VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

LEXIKÁLNÍ ANALYZÁTOR PRO VÍCEPROCESOROVÉ POČÍTAČE

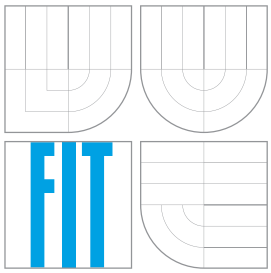
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JIŘÍ OTÁHAL

BRNO 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

LEXIKÁLNÍ ANALYZÁTOR PRO VÍCEPROCESOROVÉ POČÍTAČE

LEXICAL ANALYZER FOR MULTIPROCESOR COMPUTERS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JIŘÍ OTÁHAL
AUTHOR

VEDOUCÍ PRÁCE — Ing. MARTIN ČERMÁK
SUPERVISOR

BRNO 2010

Abstrakt

Cílem práce je vymyslet metodu, která urychlí analýzu zdrojových textů na víceprocesorových počítačích. Pro tento účel aplikace využívá spuštění více procesů pod systémem UNIX. Každý takto vytvořený proces analyzuje předem určený blok ve zdrojovém souboru a poté se ukončí. Výstupem těchto procesů jsou vnitřní struktury, které reprezentují právě daný blok. Ze struktur je již sekvenčně vytvořen mezikód, který se následně interpretuje. Takto provedená paralelní analýza vedla ke zrychlení oproti klasické sekvenční.

Abstract

Aim of this thesis is to invent method, which should accelerate speed of the analysis of source texts with multiprocessor computers. For this purpose application runs multiple process in Unix system. Each undergoing process analyzes exact partition in source file and then closes itself. Outcome of this process are internal structures, which presents exact partition. Inter-code is sequentially built from the structures which are subsequently interpreted. This kind of parallel analysis achieves acceleration of speed on the contrary of typical sequential analysis.

Klíčová slova

interpret, překladač, lexikální analyzátor, syntaktický analyzátor, paralelní programování, bezkontextová gramatika, konečné automaty

Keywords

interpreter, compiler, lexical analyzer, parser, parallel programming, context-free grammar, finite state machine

Citace

Jiří Otáhal: Lexikální analyzátor pro víceprocesorové počítače, bakalářská práce, Brno, FIT VUT v Brně, 2010

Lexikální analyzátor pro víceprocesorové počítače

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Martina Čermáka.

. . . . . .
Jiří Otáhal
18. května 2010

Poděkování

Rád bych poděkoval panu Ing. Martinu Čermákovi za odborné připomínky vedoucí k celkovému zkvalitnění jak tohoto textu, tak aplikace vyvíjené v tomto projektu. Dále děkuji všem co mi pomohli se slohovou a typografickou stránkou tohoto textu.

Obsah

Kapitola 1

Úvod

Tento dokument se zabývá návrhem paralelních metod lexikální a syntaktické analýzy. Pro návrh jsem využil znalostí z předmětů „Formální jazyky a překladače“ a „Operační systémy“. Jako prezentaci funkčnosti a využitelnosti v praxi bude sestrojen interpret, který vybranou metodu využívá.

Zpráva je rozčleněna do několika kapitol. Kapitola 2 je věnována definování základních pojmů, bez kterých by tento projekt nebylo možné pochopit. Dále pak podrobněji popisuje fáze samotného překladače, které je nutné implementovat pro správnou funkci aplikace.

Důležitou část představuje kapitola 3. Stručně nás seznámí s paralelním programováním. Dozvíme se jak mezi sebou procesy komunikují a také jak se synchronizují. V neposlední řadě si povíme, jaké prostředky můžeme pro rozdělení procesu využít.

Následuje kapitola 4 – Analýza problémů a návrh jejich řešení. Jak již název napovídá, v této kapitole si povíme o možných komplikacích, které nás mohou potkat při implementaci tohoto projektu. Zároveň si předvedeme i jejich možná řešení. Navíc kapitola obsahuje i návrh metod pro průchod zdrojového souboru.

Jedním z bodů požadovaného v zadání projektu je navržení vlastního jednoduchého jazyka. Syntaxi, sémantiku a popis lexémů tohoto jazyka najdeme v kapitole 5.

Poslední dvě kapitoly, nepočítáme-li závěr, se věnují samotné aplikaci vyvíjené v tomto projektu. Zatímco část 7 obsahuje konkrétní testy aplikace, tak kapitola 6 pojednává o samotné implementaci aplikace. Zde se dovíme jaké metody a prostředky program používá, jak mezi sebou komunikují jednotlivé procesy, nebo třeba jak je implementovaná tabulka symbolů.

Dosažené výsledky a možné vylepšení aplikace obsahuje závěrečná 8. kapitola.

Kapitola 2

Teorie překladačů

Abychom pochopili činnost aplikace vyvíjené v tomto projektu, musíme si zavést základní definice prostředků, které budeme pro implementaci využívat. Dále si v této kapitole více přiblížíme problematiku překladu ze zdrojového jazyka do cílového jazyka. Všechny pojmy a definice zmíněné v této kapitole jsou převzaty z [1].

2.1 Základní pojmy

Zde se seznámíme se základními prostředky používanými ve spojení s překladači.

2.1.1 Abeceda a symboly

Příkladem může být binární abeceda, kterou představuje abeceda $\{0, 1\}$. Symboly zde jsou 0 a 1. Definice abecedy zní takto.

Definice 2.1.1. Abeceda je konečná, neprázdná množina elementů, které nazýváme symboly.

2.1.2 Řetězec

V literatuře se někdy můžeme setkat s označením řetězce jako řetěz nebo také slovo. Než si uvedeme operace, které s řetězci můžeme provádět, musíme si řetězec nadefinovat. Prázdný řetězec budeme označovat jako ε.

Definice 2.1.2. Nechť Σ je abeceda.

- ε je řetězec nad abecedou Σ
- pokud x je řetězec nad Σ a $a \in \Sigma$, potom xa je řetězec nad abecedou Σ

Délka řetězce

Definice 2.1.3. Nechť x je řetězec nad abecedou Σ. Délka řetězce x ($\mid x \mid$), je definována:

- pokud $x = \epsilon$, pak $\mid x \mid = 0$
- pokud $x = a_1 \ldots a_n$, pak $\mid x \mid = n$ pro $n \geq 1$ a $a_i \in \Sigma$ pro všecha $i = 1, \ldots, n$

Konkatenace řetězců

Definice 2.1.4. Nechť x a y jsou dva řetězce nad abecedou Σ. Konkatenace x a y je řetězec xy.

Mocnina řetězce

Definice 2.1.5. Nechť x je řetězec nad abecedou Σ. Pro $i \geq 0$, i-tá mocnina řetězce x, x^i, je definována: $x^0 = \varepsilon$ a pro $i \geq 1 : x^i = xx^{i-1}$

Reverzace řetězce

Definice 2.1.6. Nechť x je řetězec nad abecedou Σ. Reverzace řetězce x reversal(x), je definována:

- pokud $x = \epsilon$ pak reversal(ϵ) $= \epsilon$
- pokud $x = a_1 \dots a_n$, pak reversal($a_1 \dots a_n$) $= a_n \dots a_1$ pro $n \geq 1$ a $a_i \in \Sigma$ pro všechna $i = 1, \dots, n$

Prefix řetězce

Definice 2.1.7. Nechť x a y jsou dva řetězce nad abecedou Σ. Potom x je prefixem y, pokud existuje řetězec z nad abecedou Σ, přičemž platí $xz = y$.

Sufix řetězce

Definice 2.1.8. Nechť x a y jsou dva řetězce nad abecedou Σ. Potom x je sufixem y, pokud existuje řetězec z nad abecedou Σ, přičemž platí $zx = y$.

Podřetězec

Definice 2.1.9. Nechť x a y jsou dva řetězce nad abecedou Σ. Potom x je podřetězcem y, pokud existuje řetězec z, z' nad abecedou Σ, přičemž platí $zxz' = y$.

2.1.3 Jazyk

Jazyk může být konečný nebo nekonečný. Jazyk *L* je konečný, pokud *L* obsahuje konečný počet řetězců, jinak je nekonečný. Nad jazyky můžeme provádět hned několik operací. Než si je popíšeme, uvedu definici samotného jazyka.

Definice 2.1.10. Nechť Σ^* značí množinu všech řetězců nad Σ. Každá podmnožina $L \subseteq \Sigma^*$ je jazyk nad Σ.

Sjednocení

Slovně můžeme sjednocení popsat jako všechny řetězce obsažené ve sjednocovaných jazycích.

Definice 2.1.11. Nechť L_1 a L_2 jsou dva jazyky nad Σ. Sjednocení jazyků L_1 a L_2, $L_1 \cup L_2$ je definováno: $L_1 \cup L_2 = \{x : x \in L_1 \text{ nebo } x \in L_2\}$

Průnik

Každý řetězec, který vyhovuje průniku, musí být obsažen ve všech jazycích, nad kterými operaci provádíme.

Definice 2.1.12. Nechť L_1 a L_2 jsou dva jazyky nad Σ. Průnik jazyků L_1 a L_2, $L_1 \cap L_2$ je definováno: $L_1 \cap L_2 = \{x : x \in L_1 \text{ a } x \in L_2\}$

Rozdíl

Výsledkem jsou jen ty řetězce, které jsou obsaženy v prvním jazyce a zároveň nejsou obsaženy v jazyce druhém.

Definice 2.1.13. Nechť L_1 a L_2 jsou dva jazyky nad Σ. Rozdíl jazyků L_1 a L_2, $L_1 - L_2$ je definováno: $L_1 - L_2 = \{x : x \in L_1 \text{ a } x \notin L_2\}$

Doplněk

Doplňku vyhovují ty řetězce, které se nenachází v daném jazyce, ale je možné je pomocí abecedy sestrojit.

Definice 2.1.14. Nechť L_1 a L_2 jsou dva jazyky nad Σ. Doplněk jazyků L, $\overline{L}$, je definován: $\overline{L} = \Sigma^* - L$

Konkatenace

Konkatenace je spojení řetězců daných jazyků - každý s každým.

Definice 2.1.15. Nechť L_1 a L_2 jsou dva jazyky nad Σ. Konkatenace jazyků L_1 a L_2, L_1L_2, je definován: $L_1L_2 = \{xy : x \in L_1 \text{ a } y \in L_2\}$

Reverzace

Reverzace je otočení všech řetězců patřící do daného jazyka.

Definice 2.1.16. Nechť L je jazyk nad Σ. Reverzace jazyka L reverse(L), je definována: reverse(L) = {reverse(x):$x \in L$}

Mocnina

Definice 2.1.17. Nechť L_1 je jazyk nad Σ. Pro $i \geq 0$, i-tá mocnina jazyka L, L^i, je definována: $L^0 = \{\varepsilon\}$ a pro $i \geq 1 : L^i = LL^{i-1}$

2.1.4 Reprezentace jazyků

Konečné jazyky můžeme jednoduše specifikovat výčtem jejich slov. Nekonečné jazyky (např. programovací jazyky) takto specifikovat nemůžeme. Pro jejich popis si zavedeme následující gramatiky a automaty. Tyto prostředky totiž představují konečnou reprezentaci nekonečných, ale i konečných jazyků.

Každá gramatika obsahuje konečnou množinu pravidel, pomocí kterých generujeme daný jazyk. Stejně tak i automaty definují jazyk pomocí konečných prostředků, na jejichž základě algoritmus rozhodne, zda daný řetězec patří do jazyka či nikoliv.

Regulární výrazy

Důležitou notací pro specifikaci vzorů symbolů jsou regulární výrazy. Slouží pro pojmenování množiny řetězců. Jinými slovy zadává lexikálním symbolům jejich podobu (např. identifikátor nesmí začínat číslem, může obsahovat podtržítko, apod.).

Definice 2.1.18. Nechť Σ je abeceda. **Regulární výraz** nad abecedou Σ a **jazyky**, které **značí**, jsou definovány následovně:

- $\varnothing$ je RV značící prázdnou množinu (prázdný jazyk)
- ε je RV značící jazyk ε
- a, kde $a \in \Sigma$, je RV značící jazyk a
- Nechť r a s jsou regulární výrazy značící po řadě jazyky L_r a L_s, potom:
  - $(r.s)$ je RV značící jazyk $L = L_r L_s$
  - $(r + s)$ je RV značící jazyk $L = L_r \cup L_s$
  - (r^*) je RV značící jazyk $L = {L_r}^*$

Konečný automat

Pro každý symbol jazyka můžeme sestrojit konečný automat. Všechny tyto automaty pak pomocí ε-přechodů můžeme spojit v jediný automat, který bude reprezentovat požadovaný jazyk. Konečný automat představuje prostředek, pomocí kterého je realizována lexikální analýza.

Definice 2.1.19. Konečný automat (KA) je pětice: $M = (Q, \Sigma, R, s, F)$, kde

- Q je konečná množina stavů
- Σ je vstupní abeceda
- R je konečná množina pravidel ve tvaru: $pa \to q$ kde $p, q \in Q$, $a \in \Sigma \cup \{\epsilon\}$
- $s \in Q$ je počáteční stav
- $F \subseteq Q$ je množina koncových stavů

Bezkontextová gramatika

Bezkontextovou gramatikou (2.1.20) se nejčastěji popisuje syntaxe programovacích jazyků. Pro názorné zobrazení derivace (struktury věty) v bezkontextové gramatice slouží *derivační strom* (2.1.21). Nejlepší ukázkou zřejmě bude příklad takového stromu. Nejdříve si však uvedeme formální definice bezkontextové gramatiky a derivačního stromu.

Definice 2.1.20. Bezkontextová gramatika je čtveřice: $G = (N, T, P, S)$, kde

- N je abeceda neterminálů
- T je abeceda terminálů, přičemž $N \cap T = \varnothing$
- P je konečná množina pravidel ve tvaru: $A \to x$, kde $A \in N, x \in (N \cup T)^*$

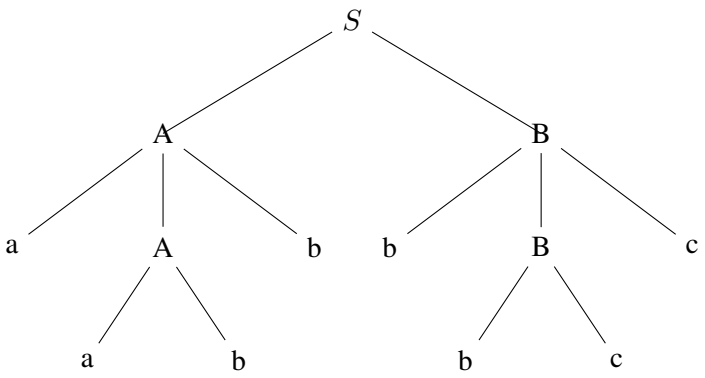
- $S \in N$ je počáteční neterminál

Definice 2.1.21. Nechť $G = (N, T, P, S)$ je bezkontextová gramatika. Strom je derivační strom v G, jestliže.

- Každý uzel je ohodnocen symbolem z $N \cup T$.
- Kořen je ohodnocen symbolem S.
- Jestliže uzel má nejméně jednoho následovníka, pak je ohodnocen symbolem z N.
- Jestliže $b_1, b_2, \ldots, b_k$ jsou přímí následovníci uzlu a, jenž je ohodnocen symbolem A, v pořadí zleva doprava s ohodnocením $B_1, B_2, \ldots, B_k$, pak $A \to B_1 B_2 \ldots B_k \in P$.

Příklad 2.1.22. Mějme gramatiku $G = (\{S, A, B\}, \{a, b, b\}, P, S)$, kde P obsahuje pravidla $S \to AB$, $A \to aAb \mid ab$, $B \to bBc \mid bc$. Pak v této gramatice můžeme generovat řetězec $aabbbbcc$ například takto:

$$S \Rightarrow AB \Rightarrow aAbB \Rightarrow aAbbBc \Rightarrow aAbbbcc \Rightarrow aabbbbcc$$



Obrázek 2.1: Derivační strom pro příklad 2.1.22

2.1.5 Tabulka symbolů

V této tabulce se uchovávají informace o objektech deklarovaných explicitně (uživatelské typy, proměnné, procedury, apod.) nebo implicitně (standardní typy, pomocné typy vytvořené překladačem, apod.).

Tabulka nachází využití při provádění typové kontroly (sémantická analýza), řešení kontextových vazeb (vztahy mezi deklarací a použitím objektu) a generování intermediárního a cílového kódu. Tabulku může vytvářet buď lexikální nebo sémantická analýza. V prvním případě by se tabulka vytvářela již v první fázi překladu. V dalších částech překladu by jména objektů byla reprezentována už jen jako ukazatelé do tabulky. Pokud by tabulku vytvářela až sémantická analýza, tak by lexikální analyzátor vracel textové řetězce reprezentující jména objektů místo ukazatelů do tabulky.

V praxi se tabulka symbolů implementuje jako binární vyhledávací strom nebo hash tabulkou. Hlavním kritériem pro tabulku symbolů je rychlost vyhledávání. Jak binární vyhledávací strom, tak hash tabulka, toto kritérium splňuje.

2.2 Fáze překladu

Samotná kompilace je rozčleněna do několika fází. Těmito částmi musí zdrojový program zapsaný ve zdrojovém jazyce projít, aby se z něj stal cílový program v cílovém jazyce.

2.2.1 Lexikální analyzátor

První fází překladače je lexikální analyzátor (lexical analysis, scanning). Vstupem lexikální analýzy je soubor se zdrojovým kódem a výstupem jsou tokeny. Funkcí lexikální analýzy je číst znaky ze vstupního souboru a tvořit z nich lexikální symboly. Každý takový symbol představuje logickou posloupnost znaků a nazývá se lexém. Výstupem lexikální analýzy má být však token. Ten je složen z daného lexému a jeho hodnoty. Token je tedy dvojice lexém:hodnota (např. int:20). Pokud daný lexém nemá hodnotu, můžeme jí vynechat (např. rovnítko). Z následující posloupnosti znaků budou vytvořeny tyto tokeny:

$$promenna = 20 * 4.00;$$

Pořadí	Typ	Hodnota
1	identifikátor	promenna
2	rovnítko	
3	int	20
4	operátor	*
5	double	4.00
6	středník	

Tabulka 2.1: Fronta lexikálních symbolů

V tabulce 2.1 můžeme vidět výstup lexikální analýzy daného výrazu. Každý řádek představuje jeden token (pro lepší přehlednost je v tabulce navíc sloupeček pořadí). Syntaktický analyzátor (viz. 2.2.2) by požadoval tokeny postupně v pořadí od 1 (v našem příkladu tedy od identifikátoru).

V praxi se pro implementaci lexikální analýzy využívá konečný automat 2.1.4. Symboly nepotřebné pro syntaktickou analýzu („bílé znaky“, komentáře, ...) jsou konečným automatem vynechány a lexikální analyzátor je tedy na svém výstupu vůbec neuvádí.

2.2.2 Syntaktický analyzátor

Výstupem syntaktické analýzy je derivační strom nebo určitá posloupnost akcí, které uchovávají vnitřní reprezentaci struktury zdrojového textu a sémantiky těchto struktur. Pro vytvoření takové struktury (příp. derivačního stromu) syntaktický analyzátor potřebuje tokeny. Ty dostává od lexikálního analyzátoru popsaného v předchozí sekci. Během syntaktické analýzy překladač kontroluje správné pořadí lexémů. Pokud zdrojový text obsahuje chyby, může se využít techniky pro zotavení se z chyb a analýza i po chybě může pokračovat dále. Pokud se této techniky nevyužije, tak analyzátor skončí s kontrolou hned na první chybě. Vhodné jsou i detailnější výpisy o nalezené chybě, protože v praxi nám většinou nestačí pouze informace o správném či špatném zápisu programu – chceme například vědět i pozici chyby v souboru.

Pro implementaci této fáze jsou nejčastěji používané principy shora dolů nebo zdola nahoru. Tyto názvy odpovídají postupu při vytváření derivačního stromu. Více se o těchto principech a obecně o syntaktické analýze můžeme dozvědět v [1].

2.2.3 Sémantický analyzátor

Vstupem tomuto analyzátoru je právě vnitřní reprezentace struktury zdrojového kódu vytvořená syntaktickým analyzátorem. Sémantický analyzátor provádí typovou kontrolu výrazu. Zjišťuje, zda všechny použité operátory ve výrazu mají operandem povolené specifikace. Například kontroluje, zda řetězcovou konstantu přiřazuje do proměnné, jejíž typ může řetězcovou konstantu uchovávat. Některé překladače však dovolují implicitní typovou konverzi. Sémantický analyzátor tedy provádí kontrolu, zda gramaticky správné fráze neporušují kontextová omezení (např., použité proměnné musí být deklarované).

2.2.4 Generování mezikódu

Po lexikální, syntaktické a sémantické kontrole následuje generování intermediární reprezentace zdrojového programu (mezikód). V praxi většinou nejde o samostatnou část překladu. Generování vnitřní formy programu se přímo spojuje se syntaktickou analýzou. V našem případě však půjde o samostatnou část, protože paralelní kontrola nám nedovolí generovat přímou cestou tento kód. O tom si povíme v následujících kapitolách tohoto textu.

Intermediární kód slouží jako podklad pro optimalizaci a generování cílového kódu. Konkrétně pro nás ale znamená konečný produkt překladače a interpret mezikód přímo provádí. Zde se interpret stará i o chyby, které jsou způsobené vnějšími okolnostmi (např. zaplnění disku při zápisu do souboru).

Kapitola 3

Paralelní programování

Při psaní této kapitoly jsem využil znalostí z předmětu „Operační systémy“ (viz. [2]). Zde se budeme zabývat jedním z důležitých prostředků našeho projektu. Hlavním cílem paralelního programování je zkrácení času, potřebného pro zpracování nějaké úlohy. (V našem případě budeme chtít docílit rychlejší lexikální a syntaktické analýzy díky využití této techniky.)

Zrychlení docílíme zaměstnáním všech dostupných procesorů v počítači. K tomu můžeme využít spuštění více procesů nebo vláken. V následujícím odstavci si ujasníme rozdíl mezi procesy a vlákny.

3.1 Rozdělení procesu

Na výběr máme tedy vlákna nebo procesy. Vlákna vznikají „rozštěpením“ hlavního procesu a všechna spolu sdílejí jednu paměť. Mohou tak pomocí ní jednoduše komunikovat. Další vlastností vláken je výrazně vyšší rychlost oproti procesům. Tím není myšlena rychlost výpočetní, ale rychlost při vytváření a rušení vláken (nemusí se totiž kopírovat paměťový prostor). V neposlední řadě tak ušetříme i paměť. Naproti tomu proces má přidělenou svou vlastní paměť, která je chráněna před přístupem od ostatních procesů. Procesy spolu tedy nemohou komunikovat přímo.

Pro rozdělení jednoho programu na více procesů slouží na systémech UNIX systémové volání `fork()` (pro operační systém Windows bychom mohli využít funkce `CreateProcess()`). Po tomto volání se vytvoří kompletní a přesná kopie procesu, ze kterého byl `fork()` volán (včetně všech proměnných a dat v okamžiku volání). Rozlišit mezi vzniklými procesy můžeme díky návratové hodnotě. Pro potomka vrací `fork()` nulu a pro rodičovský proces vrací `fork()` pid (číslo procesu) potomka. Pokud je volání `fork()` neúspěšné, návratová hodnota se rovná mínus jedné. Následující kód demonstruje rozdělení procesů po volání `fork()`.

```
switch(fork()){
    case 0:  // bude provedeno potomkem
       exit(); // možné ukončení
    break;
    case -1: // chyba při volání fork()
    break;
    default: // bude provedeno rodičem
        wait(); // čekání na dokončení potomka
    break;
}
```

Pro ukončení potomka je možné využít volání `exit`. Důležité může být i volání `wait`, které pozastavuje proces rodiče, dokud neskončí proces potomka. Pokud potomek dokončil svou činnost dříve než rodič volá `wait()`, tak se rodičovský proces samozřejmě uspávat nebude.

Pro vytváření multiprocesových aplikací existují dva požadavky:

1. Synchronizace procesů
2. Komunikace mezi procesy

Například v situaci, kdy jeden proces ošetřuje vstupy dat a druhý nad nimi provádí nějaké výpočty, musíme řešit jak synchronizaci procesů, tak jejich vzájemnou datovou komunikaci.

3.2 Synchronizace procesů

Pokud by několik procesů přistupovalo ke sdíleným zdrojům současně, mohlo by to vést ke špatným výpočtům. Chyby objevující se v závislosti na předchozí větě se velice špatně odhalují, protože nastávají prakticky náhodně. Proto je velice důležité využívat mechanismy pro synchronizaci procesů.

V souvislosti se synchronizací procesů můžeme narazit na pojem **kritická sekce**. Tímto rozumíme úsek řídícího programu, kde dochází k manipulaci se sdílenými daty (respektive k jejich změně). Je důležité zajistit, aby při „spuštění“ kritické sekce jedním procesem nemohl kritickou sekci spustit jakýkoliv jiný proces. Zároveň je však potřeba vyhnout se uváznutí, blokování a stárnutí.

Uváznutí Situace, kdy akce prvního procesu může být úspěšně dokončena až po ukončení akce druhého procesu. Ta naopak může být ukončena až po úspěšném dokončení akce prvního procesu. Vzniká tak cyklická závislost a procesy na sebe navzájem musí čekat. V reálné světě bychom toto mohli přirovnat k situaci, kdy do uličky pro jedno auto proti sobě najedou auto dvě. Řešením je například couvání. Prevencí bychom mohli rozumět povolení průjezdu pouze v jednom směru.

Blokování Proces žádající o vstup do kritické sekce musí čekat i přesto, že je tato sekce volná (např. protože si procesy chtějí navzájem vyhovět). Například když se dva lidé potkají v úzké uličce a chtějí se vyhnout, tak jeden udělá krok stranou, ale druhý se snaží vyhnout stejným směrem.

Stárnutí Tato situace nastane, když proces čeká na přidělení sdíleného zdroje, který je však stále přidělován jiným procesům (například z důvodu jejich vyšší priority). Stárnutí v reálném životě můžeme vidět opět v motorismu. V případě, kdy vozidlo musí objet překážku ve svém pruhu s využitím protisměrného pruhu, kde však stále jezdí jiná vozidla.

3.2.1 Semafory

Pro synchronizaci procesů můžeme využít semafory (viz [5]). Tyto semafory se používají podobně jako segmenty sdílené paměti. Pro alokaci a uvolnění se používají funkce `semget()` a `semctl()`. Každý semafor využívá funkce `sem_op()`, pomocí níž žádáme o přístup do kritické sekce. Semafor si můžeme představit jako čítač. Když proces zažádá o přístup do kritické sekce, sníží se hodnota semaforu. Pokud je hodnota kladná může proces vstoupit. V opačném případě se proces uspí a čeká tak dlouho, dokud se hodnota nezvýší.

3.3 Komunikace procesů

Způsobů, jak mohou procesy vzájemně komunikovat, je hned několik.

- Signály
- Roury
- Sdílená paměť

Další můžeme najít v [2]. My si povíme pouze o sdílené paměti, která v tomto projektu bude využitá.

3.3.1 Sdílená paměť

Komunikace pomocí sdílené paměti je jedním z nejjednodušších prostředků, jak mohou procesy komunikovat. Jak již název napovídá, do této paměti mohou přistupovat všechny procesy. Pokud v ní jeden proces provede změnu, tak se tato změna projeví i v ostatních procesech. Přístup do této paměti je stejně rychlý jako přístup do nesdílené paměti. Je však nutné řešit synchronizaci. O tom jak procesy synchronizovat jsme si řekli v předchozí sekci (viz. 3.2).

Alokaci sdílené paměti provádí vždy jen jeden proces pomocí funkce `shmget()`. Ostatní procesy se k takto alokované paměti připojují pomocí funkce `shmat()`. Argumenty funkce si pak volí přístup pouze pro čtení nebo i pro zápis. Pro odpojení od sdílené paměti slouží funkce `shmdt()`. Nakonec je potřeba alokovaný blok uvolnit. To zařídí opět jeden proces pomocí funkce `shmctl()`.

Kapitola 4

Analýza problémů a návrh jejich řešení

Pro paralelní analýzu bylo potřeba navrhnout takovou metodu, která zkontroluje zdrojový kód od začátku do konce pomocí více procesů. Z toho plyne několik problémů, které mohou nastat. A právě v této kapitole nastíním, jak je možné tyto komplikace vyřešit.

4.1 Rozdělení souboru

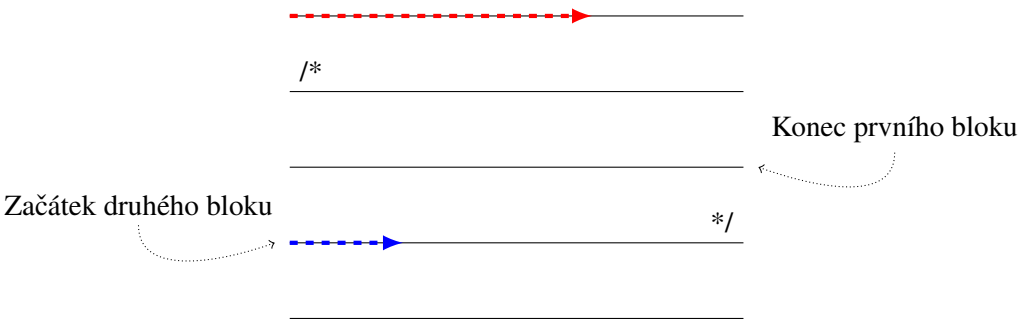
Před začátkem samotné analýzy je nutné soubor rozdělit do menších částí a každému spuštěnému procesu dát k analýze jednu nebo více částí. Při rozdělování může dojít k „přetrhnutí“ víceřádkových lexémů nebo k rozpůlení některého z bloků (`while`, `if`, apod.). Obě situace jsou nepříjemné, proto se na ně podíváme blíže.

Víceřádkový lexém

Tuto situaci můžeme vidět na obrázku 4.1. Jeden proces tak bude analyzovat počáteční část a druhý konečnou část lexému. Zatímco u prvního procesu to není až tak závažné, tak u druhého ano. Ten totiž neví, že se nachází uprostřed lexému a s analýzou bude začínat počátečním neterminálem gramatiky našeho jazyka.

Jedním ze způsobů řešení tohoto problému je nechat lexikální analyzátor vracet uvozující znaky takových lexém jako znaky speciální. Syntaktický analyzátor pak vše mezi těmito znaky jednoduše vynechá. To sebou však nese další komplikace. Zatímco u víceřádkových komentářů by nebyl problém jasně definovat, který znak je počáteční a který konečný, tak u víceřádkových řetězců už by to problém byl. Navíc pokud budeme chtít při lexikální analýze souběžně spustit i syntaktickou analýzu, nepůjde lehce přeskočit tokeny, které se nachází mezi těmito znaky. Tohle by se zřejmě dalo eliminovat zavedením zotavení se z chyb. Poté, při získaní tokenu reprezentujícího ukončovací symbol víceřádkové lexémy, by se syntaktický analyzátor jednoduše „vrátil“ na své počáteční hodnoty.

Pokud máme možnost definovat si vlastní jazyk, tak se tato situace dá řešit mnohem elegantněji. Zavedeme pravidlo, které určí, že každý takový lexém bude mít na konci řádky znak zpětného lomítka (podobně jak to známe například u řetězců z jazyka C/C++). Když tohle pravidlo zavedeme, můžeme se komplikacím vyhnout již při rozdělování souboru. Najdeme pouze znak konce řádky (`'\n'`), před kterým se nenachází zpětné lomítko a tam soubor rozdělíme.



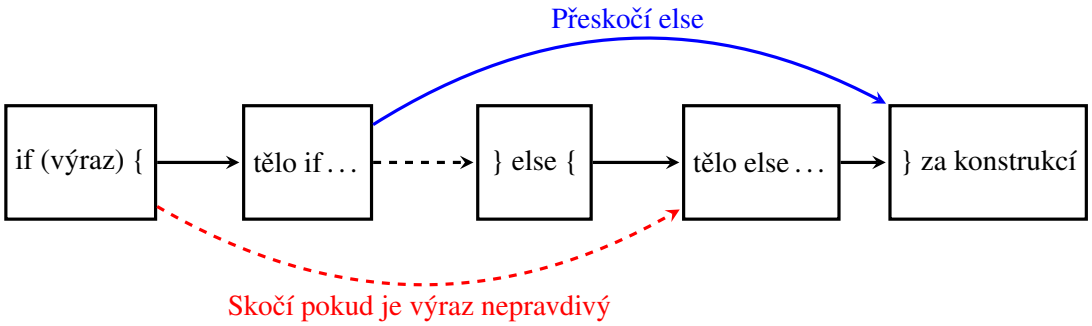
Obrázek 4.1: Problém s rozdělením víceřádkových lexém

Bloky

Kdybychom za blok považovali například funkci, tak by tato komplikace nebyla až tak závažná. Mnohem horším rozdělením je uprostřed některého z podmíněných skoků. Pro pochopení tohoto problému si nejdříve povíme, jak kompilátor zpracovává podmíněné skoky. Nejlepší ukázkou zřejmě bude konstrukce `if - else`. Zjednodušeně si popíšeme chování této konstrukce (podrobněji později v sekci 5.3):

1. zkontroluj výraz uvedený za `if`,
2. pokud je výraz pravdivý, tak proveď tělo `if`,
3. jinak proveď tělo `else`.

Seznam po sobě jdoucích operací, jaký můžeme vidět na obrázku 4.2, se tvoří během některé z analýz. Do seznamu vkládáme jednotlivé bloky, přičemž když narazíme na blok, zde označený jako `tělo else`, musíme do bloku `if` uložit informaci o jeho poloze (červená šipka). Blok `if` tak bude vědět, jaké instrukce mají následovat, pokud je výraz vyhodnocen jako nepravdivý. Pokud je výraz pravdivý, provede se tělo if. V takovém případě je nutné, aby už se neprovedla větev `else`. Proto si interpret vybere cestu přes modrou šipku, která vede až za celou prováděnou konstrukci. Jelikož analýzu provádíme zleva doprava, tak modrou a červenou šipku musíme doplnit zpětně. Dříve totiž nevíme, kde se bude nacházet jejich cíl (tělo if, stejně jako tělo else, může být libovolně dlouhé).



Obrázek 4.2: Zpracování konstrukce `if - else` interpretem

Při klasické sekvenční analýze postupujeme zleva doprava a přes celou konstrukci. Můžeme tak jednoduše zjistit jakoukoliv syntaktickou nesrovnalost. Navíc se většinou kontrola takové konstrukce implementuje v jedné funkci. Není tedy problém pamatovat si umístění již analyzovaných bloků a následně do nich zapsat cíl modré a červené šipky. Toto však neplatí při paralelní analýze. Je pravděpodobné, že si minimálně jeden takový blok ve zdrojovém kódu rozdělíme na více částí. Kontrolu nepůjde implementovat do jedné funkce, dokonce kontrola nebude provedena ani ve společném procesu.

Zřejmě jako první řešení nás napadne počítat levé a pravé složené závorky, které představují začátek a konec těla bloku. Jenomže následující kód by prošel bez větších problémů kontrolou (kdybychom ho rozdělili někde v těle if), i když by neměl:

```
if(výraz){
    //tělo if
else{
    //tělo else
}
}
```

Mnohem lepším řešením je ukládat informace o začátku a konci takových bloků a jejich syntaktickou správnost kontrolovat až při sémantické analýze (při generování mezikódu). Jednoduše tak začátek bloku vložíme na zásobník a při jeho konci ho ze zásobníku odebereme. Kontrolovat tak můžeme i zanořování bloků. Pokud přijde konec bloku v době, kdy na zásobníku žádný začátek bloku nebude, jedná se syntaktickou chybu. Nesmíme však blok rozdělit před levou složenou závorkou, to bychom potom začátky dostaly dva. Tomuto se vyhneme pokud soubor rozdělíme až za prvním nalezeným středníkem.

4.2 Paměť

Při paralelní syntaktické analýze vzniká několik syntaktických stromů. Tyto stromy je nutné po analýze spojit. Jedná-li se o opravdu velký soubor, tak jistě nepůjde o malý strom. Navíc pro paralelní programovaní platí, že procesy si navzájem do paměti „nevidí“. Je tedy potřeba vyřešit, kam vytvořené stromy ukládat.

Jedním ze způsobů jak problém řešit je využití sdílené paměti. Jako další možnost se nabízí sdílet syntaktické stromy v souborech na disku. Obě tyto možnosti si vybírají daň v podobě vyššího času běhu samotného programu.

4.3 Paralelní syntaktická analýza a interpret

Klasická syntaktická analýza je popsaná v sekci 2.2.2. Jelikož samotný interpret kódu již nelze rozdělit na několik částí, musíme sémantické akce vytvořené během syntaktické analýzy spojit a tyhle spoje také zkontrolovat. Ať už budeme implementovat kteroukoliv metodu z níže uvedených, musíme splnit požadavky, které s sebou paralelní syntaktická analýza nese.

Nutnost použití sémantického stromu

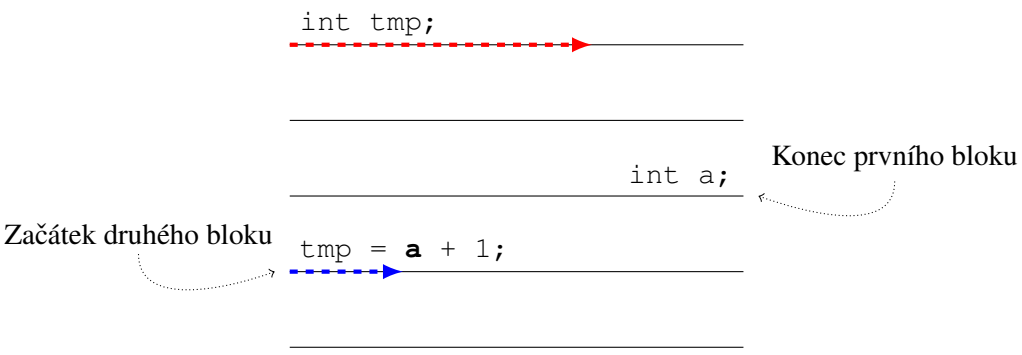
Pokud paralelní analýzu budeme využívat k interpretaci zapsaného kódu, musíme vytvářet strom sémantických akcí. Z těchto sémantických akcí je následně vytvořen intermediární kód, který interpret může přímo provádět. V tomto projektu totiž není možné, abychom symboly ukládali již

při lexikální či syntaktické analýze. Důvod je jednoduchý. Těchto analýz běží současně a separátně hned několik. V kapitole o paralelním programování (viz. 3) jsme se mohli dozvědět, že si procesy navzájem „nevidí“ do paměťového prostoru. Je však potřeba, aby tabulka symbolů byla právě jedna – kvůli sémantické analýze, která kontroluje deklaraci proměnných (viz. 2.2.3).

Tuto komplikaci by mohlo vyřešit požádání o přidělení sdílené paměti, kde by tabulka byla uložena. Do takové paměti by pak přistupovali všechny procesy, což ale není ideální. Znamenalo by to řízení přístupu do kritické sekce. A jelikož tabulka symbolů představuje objekt, ve kterém překladač stále symboly hledá, přidává či maže, tak by na sebe ostatní procesy musely neustále čekat. Dle mého názoru by to vedlo k patrnému zpomalení analýzy.

Navíc se však může stát, že na začátku analýzy posledního procesu bude použitá ve výrazu proměnná, která je deklarovaná až na konci analýzy prvního procesu. Tuto situaci nám znázorňuje obrázek 4.3. Vidíme, že první proces (červený) již uložil do tabulky symbolů deklaraci proměnné `tmp` – nikoliv však deklaraci proměnné `a`. Mezitím už druhý proces (modrý) analyzuje výraz `tmp = a + 1`. Pokud bychom generovali tabulku symbolů pouze jednu (např. při syntaktické analýze), došlo by na místě, kde se právě nachází druhý proces, k sémantické chybě.

Toto je tedy důvod, proč musíme tabulku symbolů generovat až při sémantické analýze. Do vygenerování tabulky symbolů bude proměnné reprezentovat jejich název a nikoliv adresa do této tabulky.



Obrázek 4.3: Sémantická chyba

4.4 Návrh metod pro rozdělení a průchod souboru

Zde se podíváme na možné metody pro průchod zdrojového souboru a problémy s nimi spojené. Pro lepší pochopení si budeme metody přirovnávat k nalíčení jedné zdi dvěma malíři. U této činnosti (stejně jako v následujících příkladech) je jednoduché nalézt rohy zdi (začátek a konec souboru), není však jednoduché přidělit malířům stejné úseky zdi (bloky souboru). Poznamenejme, že hledáme takovou metodu, která zaměstná malíře (procesy) stejnou prací s minimálním vynaložením sil na rozdělení práce.

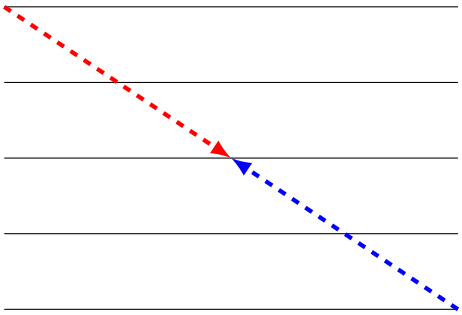
4.4.1 Metoda zpětného procesu

U této metody máme k dispozici dva procesy. Na obrázku 4.4 můžeme vidět postup obou těchto procesů. Jeden analyzuje zleva doprava od začátku souboru a druhý analyzuje zprava doleva od konce souboru. Procesy se tak „potkají“ kolem středu souboru. V naší reálné situaci by to zname-

nalo, že jeden malíř bude postupovat od horního rohu zleva a druhý od dolního rohu zprava. Pokud budou líčit stejnou rychlostí potkají se uprostřed místnosti.

Velkou výhodou této metody je, že malíři prakticky nemusí hledat začátek svého úseku zdi a mohou začít rovnou líčit. Není tedy třeba žádná režie na rozdělení souboru a navíc tato metoda minimalizuje možnost rozdělení víceřádkových lexém (viz. 4.1). Bohužel je však spojená i s několika nevýhodami.

- Pro proces, který postupuje zpětně se musí sestrojit i „zpětný konečný automat“.
- Pro otočení pořadí lexém se musí použít zásobník.
- Je komplikované spustit s každým procesem souběžně i syntaktickou analýzu. Pokud by běžela zleva doprava od začátku do konce souboru, mohlo by se stát, že by do středu dorazila rychleji než zpětné vlákno a musela by na něj čekat. Pokud by běžela zároveň i s druhým procesem musela by běžet také zpětně.
- Problém se spuštěním více jak dvou procesů k analýze. Zvýšila by se režie spojená s rozdělováním souboru.



Obrázek 4.4: Metoda zpětného procesu

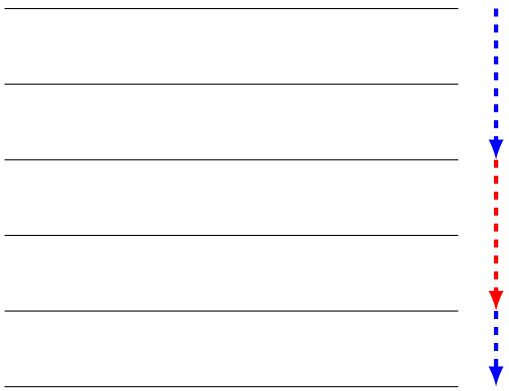
4.4.2 Metoda postupující po blocích

Tuto metodu graficky znázorňuje obrázek 4.5. Předem se stanoví velikost bloku, který bude proces analyzovat. O tuto určenou velikost se poté procesy navzájem přeskakují. Kdybychom například řekli, že velikost bloku bude právě jeden řádek v souboru, tak jeden proces bude analyzovat liché řádky a druhý bude analyzovat sudé řádky. U našich malířů nás zřejmě hned napadne zbytečné obcházení se navzájem. Jeden začne v levém horním rohu, druhý o metr dál (už zde si musí malíř vyměřit právě jeden metr plochy zdi). Až první malíř nalíčí první metr zdi, vyměří si další jeden metr a pokračuje. Analogicky se pak postupuje až do konce zdi.

Tato metoda minimalizuje možnost, že by paralelně probíhající syntaktická analýza musela čekat na tokeny (jako je tomu u předchozí metody). Všechny spuštěné procesy zde běží zleva doprava (tedy běžným způsobem). Odpadají tak některé nevýhody, které měla předchozí metoda (viz. 4.4.1). Na druhou stranu se zas objevují nové.

- Zvyšuje se režie na vyhledávaní začátků dalších bloků pro daný proces.
- Projeví se zde ve větší míře problém s rozdělováním souborů (viz. 4.1)

- Je komplikované spustit s každým procesem souběžně i syntaktickou analýzu. Stejně jako u předchozí metody by bylo lepší, kdyby syntaktická analýza byla jen jedna a běžela od začátku do konce souboru.



Obrázek 4.5: Metoda procházení po blocích

4.4.3 Metoda rovnoměrného rozložení

Při této metodě každý proces dostane k analýze blok souboru, který je předem rovnoměrně rozdělen. Počet bloků je roven počtu spuštěných procesů. Pokud si tedy pro analýzu zvolíme pouze jeden proces, vstupní soubor se dělit nebude a celá analýza proběhne běžným způsobem. Pro dva procesy se soubor rozdělí napůl, pro čtyři na čtvrtiny, atd. U našich malířů je tato metoda nejvíce přizpůsobena k rozšíření zaměstnanců (např. na tři malíře). Hned na začátku pracovního dne si rozdělí rovnoměrně práci a po jejím dokončení jejich pracovní doba končí. Zůstává tak potřeba rozdělit práci, ale mizí obcházení se navzájem.

V podstatě je to metoda velice podobná metodě zpětného vlákna (viz. 4.4.1). Rozdíl je však v tom, že zde všechny procesy postupují zleva doprava. Odpadá tak konstrukce „zpětného konečného automatu“ a potřeba využít zásobníku k seřazení tokenů. Tím, že procesy této metody po analýze své části souboru končí se nabízí využití souběžně spuštěné syntaktické analýzy.

Obrázek 4.6: Metoda rovnoměrného rozložení

Kapitola 5

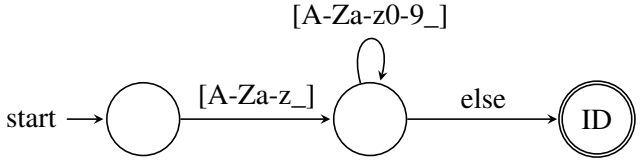
Návrh jazyka

Jedním bodem v zadání je navržení jednoduchého programovacího jazyka. Jeho návrh inspiroval opět předmět Formální jazyky a překladače. Následující odstavce blíže specifikují jeho vlastnosti a gramatiku.

5.1 Obecné vlastnosti a datové typy

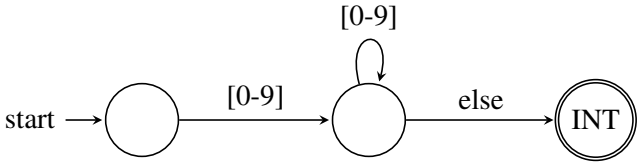
Programovací jazyk je case-sensitive (tzn. rozlišuje se velikost znaků) a typovaný (tzn. jednotlivé proměnné musí být předem deklarované). Náš jazyk rozeznává následující lexémy (za označením je uveden regulární výraz pro daný lexém):

1. **Identifikátor** `[_A-Za-z][_A-Za-z0-9]*`
   Může obsahovat pouze číslice, znaky od a-z (malé i velké) nebo znak podtržítka ('_'). Identifikátor nesmí začínat číslicí. Dále se nesmí shodovat s klíčovými slovy tohoto programovacího jazyka (`main`, `int`, `double`, `string`, `if`, `else`, `while`, `cout`, `cin`, `return`).



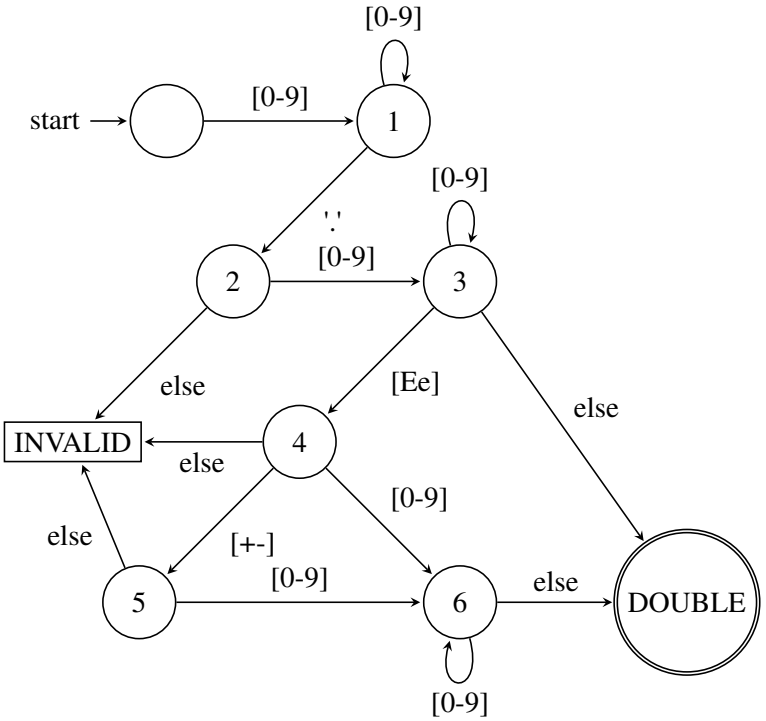
Obrázek 5.1: Konečný automat pro identifikátor

2. **Celočíselná konstanta** `[+-]?[0-9]+`
   Vyjadřuje hodnotu čísla v desítkové soustavě. Odpovídá rozsahu `int` v jazyce C.



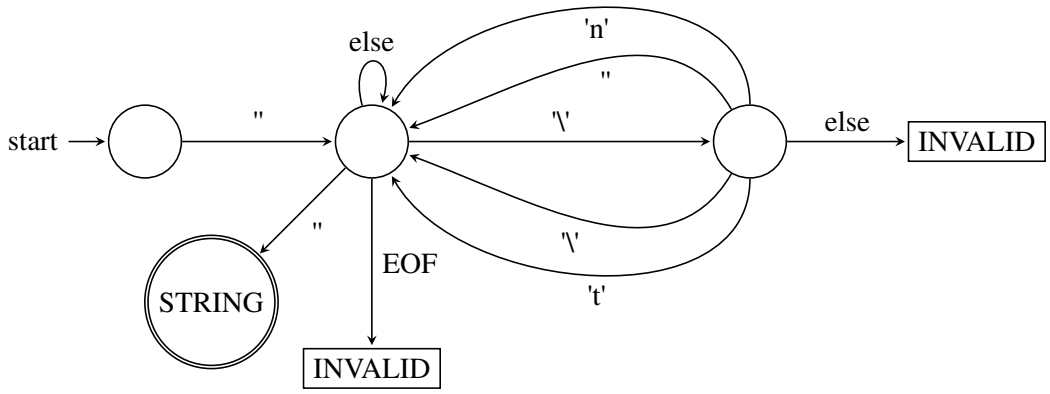
Obrázek 5.2: Konečný automat pro celočíselnou konstantu

3. **Desetinná konstanta** `[+-]?[0-9]+\.?[0-9]+([eE][-+]?[0-9]+)?`
   Odpovídá rozsahu `double` v jazyce C. Opět vyjadřuje čísla v desítkové soustavě, přičemž musí obsahovat celou část a desetinnou část, celou část a exponent nebo celou část, desetinnou část a exponent. Exponent začíná znakem 'e' nebo 'E' a před ním se může vyskytovat znaménko '+' nebo '-'. Celou a desetinnou část odděluje znak tečky ('.').



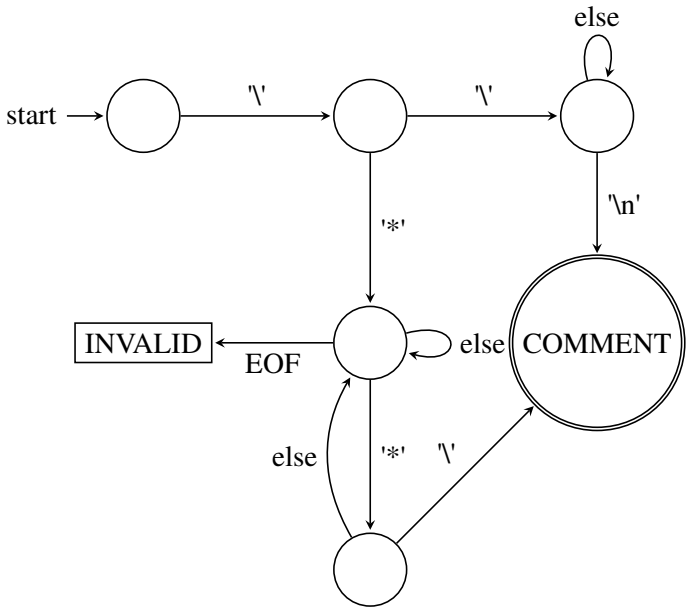
Obrázek 5.3: Konečný automat pro desetinnou konstantu

4. **Řetězcová konstanta** `"[^"]"`
   Řetězcová konstanta podobně jako v jazyce C/C++ je ohraničena uvozovkami. Obsahovat pak může jakýkoliv znak. Pokud bude konstanta uvedena na více jak jednom řádku, musí být každý řádek ukončen opačným lomítkem. Zároveň platí, že konstanta může být teoreticky nekonečná.



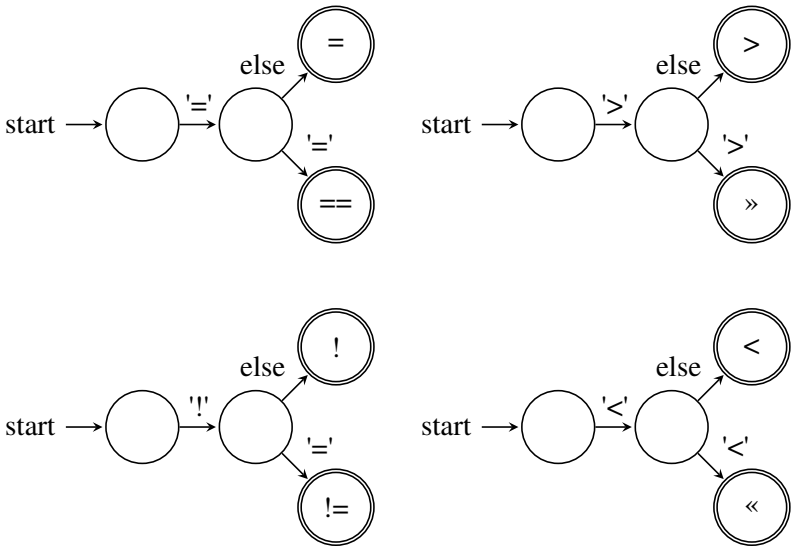
Obrázek 5.4: Konečný automat pro řetězcovou konstantu

5. **Komentáře**
   Komentář může být jednořádkový nebo víceřádkový. Na začátku jednořádkového komentáře musí být dvě lomítka a do konce řádku pak může pokračovat jakýmkoliv znakem. Víceřádkový komentář začíná lomítkem a hned za ním následuje hvězdička. Ukončen je pak opačným pořadím těchto dvou znaků. Tedy hvězdičkou a za ní následuje lomítko. Stejně jako u řetězců, i zde musíme ukončovat každý řádek opačným lomítkem.

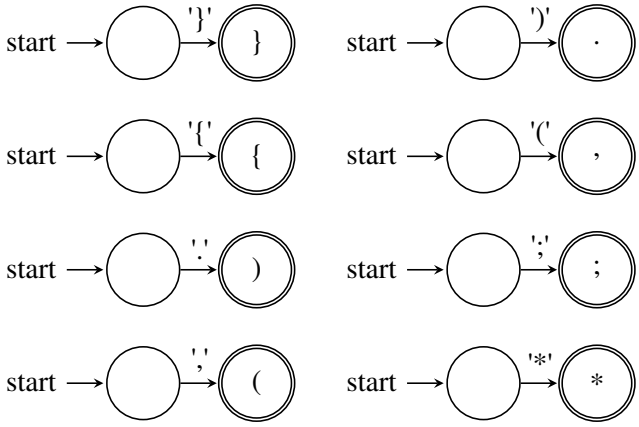


Obrázek 5.5: Konečný automat pro komentáře

6. **Operátory**
   Pro úplnost si uvedeme konečné automaty jednoznakové a dvouznakové lexémy používané většinou ve výrazech.



Obrázek 5.6: Jednotlivé konečné automaty pro dvouznakové lexémy



Obrázek 5.7: Jednotlivé konečné automaty pro jednoznakové lexémy

5.2 Syntaxe jazyka

V této sekci si něco povíme o syntaktických pravidlech pro náš jazyk. Program může obsahovat deklarace na globální úrovni. Interpret však očekává minimálně následující konstrukci *int main() { sekvence příkazů }*. Přičemž sekvence příkazů může být i prázdná. Syntaxi našeho jazyka si popíšeme pomocí bezkontextové gramatiky (viz. 2.1.4). Podle výše zmíněné definice je třeba určit abecedu neterminálů (N), abecedu terminálů (T), konečnou množinu pravidel (P) a počáteční neterminál (S).

- **Neterminály**
  <Program>, <GDList>, <DList>, <IDOrMain>, <STList>, <Stat>, <RightCin>, <RightCout>, <Condition>, <Expr>
- **Terminály**
  int, string, double, id, main, if, else, while, (,), {, }, ;, =
- **Konečná množina pravidel**
  1. <Program> → <GDList>
  2. <GDList> → int <IDOrMain>
  3. <GDList> → string id ; <GDList>
  4. <GDList> → double id ; <GDList>
  5. <IDOrMain> → id ; <GDList>
  6. <IDOrMain> → main ( ) { <STList>
  7. <DList> → double id ; <STList>
  8. <DList> → int id ; <STList>
  9. <DList> → string id ; <STList>
  10. <STList> → <stat> <STList>
  11. <STList> → <DList>
  12. <STList> → <}>
  13. <Stat> → <Expr> ;
  14. <Stat> → cin <RightCin>
  15. <Stat> → cout <RightCout>
  16. <Stat> → <Condition>
  17. <Stat> → return <Expr> ;
  18. <Condition> → if <Expr> { <STList> else { <STList>
  19. <Condition> → while <Expr> { <STList>
  20. <RightCout> → « <Expr> <RightCout>
  21. <RightCout> → ;
  22. <RightCin> → » id <RightCin>
  23. <RightCin> → ;
  24. <Expr> → id = <Expr> ;
  25. <Expr> → (<Expr>) ;
- **Počáteční neterminál**
  <Program>

5.3 Sémantika příkazů

1. `cin` $>> id_1 >> id_2 >> \ldots >> id_n$
   Tento příkaz slouží pro načtení hodnot ze standardního vstupu do proměnných. Příkaz začíná klíčovým slovem `cin`, dále pak pokračuje dvojznakem $>>$ (dvě většítka ihned za sebou), za kterým je očekávána již dříve deklarovaná proměnná. V příkazu může být uvedeno nekonečně mnoho proměnných, avšak musí být oddělené dvojznakem $>>$. Příkaz musí končit středníkem. Sekvence, která následuje za klíčovým slovem `cin` musí být neprázdná (tzn. musí obsahovat minimálně jednu proměnnou).

   Ze standardního vstupu načítá jednotlivé hodnoty a ukládá je do daných proměnných v takovém pořadí, v jakém jsou uvedeny v příkazu. Zapsané proměnné musí být typu, který může uchovávat hodnotu zadanou na standardním vstupu (tzn. typ hodnoty na standardním vstupu musí souhlasit s typem proměnné, do které je hodnota ukládána).

2. `cout` $<< vyraz_1 << vyraz_2 << \ldots << vyraz_n$
   Tento příkaz slouží pro výpis zadaných proměnných na standardní výstup. Příkaz začíná klíčovým slovem `cout`, dále pak pokračuje dvojznakem $<<$ (dvě menšítka ihned za sebou), za kterým je očekáván výraz. Popisem výrazů se budeme zabývat později v tomto textu. Podobně jako v předchozím případě, tak i tento příkaz může obsahovat nekonečně mnoho výrazů. Mezi nimi musí být dvojznak $<<$ a musí být uveden alespoň jeden výraz. Za posledním výrazem se očekává středník.

   Postupně vyhodnocuje výrazy a jejich hodnotu vypisuje na standardní výstup ihned za sebe. Nijak tedy nezasahuje do formátování výstupních dat.

3. `if` *vyraz* { *sekvence prikazu 1* } `else` { *sekvence prikazu 2* }
   Nejdříve se vyhodnotí výraz. Výsledek výrazu musí být celé číslo, jinak půjde o sémantickou chybu. Znamená to, že ve výrazu se nesmí objevit například řetězcová konstanta. Pokud je hodnota výrazu rovna 0, tak je výraz považován ze nepravdivý. Ve všech ostatních případech je pravdivý. Sekvence příkazů 1 se provede právě tehdy, pokud je výraz pravdivý. Sekvence příkazů 2 se provede v ostatních případech. Jak sekvence příkazů 1 tak sekvence příkazů 2 mohou být i prázdné.

4. `while` *vyraz* { *sekvence prikazu* }
   Pravidla pro určení pravdivosti výrazu jsou stejná jako u předchozího příkazu. Tento příkaz opakuje sekvenci příkazů ve svém těle tak dlouho, dokud je výraz pravdivý.

5. `return` *vyraz*;
   Program je ukončen a je předána návratová hodnota výrazu. Tato hodnota musí být typu `int`.

6. *vyraz*;
   Příkaz reprezentující výraz musí být ukončený středníkem. O výrazech si povíme později v textu.

5.4 Výrazy

Pro výrazy platí následující gramatika.

- **Neterminály**
  <E>
- **Terminály**
  (,), i, /, *, +, -, >, >=, <, <=, ==, !=, =
- **Konečná množina pravidel**

  1. <E> → <E> + <E>
  2. <E> → <E> - <E>
  3. <E> → <E> * <E>
  4. <E> → <E> / <E>
  5. <E> → <E> < <E>
  6. <E> → <E> <= <E>
  7. <E> → <E> > <E>
  8. <E> → <E> >= <E>
  9. <E> → <E> == <E>
  10. <E> → <E> != <E>
  11. <E> → <E> = <E>
  12. <E> → (<E>)
  13. <E> → i

- **Počáteční neterminál**
  <E>

Priorita operátorů a jejich asociativita je znázorněna v tabulce 5.1.

Priorita	Operátor	Asociativita
1	* /	→
2	+ -	→
3	< <= > >=	→
4	== !=	←
5	=	←

Tabulka 5.1: Priorita operátorů

Kapitola 6

Použité metody a implementace

V této kapitole se dozvíte něco o konkretní implementaci interpretu využívajícího paralelní analýzy.

6.1 Volba prostředků

Projekt bude vyvíjený pod operačním systémem UNIX s využitím jazyka C/C++. Ve většině případů se jedná o objektový návrh. Pokud bych použil pro implementaci samotný jazyk C, zřejmě by byl běh programu o pár setin sekundy (u větších souborů i sekund) rychlejší, ale cílem projektu nebylo zkonstruovat nejrychlejší interpret. Cílem bylo vytvořit paralelní analýzu, která by mohla přinést zrychlení oproti klasické sekvenční analýze. Což se samozřejmě bude testovat na stejné aplikaci. Navíc volba jazyka C/C++ nám dává mnohem větší přehlednost ve zdrojových kódech.

6.2 Volba metod

Program bude spouštět společně s lexikální analýzou i syntaktickou analýzu. Navíc se aplikace nebude omezovat na spuštění pouze dvou procesů. Počítá tak do budoucna i s tím, že součástí počítačů budou více jak dvoujádrové procesory. Pro nás to znamená více syntaktických stromů a nutné spojení těchto stromů po dokončení jednotlivých analýz. Vyloučil jsem metodu zpětného procesu (viz. 4.4.1), kde by bylo nutné sestrojit „zpětný konečný automat“ a i syntaktická analýza by musela postupovat zpětně. Dle mého názoru je nejlepší volbou metoda rovnoměrného rozložení, která jednoduše dovolí spustit více jako dva procesy. Tedy každý spuštěný proces pro sebe získá blok ze vstupního souboru, který analyzuje. Spuštění více jak dvou procesů na dvoujádrovém počítači (tj. testovací počítač) sice nepovede k reálnému zrychlení, ale pomocí výpisů uvidíme, že s lepším vybavením bychom docílili lepších výsledků.

Nutné také bylo vyřešit problémy z hlediska paměti a komunikace (viz. 4.2) mezi procesy. Aplikace je navržena tak, aby nemusela používat společné zdroje (resp. je používala co nejméně) a tím minimalizovala komplikace spojené s paralelním programováním (viz. 3). Společnou mají pouze jednu proměnnou `counter`, která slouží pro výpočet jmen pomocných proměnných ukládaných v tabulce symbolů. Tím docílíme toho, že jednotlivé procesy se uspávají co nejméně a nemusí čekat na ostatní. Jejich práce se tak výrazně urychlí. Přístup k této kritické sekci (viz. 3.2) se řídí pomocí semaforu (viz. 3.2.1).

6.3 Implementace tabulky symbolů

Tabulka symbolů je popsána v sekci 2.1.5. Ze dvou možných principů, kdy ukládat symboly do tabulky, vybereme ukládání při sémantické analýze. Prakticky ani žádnou jinou volbu nemáme (viz. 4.3).

Pro samotnou implementaci tabulky symbolů jsem využil binární vyhledávací strom, který představuje prostředek pro časté, ale hlavně rychlé vyhledávání.

6.4 Volba komunikace

Z výše uvedených odstavců plyne potřeba vymyslet vnitřní reprezentaci struktury zdrojového kódu – pro pozdější sémantickou kontrolu. Kdybychom tedy zdrojový kód nechtěli interpretovat, dalo by se říci, že výstupem aplikace budou právě tyto struktury. Ty se budou uchovávat v xml souborech na disku (více o xml v [3] a [4]). Pro práci s tímto formátem byla použita knihovna TinyXML (viz. [6]). Volba komunikace přes pevný disk patrně velice zpomalí běh jednotlivých analýz, které na něj musí neustále přistupovat. Vše se však ukládá jen kvůli pozdější interpretaci a ukázce, že lze takové analýzy v praxi využít. Kdybychom jako ukázku využili například textový editor se zvýrazňováním a kontrolou syntaxe, tak by se vnitřní reprezentace struktury zdrojového kódu vůbec ukládat nemusela. Samozřejmě aplikace umožňuje pomocí parametrů vypnout samotnou interpretaci a tedy i přístup na disk. Výsledkem aplikace je pak pouze lexikální a syntaktická analýza zdrojového souboru. Tedy zpráva o tom, zda je zdrojový soubor zapsán podle daných pravidel, či nikoliv. Tato možnost se hodí při srovnávání časů sekvenční a paralelní analýzy.

6.5 Použití aplikace

Aplikaci je možné použít jako interpret smyšleného jazyka popsaného v kapitole 5. Je nutné, abychom přes parametr `-F` zadali jméno souboru se zdrojovým kódem. Pro testování si můžeme zapnout vypisování statistik (přepínač `-D`), kde se dozvíme kolik tokenů jaký proces zpracoval a za jak dlouho. Pomocí přepínače `-I` si můžeme nastavit, zda chceme, aby aplikace interpretovala zdrojový kód nebo pouze provedla lexikální a syntaktickou analýzu. Kvůli zvolené komunikaci (viz. 6.4) se zde analýza urychlí až o 50 %. V neposlední řadě můžeme využít přepínač `-P`, kterým určíme kolik procesů se má podílet na analýze. Jak už jsem psal v sekci 6.2, zvolením více procesů sice získáme rychlejší časy strávené na procesoru jednotlivých syntaktických analýz, ale pokud nemáme potřebný hardware, tak reálné zrychlení nijak nepocítíme.

6.6 Činnost aplikace

Činnost aplikace bychom mohli rozdělit do třech logických bloků. Právě tato sekce popisuje detailněji každý tento blok.

Rozdělení souboru

Prvním krokem je rozdělení souboru na daný počet částí. Aplikace přitom dodržuje zásad uvedených v sekci 4.1. Hlavně to, aby soubor nebyl rozdělen uprostřed víceřádkového lexému. Nejdříve se soubor rozdělí rovnoměrně a poté se dohledává pomocí „hladšího“ rozdělení nějaký vyhovující bod pro ukončení bloku (nejčastěji hned za středníkem, který ukončuje konstrukce). Počet bloků je přímo úměrný počtu zvolených procesů s tím, že se počet procesů ještě vnitřně reguluje podle

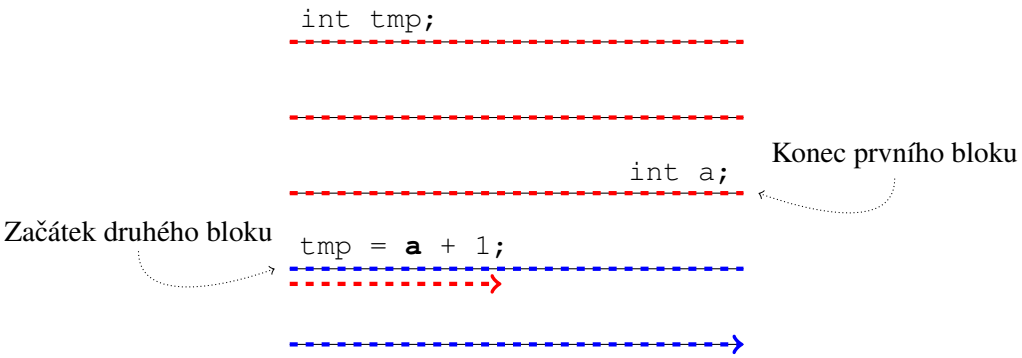
velikosti souboru. Tedy například pro dvaceti řádkový soubor si nemůžeme zvolit stovku procesů. Respektive můžeme, ale aplikace je sníží.

Analýza

Následuje samotná analýza souboru. Pomocí systémového volání `fork` se vytvoří požadováný počet procesů. Každý proces dostane své jedinečné `id`. Pod tímto číslem se pak dostane k údajům, kde začíná a kde končí jeho blok ve vstupním souboru. Od tohoto momentu každý proces pracuje stejně jako klasický sekvenční analyzátor. Tímto bodem také začíná odpočet času, který na konci získáme ve výpisu statistik. Aby se zkontrolovalo i správné napojení na následující blok, tak proces vždy o pár lexémů svůj vymezený prostor přesáhne. Nutné je, aby sémantické akce spojené s tímto přesahem již do svého xml souboru nezapsal, protože ty se objevují na začátku xml souboru následujícího procesu. Syntaktická analýza každého procesu si uchovává statistiku o svém průběhu a my si jí můžeme nechat vypsat. Nejdůležitější veličinou pro tento projekt je čas, za který syntaktickou analýzu proces provede. Po dokončení syntaktické analýzy proces zapíše statistiky do sdíleného pole a svou činnost ukončí. Dál už pokračuje pouze rodičovský proces, který ostatní procesy vytvořil.

Interpretace

Rodičovský proces tedy počká na dokončení všech synovských procesů a pokračuje k interpretaci kódu. (V tomto bodě rodič ukončí a zapíše celkový čas analýzy všech procesů.) Pokud některý s procesů skončil se syntaktickou chybou, rodič vypíše detaily o chybě a končí. V opačném případě následuje sémantická kontrola. Postupně otevírá všechny vytvořené soubory a vytváří intermediární kód (viz. 2.2.4). Při tomto generování je zároveň kontrolováno správné ukončení všech bloků. Zbývá už jen vytvořený mezikód interpretovat.



Obrázek 6.1: Přesah procesu

Kapitola 7

Testovaní

Dostáváme se ke kapitole, kvůli které jsme celý projekt tvořili. Zde si ověříme, zda naše snažení bylo úspěšné a jaké přineslo výsledky. Nejprve se však seznámíme s podobou statistik, které produkuje každý proces samostatně.

7.1 Statistiky

Následující výpis statistik dostaneme pro každý spuštěný proces. První řádek v tabulce nám označuje proces. Další dva řádky se věnují času. Jeden čas je reálný (ten, po který uživatel skutečně čeká na dokončení) a druhý čas určuje, jak dlouho proces zaměstnal procesor. Do měření času je zapojena pouze lexikální a syntaktická analýza. Neměří se tedy sémantická analýza, generování kódu a ani interpretace. Počet analyzovaných tokenů a řádků snad není třeba blíže vysvětlovat. Upozorním jen na skutečnost, která na první pohled není jasná. Díky přesahu kontroly přes své přidělené bloky, nám vzniká situace, kdy při stejně zvoleném souboru se nám tyto počty zvyšují s přibývajícím počtem procesů. Na závěr je uvedena statistika celková. Zde je informace o celkovém počtu analyzovaných tokenů a reálný čas všech paralelně běžících analýz – opět pouze lexikálních a syntaktických.

Proces: 0	
Čas (CPU) analýzy:	0.00s
Čas (Real) analýzy:	0.00s
Počet analyzovaných tokenu:	0
Počet analyzovaných řádků:	0

Tabulka 7.1: Podoba výstupních statistik

7.2 Výsledky

V této sekci si uvedeme konkrétní výsledky aplikace na testovacím počítači[1] při stejně velkých souborech. Následující testy je možné spustit pomoci souboru `test.sh` umístěného na přiloženém cd.

[1] AMD Athlon 64 X2 Dual Core 4200+ (2x 2.2GHz), 1,5GB RAM

Test 1

Následuje test velkého souboru (přibližně 1.5 milionu řádků) postupně pro 1, 2 a 4 procesy. Tento test byl spuštěn třicetkrát bez interpretace zdrojového textu. Časové údaje v tabulkách představují průměry těchto výsledků.

Proces: 0	
Čas (CPU) analýzy:	7.74s
Čas (Real) analýzy:	7.74s
Počet analyzovaných tokenů:	1 920 507
Počet analyzovaných řádků:	1 537 019

Tabulka 7.2: Test velkého souboru (1.5 mil. řádků) **jedním** procesem

Proces: 0		Proces: 1	
Čas (CPU) analýzy:	4.52s	Čas (CPU) analýzy:	3.98s
Čas (Real) analýzy:	5.50s	Čas (Real) analýzy:	5.02s
Počet analyzovaných tokenů:	961 923	Počet analyzovaných tokenů:	958 588
Počet analyzovaných řádků:	768 707	Počet analyzovaných řádků:	768 315

Tabulka 7.3: Test velkého souboru (1.5 mil. řádků) **dvěma** procesy

Proces: 0		Proces: 1	
Čas (CPU) analýzy:	2.86s	Čas (CPU) analýzy:	2.90s
Čas (Real) analýzy:	9.11s	Čas (Real) analýzy:	9.16s
Počet analyzovaných tokenů:	482 647	Počet analyzovaných tokenů:	479 280
Počet analyzovaných řádků:	384 535	Počet analyzovaných řádků:	384 179
Proces: 2		**Proces: 3**	
Čas (CPU) analýzy:	2.87s	Čas (CPU) analýzy:	2.59s
Čas (Real) analýzy:	9.10s	Čas (Real) analýzy:	9.00s
Počet analyzovaných tokenů:	479 289	Počet analyzovaných tokenů:	479 303
Počet analyzovaných řádků:	384 185	Počet analyzovaných řádků:	384 133

Tabulka 7.4: Test velkého souboru (1.5 mil. řádků) **čtyřmi** procesy

Můžeme si všimnout, že paralelní analýza dvěma procesy zde byla rychlejší přibližně o 40 % reálného času. Délka času, kterým byl zaměstnán procesor se snížil zhruba o polovinu. Celkový počet tokenů je zvýšen o 4. Toto je způsobeno přesahem, který kvůli syntaktické kontrole musel provést první proces. Tabulka 7.4 uvádí výsledky analýzy čtyřmi procesy. Jak vidíme, u kontroly čtyřmi procesy byl opět zvýšen počet kontrolovaných tokenů. Každý proces zaměstnával procesor zase o něco méně. Přibližně jde o 40% pokles. Reálný čas je téměř dvojnásobný jako u analýzy dvěma procesy. Důvodem je nedostatečná HW vybavenost testovacího počítače. O dva procesy se stará pouze jeden procesor, čímž je způsobeno toto zvýšení.

Test 2

V tomto testu budeme analyzovat menší soubor (cca. 350 tis. řádků) i s volbou interpretace. Test byl opět spuštěn třicetkrát a v tabulce jsou uvedeny průměrné hodnoty.

Proces: 0	
Čas (CPU) analýzy:	4.04s
Čas (Real) analýzy:	4.04s
Počet analyzovaných tokenů:	283 254
Počet analyzovaných řádků:	322 772

Celkem	
Tokenů celkem:	283 254
Čas (Real) celkem:	4.29s

Tabulka 7.5: Test souboru (350 tis. řádků) **jedním** procesem + interpretace

Proces: 0	
Čas (CPU) analýzy:	2.19s
Čas (Real) analýzy:	2.29s
Počet analyzovaných tokenů:	141 630
Počet analyzovaných řádků:	161 397

Proces: 1	
Čas (CPU) analýzy:	2.05s
Čas (Real) analýzy:	2.20s
Počet analyzovaných tokenů:	141 629
Počet analyzovaných řádků:	161 379

Celkem	
Tokenů celkem:	283 259
Čas (Real) celkem:	2.35s

Tabulka 7.6: Test souboru (350 tis. řádků) **dvěma** procesy + interpretace

Proces: 0	
Čas (CPU) analýzy:	0.74s
Čas (Real) analýzy:	0.92s
Počet analyzovaných tokenů:	141 630
Počet analyzovaných řádků:	161 397

Proces: 1	
Čas (CPU) analýzy:	0.65s
Čas (Real) analýzy:	0.82s
Počet analyzovaných tokenů:	141 629
Počet analyzovaných řádků:	161 379

Celkem	
Tokenů celkem:	283 259
Čas (Real) celkem:	0.93s

Tabulka 7.7: Test souboru (350 tis. řádků) **dvěma** procesy

Můžeme sledovat, že opět byla paralelní analýza přibližně o 40 % rychlejší (což je zřejmé z tabulek 7.5 a 7.6). Tabulka 7.7 představuje výsledek analýzy stejného souboru s vypnutou interpretací. Velký rozdíl v časech způsobuje ukládání do souboru (viz. 6.4).

Kapitola 8

Závěr

Výsledkem projektu je funkční interpret jazyka, který je navržený a popsaný v kapitole 5. Nicméně mnohem větší důraz byl kladen na paralelní analýzu, jejímž výstupem jsou struktury reprezentující úseky zdrojového souboru. Zde projekt nechává rezervu na případné rozšíření v dalších verzích aplikace. Uložení těchto struktur a následné sjednocení kvůli interpretaci si žádá určitý čas. Kdybychom uvažovali o využití paralelní analýzy pouze společně s interpretem, tak by zřejmě bylo lepší toto ukládání implementovat přes sdílenou paměť než přes xml soubory (případně jiným rychlejším prostředkem). Analýzu je však možné využít například i pro označování a kontrolu syntaxe zdrojových kódů v grafických editorech, kde by zřejmě ukládání nebylo vůbec potřeba.

Pokud bychom zůstali u interpretu, tak se jako další zajímavé rozšíření jeví interpretace spuštěná společně s generováním vnitřního kódu. V naší verzi interpret čeká dokud se nevygeneruje mezikód až do konce souboru. Kdyby se interpret spouštěl již s první vygenerovanou instrukcí, mohlo by tak například generování pokračovat i kdyby interpret čekal na vstup od uživatele.

Tento projekt počítá pouze s vlastním navrženým jazykem. Kdybychom však chtěli aplikovat analýzu na již známé jazyky (např. jazyk C) museli bychom aplikaci rozšířit o možnost zotavení se z chyb. V tomto stavu by například nedokázala analyzovat rozdělené víceřádkové lexémy, které by nebyly ukončeny opačným lomítkem.

Celkově aplikace přinesla zrychlení analýzy zdrojového souboru, což bylo hlavním cílem tohoto projektu. Jako vedlejší cíl jsme si již v kapitole 6 zvolili spuštění více jak dvou procesů k analyzování. Dalším bodem pak byla i možnost jednoduše separovat syntaktickou a lexikální analýzu od interpretu a využít ji například pro výše zmínění grafický editor. Tento bod se povedlo splnit tím, že výstupem analýz jsou vnitřní struktury namísto mezikódu. S těmito strukturami pak můžeme naložit podle konkretního využití. Spuštění více jak dvou procesů se nám povedlo docílit vhodně zvolenou metodou, která je popsána v kapitole 4.

Aplikace byla nejdříve testována pomocí analýzy jednotlivých příkazů, které jsou součásti jazyka. A to jak jedním procesem, tak i více procesy. Odladěna byla tak, aby výsledky všech příkazů byly správné. Poté následovalo testování větších zdrojových souborů se zaměřením hlavně na jejich správnou analýzu v bodě rozdělení. Výsledky tohoto testování můžeme najít v kapitole 7.

Program byl úspěšně testován pouze na operačním systému UNIX. Pro spuštění na systémech MS Windows by bylo potřeba změnit systémové volání pro rozdělování procesu. Jak už bylo zmíněno výše v tomto textu, toto volání je totiž odlišné od volání v systémech UNIX.

Literatura

[1] Alexander Meduna: *Automata and Languages*. London: Springer, 2000, ISBN 978-1-85233-074-3.

[2] Kašpárek, T.; Kočí, R.; Peringer, P.; aj.: Operační systémy. [online], 2006-11-25 [cit. 2010-05-14].
URL https://www.fit.vutbr.cz/study/courses/IOS/public/IOS-opora.pdf

[3] Kolář, D.: XML aneb nový formát pro nové tisíciletí. [online], Poslední aktualizace: 2001-01-17 [cit. 2010-05-14].
URL http://www.builder.cz/art/html/xml_uvod.html

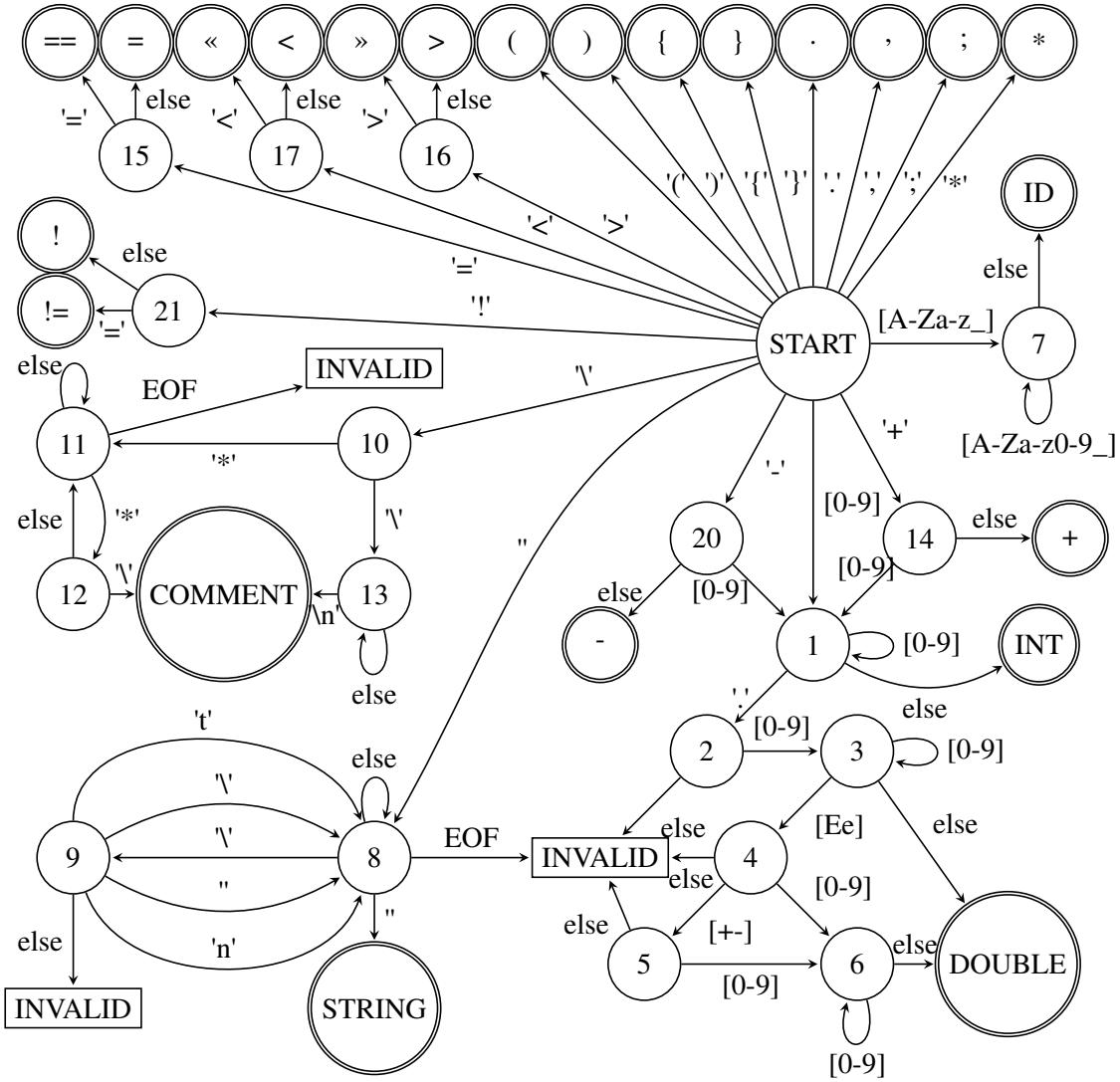
[4] Kolář, D.: Zápis správné syntaxe XML dokumentů. [online], Poslední aktualizace: 2001-01-22 [cit. 2010-05-14].
URL http://www.builder.cz/art/html/xml_syntaxe.html

[5] Matys, J.: Programování pod Linuxem pro všechny. [online], Poslední aktualizace: 2004-02-07 [cit. 2010-05-14].
URL
http://www.root.cz/clanky/programovani-pod-linuxem-semafory/

[6] Thomason, L.; Berquin, Y.; Ellerton, A.: TinyXML. [online], Poslední aktualizace: 2004-03-20 [cit. 2010-05-14].
URL http://www.grinninglizard.com/tinyxml/

Dodatek A

Konečný automat



Dodatek B

Metriky kódu

Počet souborů: 17 souborů

Počet řádků zdrojového textu: 5425 řádků

Velikost statických dat: 136109B

Velikost spustitelného souboru: 230692B (systém Linux, 64bitová architektura, překlad bez ladících informací)