává informaci o libovolném (komplexním) čísle, které se vyskytne v nějakém přepisovacím pravidle. Metoda *evaluate* vrací hodnotu čísla a metoda *debug* ji vypíše. Klíčová slova *true* a *false* se ukládají jako instance této třídy s hodnotou 1, resp 0. Tím je specifikováno chování progamu v případě, že uživatel zadá nesmyslný výraz typu $true+1$. Tomuto odpovídá výraz, který při vyhodnocení vrátí hodnotu 2.

Další, o něco složitější, třídou dědící z *Expr* je třída *Bop*. Tato třída odpovídá uzlu binárního operátoru v syntaktickém stromu. Třída si uchovává typ binárního operátoru a ukazatele na levý a pravý operand (resp. uzly, které je reprezentují). Metoda pro výpis využívá infixovou notaci a proto se nejdříve vypíše levý operand následovaný operátorem a nakonec se vypíše pravý operand. Vyhodnocování probíhá tak, že se nejdříve vyhodnotí oba operandy a na ně se následně použije daná operace.

Třída *UnMinus* také dědí z třídy *Expr* a používá se pro uložení uzlu reprezentujícího unární minus. Třída si uchovává ukazatel na nějaký výraz, který adekvátně vyhodnocuje metodou *evaluate* a vypisuje metodou *debug*.

Další třída dědící z *Expr* je *Abs*. Ta popisuje uzel derivačního stromu reprezentující absolutní hodnotu nějakého výrazu. Uchovává si ukazatel na tento výraz a pomocí metody *debug* výraz *expr* vypíše se ve formě *abs(expr)*. Metoda *evaluate* vrátí absolutní hodnotu uloženého výrazu.

Poslední třídou, která dědí z *Expr* je třída *Not*, která realizuje negaci nějakého výrazu, na který si ukládá ukazatel. Funkce *evaluate* vrací hodnotu 1, pokud je hodnota uloženého výrazu 0, v opačném případě vrací 1. Metoda *debug* pro uložený výraz *expr* vypíše $not(expr)$.

Následující třídou, která dědí z *Node* je třída *If*. Ta reprezentuje kořen derivačního stromu přepisovacího pravidla. Uchovává si ukazatel na výraz *cond*, přepisovací pravidlo *rule* a jeho délku *len*, pozici *pos*, na které se má pravidlo aplikovat, a ukazatel na identifikátor *decrease*, která se má aplikací pravidla zmenšit. Metoda *debug* by pak měla vypsat celé přepisovací pravidlo včetně podmínky v takové formě, ve které je možné jej programem znovu načíst. Metoda *evaluate* vyhodnotí podmínku *cond* a pokud je splněna, aplikuje přepisovací pravidlo *rule* na pozici *pos* tak, aby se snížila hodnota identifikátoru *decrease*. V případě, že uživatel zadá takové přepisovací pravidlo, že přičtení ani odečtení pravidla nevede ke snížení hodnoty, pravidlo se neaplikuje. V případě, že je podmínka *cond* splněna a pravidlo bylo aplikováno je návratová hodnota 1, v opačném případě 0.

6.1.3 Seznam přepisovacích pravidel

Seznam přepisovacích pravidel je implementován pomocí třídy *RuleList*. Tato třída si uchovává ukazatele na kořeny derivačních stromů jednotlivých pravidel. Ty jsou uloženy v dynamicky alokovaném poli a proto je tak nutné

uchovávat informaci o aktuálním počtu pravidel a maximálním počtu pravidel, které je možné do alokované paměti uložit.

Ve třídě *RuleList* jsou implementovány následující metody:

- ***void debug (ostream & out)*** vypíše všechna pravidla do výstupního streamz *out*. Metoda využívá metody *debug* v uzlech syntaktického stromu přepisovacích pravidel.
- ***bool getRulesFile(string file)*** načítá sérii pravidel ze souboru *file*. Načítání probíhá provedením lexikální a syntaktické analýzy na každý řádek načítaného souboru. V případě úspěšného načtení pravidel se nastaví ukazatele na tato pravidla, upraví se počet načtených pravidel a funkce vrátí $false$. Pokud dojde během načítání k chybě, funkce vrací *true*.
- ***bool getRule(void)*** se pokouší načíst pravidlo ze standardního vstupu. Načtený řetězec analyzuje a v případě úspěchu přidá nové pravidlo na konec seznamu pravidel a vrátí $false$. V případě chyby metoda vrátí *true*.
- ***void deleteRules(void)*** smaže všechna načtená pravidla a nastaví počet načtených pravidel na 0. Funkce nemaže pole ukazatelů na pravidla, ani neupravuje hodnotu maximálního počtu načtených pravidel.
- ***void setMaxLen(int new_len)*** slouží k rozšíření dynamicky alokovaného pole ukazatelů na přepisovací pravidla. Funkce vytvoří větší pole o velikosti *new_len*, zkopíruje do něj obsah původního pole a následně původní pole smaže.
- ***int getRulesNr(void)*** vrací počet aktuálně načtených pravidel.
- ***int evaluate(int rule_nr, complex<int>* num_string, int length, int position)*** vyhodnutí pravidlo *rule_nr* na pozici *position* číselného řetězce *num_string* délky *length*. V případě chyby vrací metoda *true*, jinak *false*.
- ***int iterateRule(int rule_nr, complex<int>* num_string, int length)*** iteruje přes všechny pozice vyhodnocování pravidla *rule_nr* číselného řetězce *num_string* délky *length*. V případě chyby vrací metoda *true*, jinak *false*.
- ***int iterateAllRule(complex<int>* num_string, int length)*** se pokouší aplikovat všechna pravidla postupně přes všechny pozice číselného řetězce *num_string* délky *length*. V případě chyby vrací metoda *true*, jinak *false*.
- ***int iterateUntilLimit(NumString * ns, int limit, int time)*** iteruje všechna pravidla přes všechny pozice číselného řetězce *ns*, dokud

není splněna podmínka na maximální cifru *limit* nebo nedojte k překročení časového limitu *time*. V této metodě dochází k pravidelné kontrole, zdali se číselný řetězec za několik posledních iterací změnil a případně dojde k rozšíření řetězce nulami zleva a zprava.

6.1.4 Lexikální analyzátor

Lexikální analyzátor je implementací konečného automatu 5.1 a načítá jednotlivé tokeny vstupního pravidla. Soubor *lexan.h* obsahuje výčet všech lexikálních symbolů *LexSymbol*, které automat rozpoznává. Mezi tyto symboly patří *IDENT, NUMB, PLUS, MINUS, TIMES, DIVIDE, MOD, AND, OR, NOT, kwTRUE, kwFALSE, EQ, NEQ, LT, GT, LTE, GTE, LPAR, RPAR, kwIF, kwTHEN, kwAPPLY, kwRULE, ERR, kwAT, kwPOSITION, kwDECREASING, kwABS, I, RULE*. Je zde i další výčet, který rozlišuje typ symbolu na vstupu. Rozlišujeme typy *LETTER, NUMBER, WHITE_SPACE, END, NO_TYPE*, pokud je na vstupu písmeno, resp. číslo, bílý znak, znak konce souboru nebo jiný symbol. V lexikálním analyzátoru nesmí chybět prostor pro uložení posloupnosti tokenů, v této implementaci je tento prostor tvořen čítačem a statickým polem struktur *LexElem* s implicitní délkou 500 položek, který by měla být dostačující pro většinu pravidel. Je velmi pravděpodobné, že při využití delších pravidel by samotné přepisování trvalo natolik dlouho, že by byl program prakticky nepoužitelný a byly by nutné složité optimalizace abstraktního syntaktického stromu.

Struktura *LexElem* obsahuje informaci o typu lexikálního symbolu, reálnou a imaginární část (pokud jde o číslo), jméno (pokud se jedná o identifikátor), a posloupnost komplexních celých čísel (pokud jde o pravidlo). Jedinou významnou funkcí, kterou tato struktura obsahuje je funkce pro výpis. Ačkoli samotná aplikace jednotlivé tokeny nevypisuje, lze funkci využít pro ladění při případném rozšiřování implementovaného programu.

Druhou strukturou je *ComplexVector*, která je realizována dynamicky alokovaným polem komplexních celých čísel. Do této struktury se ukládá načítané pravidlo (stavy q_i v automatu 5.1) a teprve po jeho načtení dojde k uložení do konkrétního tokenu *LexElem*.

Další funkce v lexikálním analyzátoru jsou následující:

- ***void initLexAn (void)*** je funkce, která inicializuje všechny proměnné nutné pro běh lexikální analýzy. Funkce nastavuje čítač tokenů na 0 a v případě potřeby vyčistí pole tokenů, aby bylo možné do něj nově načtené tokeny uložit.

- ***void getinput (istream & in)*** načte symbol ze vstupního streamu *in*, rozhodne o jaký typ symbolu se jedná a tuto hodnotu nastaví.

- ***void runLexAn (istream & in)*** spustí lexikální analýzu na vstupní stream *in*. Tato funkce implemetuje automat 5.1, kde aktuální stav automatu je reprezentován pozicí v programu.

- ***LexSymbol getKW (string word)*** rozpozná, zda vstupní řetězec *word* je klíčovým slovem a vrátí token *LexSymbol*, který odpovídá tomuto klíčovému slovu. Pokud *word* není klíčovým slovem, funkce *getKW* vrátí token *IDENT*, který odpovídá identifikátoru (např. proměnné).

Výstup lexikální analýzy je uložen v poli lexikálních symbolů *LexElem*. To je jediným vstupem pro syntaktickou analýzu.

6.1.5 Syntaktický analyzátor

Syntaktický analyzátor načte posloupnost tokenů získaných z lexikální analýzy a pomocí syntaktické analýzy shora dolů rozhodne, zda je vstup syntakticky správný a v tomto případě zkonstruuje derivační strom, v opačném případě signalizuje chybu. Programová realizace využívá metodu rekurzivního sestupu, zásobníkový automat je tak simulován rekurzivním voláním funkcí. Jednotlivé funkce jsou implementovány podle rozkladové tabulky 5.3, 5.4, 5.5, 5.6 a 5.7. Syntaktická analýza obsahuje tyto funkce:

- ***void compare (LexSymbol symb)*** realizuje operaci srovnání a kontroluje, jestli je na vrcholu zásobníku symbol *symb*. Pokud je tato podmínka splněna, dojde k odebrání tohoto symbolu z vrcholu zásobníku, v opačném případě bude signalizována chyba.

- ***Node * start(void)*** realizuje expanzi neterminálu *start* v gramatice 5.1.2 pomocí pravidla 1. Návratovou hodnotou je ukazatel na kořen derivačního stromu analyzovaného pravidla v případě úspěchu, *nullptr* v případě chyby.

- ***Operators relop(void)*** realizuje expanzi neterminálu *relop* v gramatice 5.1.2 pomocí pravidel 2, 3, 4, 5, 6 a 7. Návratovou hodnotou je typ načteného relačního operátoru.

- ***Operators logop(void)*** realizuje expanzi neterminálu *logop* v gramatice 5.1.2 pomocí pravidel 8 a 9. Návratovou hodnotou je typ načteného logického operátoru.

- ***Expr * expression(void)*** realizuje expanzi neterminálu *expression* v gramatice 5.1.2 pomocí pravidla 10. Výstupem je ukazatel na strukturu popisující načtený výraz.

- ***Expr * expression(Expr * ex)*** realizuje expanzi neterminálu *expression2* v gramatice 5.1.2 pomocí pravidel 11 a 12. Vstupní parametr *ex*

slouží jako syntetizovaný atribut předávaný v pravidlech 10 a 11. Tento atribut je nezbytný pro vytvoření výrazu s levým a pravým operandem.

- ***Expr * relexp(void)*** realizuje expanzi neterminálu *relexp* v gramatice 5.1.2 pomocí pravidla 13. Výstupem je ukazatel na výraz s relačním operátorem.
- ***Expr * relexp2(Expr * ex)*** realizuje expanzi neterminálu *relexp2* v gramatice 5.1.2 pomocí pravidel 14 a 15. Vstupní parametr *ex* slouží jako syntetizovaný atribut předávaný v pravidlech 13 a 14, který je důležítý pro konstrukci výrazu s levým a pravým operátorem.
- ***Expr * simple_expression(void)*** realizuje expanzi neterminálního symbolu *simple-expression* v gramatice 5.1.2 pomocí pravidla 16. Návratovou hodnotou je ukazatel na strukturu popisující jednoduchý výraz bez relačních a logických operátorů.
- ***Expr * simple_expression2(Expr * ex)*** realizuje expanzi neterminálu *simple-expression2* v gramatice 5.1.2 pomocí pravidel 17 a 18. Vstupní parametr *ex* slouží jako syntetizovaný atribut předávaný v pravidlech 16 a 17, aby mohla být vytvořena struktura operátoru s dvěma operandy.
- ***Expr * term(void)*** realizuje expanzi neterminálu *term* v gramatice 5.1.2 pomocí pravidla 19. Návratovou hodnotou je ukazatel na část výrazu načteného pomocí daného pravidla.
- ***Expr * term2(Expr * ex)*** realizuje expanzi neterminálu *term2* v gramatice 5.1.2 pomocí pravidel 20 a 21. Vstupní parametr *ex* slouží jako syntetizovaný atribut předávaný v pravidlech 19 a 20. Ten je nezbytný pro vytvoření uzlu operátoru se dvěma operandy.
- ***Expr * factor(void)*** realizuje expanzi neterminálu *factor* v gramatice 5.1.2 pomocí pravidel 22, 23, 24, 25, 26, 27, 28, 29. Návratovou hodnotou je ukazatel na část výrazu načteného pomocí konkrétního pravidla.
- ***Expr * factor2(Expr * ex)*** realizuje expanzi neterminálu *factor2* v gramatice 5.1.2 pomocí pravidel 30 a 31. Vstupní parametr *ex* slouží jako syntetizovaný atribut předávaný v pravidle 28, který je zásadní pro vytvoření výrazu se dvěma operandy.
- ***Operators addop(void)*** realizuje expanzi neterminálu *addop* v gramatice 5.1.2 pomocí pravidel 32 a 33. Výstupem je typ načteného aditivního operátoru.
- ***Operators mulop(void)*** realizuje expanzi neterminálu *mulop* v gramatice 5.1.2 pomocí pravidel 34, 35 a 36. Výstupem je typ načteného multiplikativního operátoru.

- ***Operators sign(void)*** realizuje expanzi neterminálu *sign* v gramatice 5.1.2 pomocí pravidel 37, 38 a 39. Výstupem je načtené znaménko ve formě typu operátoru.

6.2 Popis ovládání programu

Na obrázku 6.1 je vyobrazen spuštěný program na přepisování číselných řetězců. Tento program funguje jako konzolová aplikace a uživateli umožňuje ovládání skrze základní příkazy, které jsou zobrazeny v menu spuštěného programu.

V záhlaví programu lze vidět počet načtených přepisovacích pravidel, maximální jednotkovou výšku výstupního řetězce a časový limit na přepis jednoho vstupního řetězce.

Uživatel může zvolit načtení série pravidel z textového souboru. V takovém případě je vyzván k zadání názvu souboru s pravidly. Další možností uživatele je zadat pravidla ručně. Načítané pravidlo (pravidla) musí splňovat syntaxi popsanou gramatikou 5.1. Správně načtená pravidla lze zkontrolovat pomocí další volby v menu. Také je možné všechna pravidla smazat z paměti programu. V hlavním menu je dále dostupné nastavení maximální jednotkové výšky a maximální doby běhu výpočtu. Poslední možností je samotný výpočet. Vstupní řetězec je možné načíst ze standardního vstupu (klávesnice), v takovém případě se zobrazí výstup do konzole. Druhou variantou je načtení vstupu ze souboru, v takovém případě je výstup zapsán do souboru, jehož název je požadován po uživateli před zahájením výpočtu.

```
String number rewriter v1.10.1
This program rewrites string using given rules.
Active rules: 0
Maximal height: 2
Time limit to rewrite a string(s): 10
Press 1 to read rules from file
Press 2 to add rule from keyboard
Press 3 print rules
Press 4 to delete rules
Press 5 to set maximal height
Press 6 to set time limit
Press 7 to read numbers from keyboard
Press 8 to read numbers from file
Press 9 to exit
```

Obrázek 6.1: Okno se spuštěným přepisovacím programem

6.3 Asymptotický odhad časové složitosti

Implementovaný program se skládá z několika částí. První částí je lexikální analyzátor zadaných přepisovacích pravidel. Druhou částí je syntaktická analýza, během které je souběžně vytvářen derivační strom načítaných pravidel. Poslední částí je samotné přepisování číselných řetězců, které využívá uživatelem zadaná pravidla. Každá z těchto částí probíhá samostatně, a proto je lze z hlediska asymptotického časového odhadu považovat za nezávislé.

Lexikální analýza načítá posloupnost elementů znakové sady a pomocí konečného automatu je převádí na posloupnost lexikálních symbolů. Poznamenejme, že lexikálních symbolů je nejvýše stejný počet jako vstupních znaků. Jelikož konečný automat pracuje v lineárním čase vzhledem k délce vstupního řetězce, asymptotická časová složitost lexikální analýzy je $\mathcal{O}(n)$, kde n je počet znaků v načítaném přepisovacím pravidle

Během syntaktické analýzy dochází k načtení lexikálních symbolů a vytvoření odpovídajícího derivačního stromu. Při načtení každého lexikálního symbolu je vždy jasné, které pravidlo gramatiky použít, a proto nedochází k návratům. Poznamenejme však, že pro jeden lexikální symbol se může postupně volat více funkcí, než dojde k rozpoznání požadovaného vzoru. Maximální počet volání je však pro danou gramatiku konstantní. Dále zvolená gramatika splňuje vlastnosti L-atributové gramatiky, což umožňuje výpočet všech atributů během jediného průchodu. Tím je zaručena lineární časová složitost $\mathcal{O}(l)$ vzhledem k délce vstupní posloupnosti lexikálních symbolů l.

Samotné přepisování probíhá tak, že se na každé pozici vstupního řetězce program rozhoduje, zda nějaké pravidlo aplikuje. Rozhodování probíhá pomocí vyhodnocování derivačního stromu vytvořeného během syntaktické analýzy. Vyhodnocení tohoto stromu je prakticky průchod tímto stromem, kdy se každý uzel projde právě jednou, a proto má časovou složitost $\mathcal{O}(l)$, kde l je počet uzlů ve stromě (což koresponduje s počtem lexikálních symbolů v pravidle). Pro vstupní řetězec délky s a p pravidel má tak jeden průchod časovou složitost $\mathcal{O}(spl)$. Z povahy programu však nejde říci, po kolika průchodem dojde k ukončení přepisování. Vše totiž záleží na vstupní abecedě, výstupní abecedě a zvolených přepisovacích pravidlech. V případě nevhodně zvolených pravidel dokonce nemusí existovat cílový přepis. Nelze brát v potaz ani jednotkovou výšku vstupního řetězce, neboť aplikace nějakého pravidla může snížením jedné cifry zvýšit jinou a nedojde tak ke snížení jednotkové výšky. Uvažujme tak nějakou blíže nespecifikovanou metriku ω, která určuje nezbytný počet průchodů k přepsání vstupního řetězce na cílovou abecedu pomocí zvolených pravidel. Výsledná složitost přepisování je tedy $\mathcal{O}(spl\omega)$.

Celková asymptotická časová složitost programu je pak $\mathcal{O}(np+lp+spl\omega) \sim \mathcal{O}(spl\omega)$.

Závěr

Cílem této práce bylo shrnout teorii a známé výsledky na téma DUG vlastnosti algebraických číselných těles. Dalším cílem bylo na základě těchto znalostí rozšířit program pro přepisování číselných řetězců implementovaný v bakalářské práci [15] o možnost parametrizace vstupu načtením sady přepisovacích pravidel. K této implementaci bylo nutné shrnout teorii z oblasti formálních jazyků a překladů. Posledním cílem bylo analyzovat časovou složitost implementovaného programu ve smyslu asymptotického odhadu.

Důležitou částí práce bylo provedení rešerše napříč různými matematickými a informatickými disciplínami od teorie čísel, přes reprezentace čísel v bázi, kombinatoriku na slovech, formální jazyky, gramatiky a automaty, až po překladače a interprety. To vše bylo uvedeno do souvislostí s implementovaným programem.

Výstupem práce je rozšíření existujícího programu o interpret jazyka přepisovacích pravidel detailně navržený v Kapitole 5. Interpret je implementován pomocí metody rekurzivního sestupu a celý včetně lexikální analýzy je naprogramován v jazyce C++. Hlavní výhodou tohoto interpretačního překladače je, že je jednoprůchodový. Výsledný program nyní umožňuje uživateli načíst pravidla, podle kterých dojde k přepsání vstupních číselných řetězců. Oproti původní verzi je také možné počítat s komplexními celými čísly namísto celých čísel. Nyní tak implementovaný program může být nápomocný při dokazování DUG vlastnosti algebraických číselných těles, u kterých tato vlastnost zatím dokázána nebyla.

Na konci práce je pokus o asymptotický odhad časové složitosti implementovaného programu, který je však z povahy programu pouze symbolický.

Literatura

[1] Masáková, Z.; Pelantová, E.: *Teorie čísel*. Česká technika - nakladatelství ČVUT, 2010.

[2] Melichar, B.: *Jazyky a překlady*. České vysoké učení technické v Praze, 2003.

[3] Lothaire, M.: *Combinatorics on words*, ročník 17. Addison-Wesley Publishing Co., 1983.

[4] Lothaire, M.: *Algebraic combinatorics on words*, ročník 90. Addison-Wesley Publishing Co., 2002.

[5] Chomsky, N.: Three models for the description of language. *IRE Transactions on Information Theory*, ročník 2, č. 3, September 1956: s. 113–124, ISSN 0096-1000, doi:10.1109/TIT.1956.1056813.

[6] Rényi, A.: Representations for real numbers and their ergodic properties. *Acta Mathematica Academiae Scientiarum Hungarica*, ročník 8, č. 3-4, 1957: s. 477–493.

[7] Thurston, W. P.: Groups, tilings and finite state automata. 1989.

[8] Jacobson, B.: Sums of Distinct Divisors and Sums of Distinct Units. *Proceedings of the American Mathematical Society*, 1964: s. 179–183.

[9] Śliwa, J.: Sums of distinct units. *Bulletin de L'Academie Polonaise des Sciences*, 1974: s. 11–13.

[10] Belcher, P.: Integers Expressible as Sums of Distinct Units. *Bulletin of the London Mathematical Society*, 1974: s. 66–68.

[11] Belcher, P.: A test for integers being sums of distinct units applied to cubic fields. *Journal of the London Mathematical Society*, 1976: s. 141–148.

[12] Thuswaldner, J.; Ziegler, V.: On linear combinations of units with bounded coefficients. *Mathematika*, 2011: s. 247–262.

[13] Hajdu, L.; Ziegler, V.: On linear combinations of units with bounded coefficients. *Mathematics of Computation*, 2014: s. 1495–1512.

[14] Dombek, D.; Masáková, Z.; Ziegler, V.: On distinct unit generated fields that are totally complex. *ArXiv e-prints*, 2014, 1403.0775. Dostupné z: http://arxiv.org/abs/1403.0775

[15] Oppl, D.: *Přepisování číselných řetězců a jeho aplikace v teorii čísel*. Bakalářská práce, České vysoké učení technické v Praze, 2015.

[16] Dombek, D.: *Non-standard representations of numbers*. Disertační práce, České vysoké učení technické v Praze, 2014.

[17] Šárka Vavrečková: *Programování překladačů*. Slezská univerzita v Opavě, 2008.

[18] Češka, M.; Hruška, T.; Beneš, M.: *Překladače*. Vysoké učení technické v Brně, 1993. Dostupné z: http://www.fit.vutbr.cz/~meduna/fjp/skripta.pdf

Příloha **A**

Obsah přiloženého CD

```
readme.txt ..... stručný popis obsahu CD
exe ..... adresář se spustitelnou formou implementace a příklady
src ..... zdrojové kódy implementace
text ..... text práce
    thesis ..... zdrojová forma práce ve formátu LaTeX
    thesis.pdf ..... text práce ve formátu PDF
```