VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

PREKLADAČ Z FRAGMENTU JAZYKA C DO NÁSTROJA ARTMC

COMPILER OF C LANGUAGE FRAGMENT TO ARTMC TOOL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MATEJ MARUŠÁK**
AUTHOR

VEDOUCÍ PRÁCE **Doc. Mgr. ADAM ROGALEWICZ, Ph.D.**
SUPERVISOR

BRNO 2017

Zadání bakalářské práce/19558/2016/xmarus06

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Marušák Matej**
Obor: Informační technologie
Téma: **Překladač z fragmentu jazyka C do nástroje ARTMC**
Compiler of C Language Fragment to ARTMC Tool
Kategorie: Překladače

Pokyny:

1. Nastudujte vstupní formát nástroje ARTMC.
2. Vyberte vhodnou podmnožinu jazyka C s ohledem na možnosti nástroje ARTMC.
3. Navrhněte způsob překladu vybrané podmonožiny jazyka C do vstupního formátu nástroje ARTMC.
4. Navržený překladač implementujte.
5. Výsledný překladač otestujte na příkladech z distribuce nástroje ARTMC a dále na sadě 10 nových demonstračních příkladů.
6. Diskutujte možné pokračování a rozšíření tohoto projektu.

Literatura:

- Bouajjani et al.: Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. Proc. of SAS'06.
- Domovská stránka projektu ARTMC: http://www.fit.vutbr.cz/research/groups/verifit/tools/artmc/

Pro udělení zápočtu za první semestr je požadováno:

- První 3 body zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Rogalewicz Adam, Mgr., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

S narastajúcou komplexitou softvérových programov je stále viac a viac žiadaná automatizovaná analýza a verifikácia týchto programov. Výskumná skupina VeriFIT na Fakulte informačních technologií Vysokého učení technického sa zaoberá výskumom v danej oblasti. Jedným z vytvorených nástrojov v tejto skupine je aj nástroj ARTMC. Táto bakalárska práca navrhuje a implementuje prekladač z podmnožiny jazyka C do vstupného formátu nástroja ARTMC. Vytvorený prekladač výrazne uľahčuje prácu s nástrojom ARTMC, nakoľko vstupný formát nie je vhodný na manuálné vytváranie.

Abstract

With growing complexity of software programs the need for automated analysis and verification grows as well. Reasearch group VeriFIT based on Faculty of Information Technology of Brno University of Technology is involved in research of this area. One of the developed tools is the ARTMC tool. This bachelor's thesis designs and implements compiler of C language fragment into input format of the ARTMC tool. Implemented compiler makes work with ARTMC tool much easier, since the input format is not suitable for manual creation.

Kľúčové slová

ARTMC, prekladač, C, verifikácia, Python

Keywords

ARTMC, compiler, C, verification, Python

Citácia

MARUŠÁK, Matej. *Prekladač z fragmentu jazyka C do nástroja ARTMC*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Rogalewicz Adam.

Prekladač z fragmentu jazyka C do nástroja ARTMC

Prehlásenie

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doc. Mgr. Adama Rogalewicza, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Matej Marušák
6. mája 2017

Poďakovanie

Ďakujem svojmu vedúcemu doc. Mgr. Adamovi Rogalewiczovi, Ph.D za cenné rady, ochotu a pomoc pri všetkých častiach tvorby tejto práce.

Obsah

1 Úvod **3**

2 Nástroj ARTMC **4**
2.1 Funkcionalita a použitie ARTMC 4
2.2 Formát vstupného súboru 4
2.2.1 typedefs 4
2.2.2 program.py 5
2.2.3 Popis príkazov jazyka pre ARTMC 6
2.2.4 env 7
2.2.5 Nastavovanie deskriptoru 7
2.2.6 Podpora práce s dátami v ARTMC 7
2.3 Tvorba programu v ARTMC 8

3 Prekladače **10**
3.1 Lexikálny analyzátor 10
3.2 Syntaktický analyzátor 10
3.2.1 Analýza zhora-nadol 10
3.2.2 Analýza zdola-nahor 10
3.3 Sémantický analyzátor 12
3.4 Generátor vnútorného kódu 12
3.5 Optimalizátor 12
3.6 Generátor cieľového kódu 13
3.7 Syntaxou riadený preklad 13

4 Súčasný stav prekladača pre ARTMC **14**

5 Návrh implementácie **15**
5.1 Použiteľnosť a prenositeľnosť 15
5.1.1 Python 16
5.2 Udržiavateľnosť a rozširovateľnosť 16
5.2.1 Git 16
5.3 Testovateľnosť 16
5.3.1 Hromadné testy 16
5.3.2 Tox 17

6 Výsledná aplikácia **18**
6.1 Výsledná implementácia 18
6.1.1 Lexikálny analyzátor 18

6.1.2 Syntaktický analyzátor . . . 19
6.1.3 Sémantický analyzátor . . . 19
6.1.4 Generovanie kódu . . . 19
6.2 Vstupný program . . . 20
6.2.1 Nepodporované konštrukcie . . . 20
6.2.2 Neštandardné podporované konštrukcie . . . 21
6.2.3 Povinné položky . . . 21
6.2.4 Nepovinné položky . . . 21
6.3 Práca s dátami . . . 22
6.3.1 Ignorovanie dát . . . 22
6.3.2 Využitie štandardných premenných . . . 22
6.4 Použitie aplikácie . . . 23
6.5 Neštandardné vlastnosti . . . 23
6.5.1 Preskakovanie (častí) príkazov . . . 23
6.5.2 Preklad viacnásobných pointrov . . . 24
6.5.3 Vyhodnocovanie neúplných podmienok . . . 24
6.5.4 Skratové vyhodnocovanie podmienok . . . 24
6.6 Testy . . . 25
6.7 Navrhovaná práca . . . 25

7 Záver **26**

Literatúra **27**

Prílohy **28**

A Obsah CD **29**

B Zoznam testov **30**

Kapitola 1

Úvod

Verifikácia programov, teda overovanie že program spĺňa určité požiadavky, je v dnešnej dobe veľmi žiadaná. Napriek záujmu zo strán tvorcov softvéru neexistuje skutočne univerzálne a dokonalé riešenie. Avšak mnoho subjektov vytvára verfikátory, ktoré dokážu kontrolovať určité požadované vlastnosti programov.

Jedným z aktívnych subjektov pre verifikáciu a automatizovanú analýzu je aj výskumná skupina VeriFIT na Vysokom Učení Technickom v Brne. Skupina se zaoberá základným výskumom, ale taktiež aj vývojom prototypových verifikačných nástrojov. Jedným z takýchto nástrojov je aj ARTMC[1].

Cieľom tejto práce je implementovať prekladač, ktorý dokáže z validného programu napísanom v jazyku C vytvoriť validný, funkčne rovnaký, program vo vstupnom formáte nástroja ARTMC. ARTMC bol hlavne vyvíjaný na ukázanie možností verifikácie programov, ktoré pracujú s dynamicky viazanými dátovými štruktúrami. Počas vývoja sa nekládol dôraz na použiteľnosť ale skôr na funkčnosť. Od začiatku vývoja sa počítalo s prekladačom, ktorý ale nikdy nevznikol. Tento fakt má za následok veľmi obtiažne manuálne tvoriteľný vstupný formát.

Prekladač by mal byť dostatočne robustný aj napriek veľmi malej podmnožine jazyka C, ktorú je nástroj ARTMC schopný spracovať. V prípade konštrukcie, ktorá nie je podporovaná zo strany ARTMC, by sa mal zachovať adekvátne, či to už znamená skončenie chybou a oznámením čo konkrétne nie je možné spracovať, alebo dané príkazy preskočiť. Ďalej sa očakáva čitateľnosť a udržiavateľnosť napísaného prekladača pre ďalšie vylepšovanie a dopĺňanie funkcionality. V neposlednom rade je dôležitá aj prenositeľnosť.

[1] http://www.fit.vutbr.cz/research/groups/verifit/tools/artmc/

Kapitola 2

Nástroj ARTMC

ARTMC je nástroj na formálnu verifikáciu programov, ktoré manipulujú s dynamicky viazanými dátovými štruktúrami. Tento nástroj bol vyvinutý na Fakulte informačných technológií Vysokého Učení Technického v Brne v rámci výskumnej skupiny VeriFIT, ktorá sa zaoberá verifikáciou programov. V tejto kapitole je tento nástroj popísaný z dvoch hľadísk – najskôr je stručný prehľad funkcionality nástroja a následne je popísaná štruktúra vstupného formátu, ktorý je pre túto prácu najdôležitejší.

2.1 Funkcionalita a použitie ARTMC

Majme na vstupe nerekurzívny program, ktorý manipuluje s dynamicky viazanými štruktúrami s viacnásobnými next-ukazateľmi. Snahou nástroja je overiť, že nemôžu nastať nedovolené operácie (napr. zápis do null ukazateľa, použitie nedefinovaného alebo už zmazaného prvku atď.). [3]

Verifikačná metóda, ktorá je použitá v tomto nástroji je založená na abstraktom regulárnom stromovom model checkingu. [2] V rámci tejto metódy je konfigurácia pamäte programu reprezentovaná stromom nad konečnou abecedou. Nekonečná množina konfigurácií je reprezentovaná stromovými automatmi, ktoré sú zobecnením konečných automatov. Technika následne počíta množinu dosiahnuteľných konfigurácií reprezenotavaných opäť stromovým automatom.

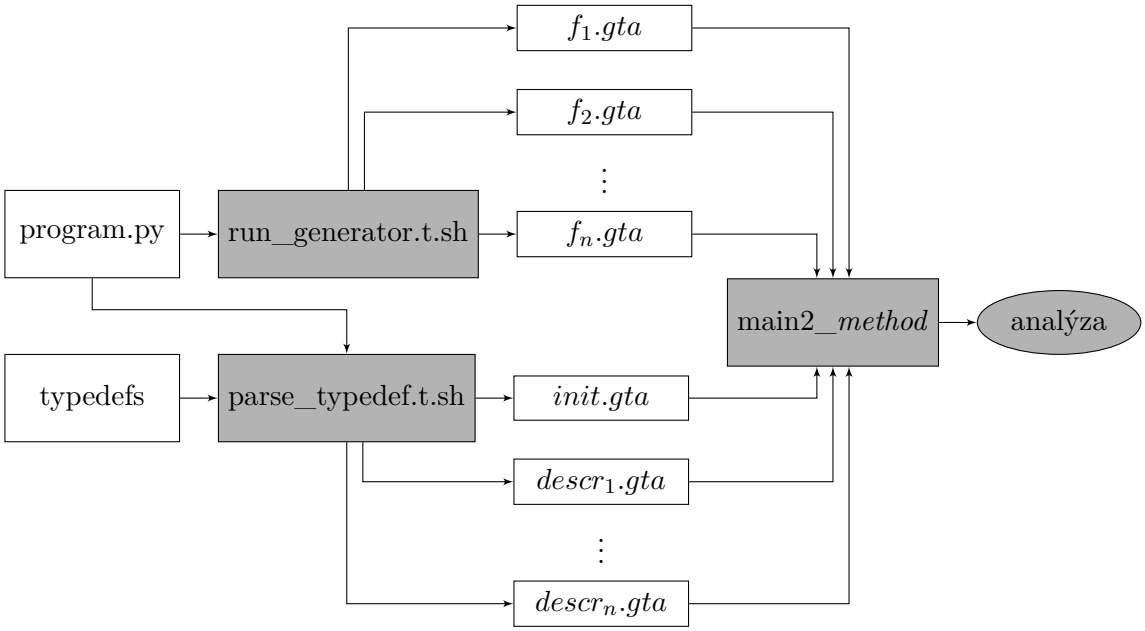
Nástroj má na vstupe dva súbory, ktoré sa za pomoci dvoch skriptov prevedú na desiatky súborov, ktoré reprezentujú automaty. Z týchto súborov sa následne vytvára analýza. Schéma fungovania nástroja ARTMC je znázornená na obrázku 2.1.

2.2 Formát vstupného súboru

Nástroj ARTMC potrebuje pre svoju analýzu dva súbory – jedná sa o súbor `typedefs`, ktorý obsahuje definíciu množiny počiatočnej konfigurácie a súbor `program.py`, obsahujúci definíciu programu vo formáte vytvorenom pre potreby tohto nástroja.

2.2.1 typedefs

Tento súbor sa používa na špecifikáciu množiny počiatočnej konfigurácie, ktorá môže byť obecne nekonečná (napr. zoznam ľubovoľnej dĺžky). Súbor obsahuje množinu typov. Následne je pamäť zostavená z uzlov špecifikovaných typov, ktoré sú prepojené za pomoci selektorov.



Obr. 2.1: Schéma práce nástroja ARTMC

Existujú dva druhy selektorov:

- *Priame spojenia*
- *Nepriame spojenia* sú popísané za pomoci cesty cez priame spojenia. Za pomoci symbolu "-" pred priamym spojením sa značí nasledovanie spojenia smerom späť.

Súbor `typedefs` pre obojsmerne viazané listy ľubovoľnej dĺžky, ktorých začiatok je odkazovaný premennou „`x`", by vyzeral nasledovne: *(+ znamená „alebo")*

```
typedef struct Tinit
/*? x ?*/
{
    struct T2 * next ;
    struct T2 * back ; /*? null ?*/
};
typedef struct T2 {
    struct T2 * next ; /*? null ?*/
    struct Any * back ; /*? -next ?*/
} + {
    struct T2 * next ;
    struct Any * back ; /*? -next ?*/
};
```

2.2.2 program.py

Jedná sa o súbor obsahujúci jednu funkciu v jazyku Python, ktorá vracia dvojicu `(program, env)`. Prvý prvok `program` je zoznam n-tíc, kde každá n-tica je jeden príkaz jazyka ARTMC. V tabuľke 2.1 sú popísané jednotlivé podporované konštrukcie.

konštrukcia	zápis pre ARTMC
x := NULL	(“x=null”,“line_num”,x,next_line)
x := y	(“x=y”,“line_num”,x,y,next_line)
x := y.next	(“x=y.next”,“line_num”,x,y,next,next_line)
x.next = y	(“x.next=y”,“line_num”,x,y,next,next_line,descr_num)
if (x == NULL)	(“ifx==null”,“line_num”,x,next_line_then,next_line_else)
if (x == y)	(“ifx==y”,“line_num”,x,y,next_line_then,next_line_else)
if (*)	(“if*”,“line_num”,next_line_then,next_line_else)
goto	(“goto”,“line_num”,next_line)
exit	(“exit”,“line_num”)
x.next = NULL	(“x.next=null”,“line_num”,x,next,next_line)
x.next = new	(“x.next=new”,“line_num”,x,next,next_line, descr_num, gen_descr)
setdata	(“setdata”,“line_num”,x,“data”,next_line)
if (x.data == “. . . ”)	(“ifdata”,“line_num”,x,“data”,next_line_then,next_line_else)
x := random_pos	(“x=random”,“line_num”,x,next_line)
new	(“new”,“line_num”,x,next_line)

Tabuľka 2.1: Podporované príkazy v ARTMC

2.2.3 Popis príkazov jazyka pre ARTMC

- *Identifikátor* - Prvá položka je vždy reťazec obsahujúci identifikátor inštrukcie.
- *Číslo príkazu* - Druhá položka označená ako „`line_num`“ identifikuje príkaz v rámci celého programu. Jedná sa o reťazec skladajúci sa z postupnosti núl a jednotiek, ktorý musí byť v rámci celého programu jedinečný. Najvhodnejší postup je číslovať inštrukcie od 0 až po N a previesť tieto čísla do binárnej podoby. Zároveň je dôležité, aby všetky výsledné binárne čísla mali rovnaký počet znakov.
- *Premenné* - Označené ako „`x`“ a „`y`“. V programe sú reprezentované prirodzenými číslami začínajúce od 1.
- *Ukazateľové položky* - Označené ako „`next`“. V programe sú reprezentované prirodzenými číslami začínajúc od 0.
- *Ukazateľ na ďalšiu inštrukciu* - Označené ako „`next_line`“. Označuje číslo riadku inštrukcie, na ktorej sa po skončení bude pokračovať. Inštrukcie sú číslované od nuly. Ak je použitá prípona „`_then`“ alebo „`_else`“ jedná sa o miesto pokračovania v prípade splnenia alebo nesplnenia podmienky danej inštrukcie.
- *Deskriptory 1* - Označené ako „`descr_num`“. Jedinečný deskriptor použitý pri vykonávaní danej inštrukcie. Deskriptory sú číslované od 1. Viac o nastavovaní hodnôt tohto deskriptoru je v podsekcii 2.2.5.
- *Deskriptory 2* - Označené ako „`gen_descr`“. Využíva sa iba pri príkaze `x.next=new` a označuje automatickú generáciu počiatočného deskriptoru. Nadobúda hodnoty 0 (vypnuté) a 1 (zapnuté). Doporučené nastavenie je 1. Vypnutie sa odporúča iba pokročilým užívateľom so znalosťou práce s ARTMC.

2.2.4 env

Jedná sa o 6ticu (`node_width, pointer_num, descr_num, next_num, err_line, restrict_var`) obsahujúcu informácie o programe. Význam jednotlivých položiek je nasledovný:

- `node_width = pointer_num + descr_num + 2 +` šírka dát
- `pointer_num` = Počet ukazateľových premenných zvýšený o 1
- `desc_num` = Počet ukazateľových deskriptorov zvýšený o 1 (viď podsekciu 2.2.5)
- `next_num` = Počet next pointerov použitých v programe
- `err_line` = Označenie riadku, na ktorý sa skáče v prípade chyby
- `restrict_var` = Automatické generovanie automatu, ktorý garantuje iba jeden výskyt programovej premennej v konfigurácii. Nadobúda hodnoty 0 (vypnuté) a 1 (zapnuté). Doporučené nastavenie je 1.

2.2.5 Nastavovanie deskriptoru

Deskriptor pozostáva z dvoch častí:

1. *Deskriptory v počiatočnej konfigurácii* - Jedná sa o deskriptory, ktoré sú použité v súbore *typedefs*. Tieto deskriptory už nemôžu byť znova použité. Číslo najvyššieho použitého deskriptoru je možné získať zo skriptu `get_typedef_descr.t.sh`, ktorý je súčasťou distribúcie ARTMC.
2. *Vlastné deskriptory* - Obvyklý postup je ku každému príkazu typu `x.next=y` alebo `x.next=new` priradiť vlastný deskriptor, ktorý ešte nebol použitý. Nemôže byť použitý ani v konfigurácii[1].

2.2.6 Podpora práce s dátami v ARTMC

Jazyk pre nástroj ARTMC obsahuje iba dve inštrukcie pre prácu s dátami. Jedná sa o príkazy `setdata` a `ifdata`. Existujú teda iba funkcie na priradenie hodnoty a na porovnanie hodnoty. Ako si je možné v tabuľke 2.1 všimnúť, tieto funkcie akceptujú iba jednú premennú. Je to z dôvodu, že nástroj ARTMC predpokladá, že štruktúra, ktorého typu je daná premenná, obsahuje práve jednu dátovú položku. V jazyku C môžu tieto dáta byť ľubovoľného typu, avšak pre ARTMC musia byť kódované ako postupnosť núl a jednotiek. Na skutočnej reprezentácii v binárnej podobe nezáleží, takže je vhodné prvým dátam dať napríklad postupnosť “00000001” ďalším “00000010” atď. Avšak je potrebné, aby jedna hodnota nebola kódovaná viackrát pod rôznymi kódmi.

Problematické je však spracovanie zložitejších výrazov, nakoľko neexistujú funkcie na sčítanie, odčítanie a pod. Napríklad výraz

```
if (x->data - 1 > y->data)
```

sa nedá zapísať do nástroja ARTMC. Je možná určitá analýza a za pomoci sledovania šírenia konštánt alebo úpravy výrazov je možné niektoré konštrukcie preložiť.

[1] Iný postup ako tento môže viesť k nesprávnemu fungovaniu nástroja ARTMC a teda sa odporúča len pokročilým užívateľom.

Ako príklad uveďme výraz:

```
if (x->data -1 == 0)
```

po úprave výrazu je ho možné prepísať ako:

```
if (x->data == 1)
```

A napríklad kód:

```
y->data = 0
...ak sa nezmení y a ani y->data
if (x->data == y->data)
    ...
```

po sledovaní šírenia konštánt je ho možné zapísať ako:

```
y->data = 0
...
if (x->data == 0)
    ...
```

Avšak takéto úpravy nie sú vždy možné, hlavne ak sa menia hodnoty vo while-cykloch. Potom je možná len „ignorácia“ práce s dátami. Pri takomto postupe sa neuvažujú žiadne inštrukcie kde sa nastavuje hodnota dát a všetky podmienky, ktoré porovnávajú dáta, sú nahradené inštrukciou `if*`. Tento prístup ale môže viesť k falošným protipríkladom (tvrdenie, že program obsahuje chybu, ak ju v skutočnosti neobsahuje), ale nespôsobí falošné pozitíva (tvrdenie, že neobsahuje chybu, ak ju v skutočnosti obsahuje).

2.3 Tvorba programu v ARTMC

Tvorba programu pre nástroj ARTMC je zložitá, ľahko sa v ňom tvoria chyby a je komplikované upravovať napísaný program. Ako príklad uveďme jednoduchý program, ktorý by v jazyku C mohol vyzerať nasledovne:

```
while(x->next != NULL)
    x = x->next;
```

Takýto program by sa do nástroja ARTMC prepísal ako:

```
("x=y.next","00000000",2,1,0,1),
("ifx==null","00000001",2,4,2),
("x=y.next","00000010",1,1,0,3),
("goto","00000011",0)
```

Okrem tohoto kódu je potrebné napísať celú funkciu, ktorá vracia tento kód a dané prostredie. Celý vstupný súbor by teda vyzeral nasledovne.

```
def get_program():
    program=[
        ("x=y.next","00000000",2,1,0,1),
        ("ifx==null","00000001",2,4,2),
        ("x=y.next","00000010",1,1,0,3),
        ("goto","00000011",0),
        ("exit","00000100")] # Na konci musí byť inštrukcia exit
    node_width=16
```

```
    pointer_num=3
    desc_num=3
    next_num=1
    err_line="11111111"
    restrict_var=1
    env=(node_width, pointer_num, desc_num, next_num, err_line,restrict_var)
    return(program, env)
```

Ako je na prvý pohľad zrejmé, napísať program v jazyku C je výrazne jednoduchšie ako pre nástroj ARTMC. Rovnako je v jazyku C výrazne menšia šanca zavedenia chyby. Pri tvorbe programu v nástroji ARTMC je potrebné počítať premenné a správne ich značiť, počítať riadky pre pokračovanie inštrukcií atď. Ešte zložitejšia je úprava kódu, kde len pridaním zmeny je častokrát potrebné prepísať takmer celý program. Predstavme si, že sa rozhodneme mierne zmeniť predchádzajúci program na:

```
while(x->next != NULL){
    x->next = malloc(sizeof(x));
    x = x->next;
}
```

výstup by sa výrazne zmenil. Podčiarknuté sú všetky riadky, ktoré sa zmenili alebo pribudli.

```
def get_program():
    program=[
        ("x=y.next","00000000",2,1,0,1),
        ("ifx==null","00000001",2,5,2),
        ("x.next=new","00000010",1,0,3,3,1),
        ("x=y.next","00000011",1,1,0,4),
        ("goto","00000100",0),
        ("exit","00000101")]
    node_width=17
    pointer_num=3
    desc_num=4
    next_num=1
    err_line="11111111"
    restrict_var=1
    env=(node_width, pointer_num, desc_num, next_num, err_line,restrict_var)
    return(program, env)
```

Kapitola 3

Prekladače

Prekladač je nástroj, ktorý číta zdrojový program a prekladá ho na cieľový program. Zdrojový a cieľový program sú si vzájomne funkčne ekvivalentné. [4] Prekladač pozostáva z viacerých častí. Schéma bežného prekladača je na obrázku 3.1.

3.1 Lexikálny analyzátor

Lexikálny analyzátor, scanner, má na vstupe program zapísaný v zdrojovom jazyku. Tento program je scannerom delený na lexémy - logicky oddelené lexikálne jednotky. Na výstupe je reťazec tokenov, pričom token reprezentuje lexémy a môže obsahovať ďalšie atribúty. [4] Tento proces je znázornený na obrázku 3.2.

3.2 Syntaktický analyzátor

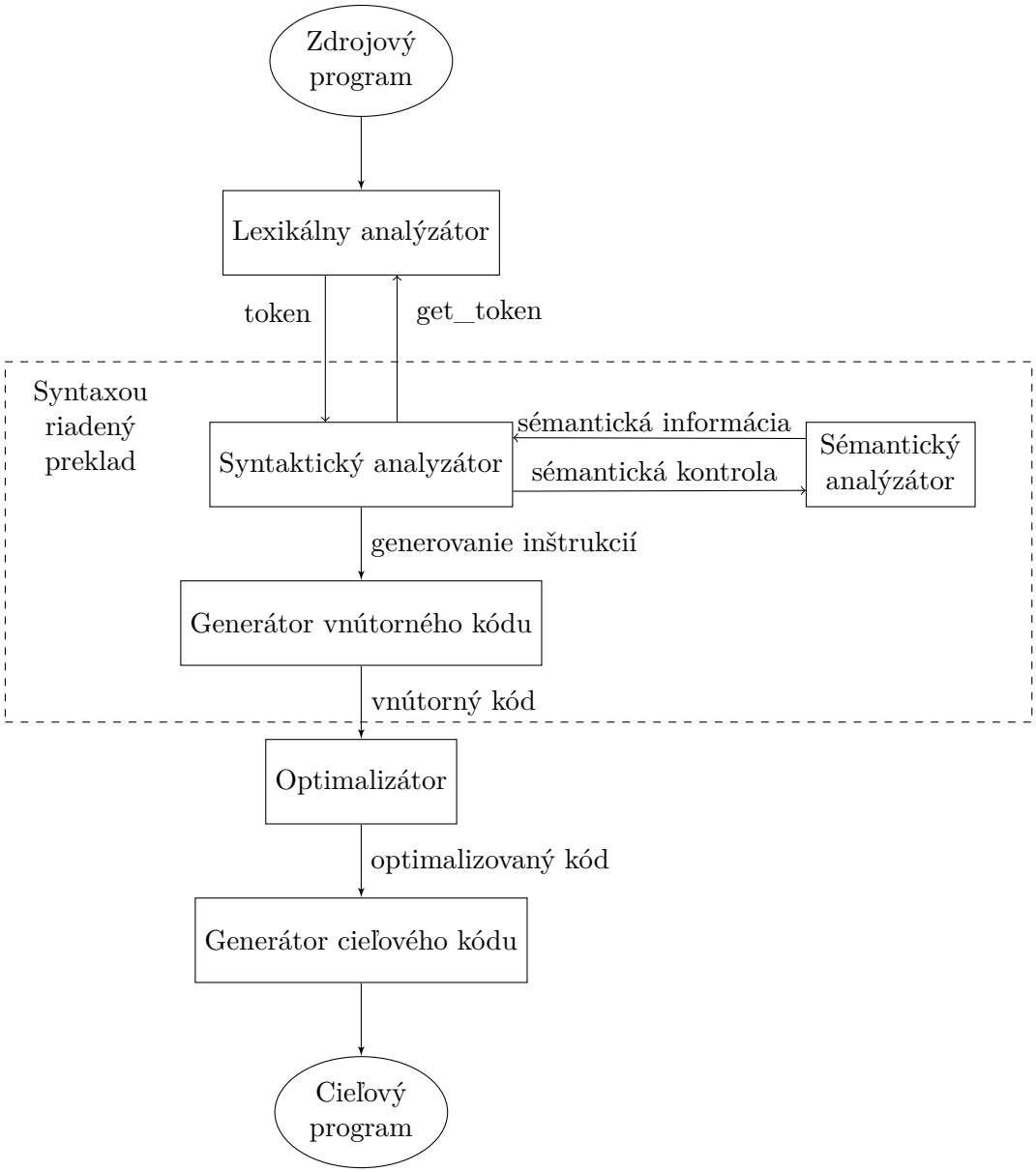
Syntaktický analyzátor, parser, má na vstupe reťazec tokenov, ktoré získal z lexikálneho analyzátora. Úlohou je skontrolovať, či reťazec tokenov reprezentuje syntakticky správne napísaný program. Táto kontrola prebieha za pomoci konštrukcie derivačného stromu. Ak je možné zostrojiť derivačný strom, jedná sa o program validný. Konštrukcia môže prebiehať dvoma spôsobmi a to zhora nadol a zdola nahor. [4]

3.2.1 Analýza zhora-nadol

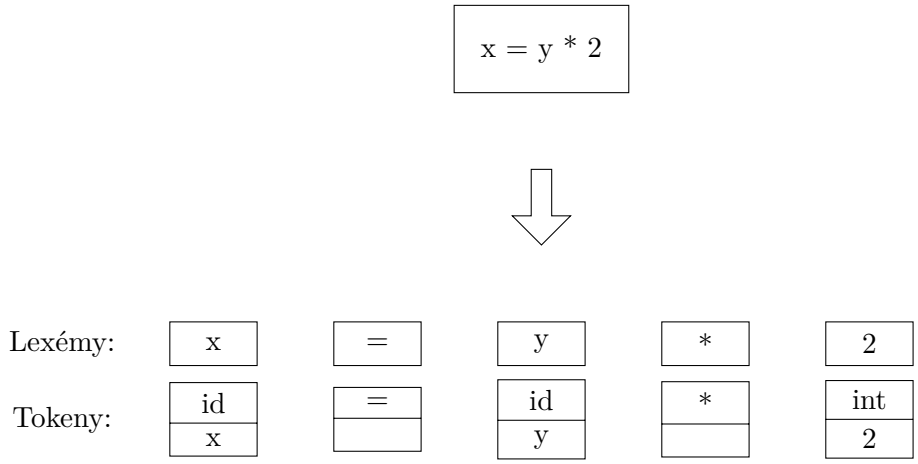
Metóda zhora-nadol sa snaží skonštruovať derivačný strom od najvyššej úrovne a postupne prechádza derivačný strom smerom dole za využita formálnych pravidiel gramatiky. To znamená, že derivačný strom sa konštruuje od koreňa smerom k listom, zľava doprava, podľa ľavej derivácie. Syntaktická analýza zhora-nadol využíva pre svoju prácu LL syntaktické analyzátory. [1]

3.2.2 Analýza zdola-nahor

Metóda zdola-nahor sa snaží skonštruovať derivačný strom od najnižšej úrovne a postupne prechádza derivačný strom smerom ku koreňu za využitia formálnych pravidiel gramatiky. To znamená, že sa najskôr identifikujú jednotlivé symboly, z ktorých sa následne vytvára derivačný strom. Táto metóda využíva precedenčnú tabuľku. [1]



Obr. 3.1: Schéma prekladača [4]



Obr. 3.2: Schéma rozdeľovania na lexémy a tokeny

3.3 Sémantický analyzátor

Na vstupe sémantického analyzátoru je derivačný strom, ktorý bol získaný od syntaktického analyzátora. Sémantický analyzátor kontroluje sémantické aspekty vstupného programu a to hlavne

- kontrola typov
- kontrola deklarácií premenných

Na výstupe je abstraktný syntaktický strom.

3.4 Generátor vnútorného kódu

Generátor vnútorného kódu, ako názov napovedá, vytvára vnútornú reprezentáciu programu z abstraktného syntaktického stromu. Medzi hlavné dôvody, prečo je vhodné vytvárať vnútorný kód patrí hlavne:

- Priame vytváranie výstupného programu je nepriehľadné
- Jednotnosť - všetky inštrukcie sa tvária rovnako
- Jednoduchá optimalizácia

3.5 Optimalizátor

Cieľom tejto časti je upraviť vnútorný kód to efektívnejšej podoby. Medzi optimalizácie môžu patriť:

- Eliminácia mŕtveho kódu
- Šírenie kopírovaním
- Šírenie konštanty

3.6 Generátor cieľového kódu

Generátor cieľového kódu prevádza vnútornú reprezentáciu programu do funkčne ekvivalentného cieľového programu a teda ukončuje proces kompilácie. Tento posledný krok je závislý od cieľového jazyka, ale aj konkrétnej platformy, pre ktorú sa preklad vykonáva. Najčastejšie je cieľovým jazykom strojový kód alebo assambler, no je možný preklad aj medzi dvoma vyššími programovacími jazykmi. [4]

3.7 Syntaxou riadený preklad

Jedná sa o prístup, v ktorom je celý preklad riadený procesom parsovania. Teda ku každému pravidlu gramatiky sú priradené ďalšie akcie, ktoré pokrývajú ostatné fázy prekladu. Medzi akcie môže napríklad patriť:

- Vyvolanie sémantickej kontroly
- Ukladanie informácií napr. do tabuľky symbolov alebo tabuľky návestí
- Generovanie kódu

Prevažne pri využití tejto techniky postačuje jeden priechod cez zdrojový programi. [1]

Kapitola 4

Súčasný stav prekladača pre ARTMC

Pre nástroj ARTMC bol vytvorený prekladač z jazyka C v rámci projektu AVERILES[1]. Jedná sa o študentský projekt napísaný v jazyku Java. Snahou tejto práce nie je tento prekladač upraviť, ale napísať úplne nový. Hlavné dôvody prečo je potrebné vytvoriť nový prekladač sú:

- Chyby v prekladači
  Prekladač nedokáže spracovať určité výrazy a niektoré výrazy sú prekladané nesprávne. Preto je potrebné pri zložitejších programoch manuálne kontrolovať výsledný program a editovať vstupný program do takej formy, aby ho prekladač dokázal preložiť.

- Ťažko rozšíriteľný
  Prekladač je napísaný v jazyku Java zložitým a nečitateľným spôsobom. Takýto kód sa veľmi ťažko modifikuje a rozširuje o ďalšie vlastnosti. Prekladač, ktorý je výstupom tejto práce si dáva za cieľ čitateľný kód, do ktorého sa jednoducho dopĺňajú ďalšie funkcionality.

- Nepraktické používanie
  Proces prekladu za pomoci AVERILES pozostáva z dvoch krokov:

  1. Tvorba XML súboru zo zdrojového programu v jazyku C
  2. Preklad XML súboru do vstupného formátu nástroja ARTMC

  Takýto postup je zložitejší, nakoľko je potrebné volať dva programy. Zároveň programy nemajú prepínače na modifikáciu ich správania, teda použitie je možné len v jednom užívateľskom prípade.

[1] http://www.lsv.fr/Projects/rntl-averiles/

Kapitola 5

Návrh implementácie

Táto kapitola sa venuje návrhu implementácie samotného prekladača. Ako implementačný jazyk bol zvolený Python. Kritériá, ktoré by mala výsledná aplikácia spĺňať sa dajú rozdeliť do troch kategórií:

1. Použiteľnosť a prenositeľnosť
2. Udržiavateľnosť a rozširovateľnosť
3. Testovateľnosť

5.1 Použiteľnosť a prenositeľnosť

Výsledná aplikácia by mala byť jednoducho použiteľná. Ako bolo spomenuté v kapitole 4, momentálne existujúci prekladač nemá jednoduché používanie. Zároveň neumožňuje špecifikovať ďalšie voľby, ktoré by užívateľ mohol potrebovať. Medzi nich, okrem iných, patria hlavne:

- Možnosť špecifikovať výstupný súbor
  Výstupný súbor je implicitne uložený na rovnaké miesto ako vstupný a je pomenovaný `program.py`. Užívateľ si avšak môže zvoliť ľubovoľné umiestenie a názov súboru.
- Možnosť špecifikovať vstupný deskriptor
  Viď kapitola 2.2.5. Vstupný deskriptor sa získava z `get_typedef_descr.t.sh` a je možné, že tento program nie je dostupný. Aby sa dal preklad spustiť, môže užívateľ túto hodnotu poskytnúť.
- Možnosť ignorácie dát
  Nakoľko podpora práce s dátami v ARTMC je výrazne obmedzená, môže byť vhodné nechať prekladač, aby prácu s dátami neprekladal.Výsledok prekladu je potom avšak nadaproximáciou vstupu.

Medzi hlavné požiadavky rovnako patrí aj prenositeľnosť. V rámci tohto bodu musíme uvažovať dva druhy prenositeľnosti:

- Operačný systém
  Program by mal byť spustiteľný a správne fungovať minimálne na operačných systémoch Linux a Windows.

- Verzia Pythonu
  Program by mal byť spustiteľný a správne fungovať na všetkých hlavných verziách jazyka Python. Viac o Pythone a verziách je v kapitole 5.1.1.

5.1.1 Python

Python je vysokoúrovňový skriptovací programovací jazyk. Podporuje viacero programovacích paradigiem. Pre túto prácu bude použité objektovo orientované paradigma. Zároveň Python podporuje dynamickú typovú kontrolu. Python má dve hlavné verzie – Python 2.X a Python 3.X. Tieto verzie sa ďalej delia na podverzie. Verzie a podverzie nemusia byť (a vo väčšine prípadov nie sú) medzi sebou kompatibilné. To prináša problém pre programátorov, ktorí chcú písať program tak, aby fungoval pre rôzne verzie. C2ARTMC by mal fungovať aspoň pre verzie `2.6, 2.7, 3.3, 3.4 ah 3.5`.

5.2 Udržiavateľnosť a rozšírovateľnosť

Program by mal byť napísaný tak, aby bolo neskôr možné dopĺňať ďalšiu funkcionalitu a prípadne opravovať chyby. Mal by byť logicky rozdelený na menšie celky (moduly, triedy, funkcie). Očakáva sa dostatočná miera komentárov, aspoň pri každom celku. Tieto vlastnosti by mali umožniť jednoduchú orientáciu v kóde a teda schopnosť neskôr tento prekladač upravovať.

Okrem vyššie spomenutých vlastností by mal byť program verziovaný za pomoci nástroja Git, ktorý umožňuje spoluprácu viacerých autorov ako aj sledovanie a vyhľadávanie zmien v celej histórii projektu.

5.2.1 Git

Verziovací systém Git umožňuje sledovať vývoj projektu a v histórií jednotlivých zmenách sa posúvať. Teda v prípade chyby je jednoduché zistiť kedy a ako chyba vznikla. Zároveň je jednoduché sa vrátiť do stavu pred chybou. Git umožňuje rovnako jednoduchú spoluprácu viacerých autorov. Táto práca je dielom jedného autora, je však možné, že po dokončení práce sa nájde chyba, alebo bude treba doplniť funkcionalitu a to umožní, aby iný autor túto zmenu vykonal.[1]

5.3 Testovateľnosť

Vývoj väčších aplikácii si vyžaduje overovanie správnej funkcionality programu. Overovanie sa vykonáva za pomoci hromadných testov, kde jedným príkazom je vyvolaných viacero testov a ako výsledok je poskytnuté zhrnutie zo všetkých testov. Táto práca by takéto testy mala obsahovať a za využitia nástroja Tox by mala overovať správne fungovania aj na rôznych verziách Pythonu.

5.3.1 Hromadné testy

Pri pridávaní novej funkcionality sa môže stať, že sa zavedie chyba v rámci inej funkcionality. Preto je vhodné po každej zmene otestovať všetky predtým naprogramované vlastnosti. To je prirodzene takmer nemožné. Preto sa píšu testy, ktoré sa dajú spúšťať hromadne a je

[1] Táto práca využíva GitHub a je dostupná na `https://github.com/marusak/C2ARTMC`.

ľahko a rýchlo možné overiť, že neprišlo k zavedeniu chyby. Rovnako je odporúčané na každú novú funkcionalitu napísať test, ktorý overí, že táto vlastnosť bola správne implementovaná. Neskôr tento test bude overovať, že táto funkcia stále funguje správne a nebola narušená.

5.3.2 Tox

Aby bolo možné spúšťať testy pod rôznymi verziami Pythonu a overiť, že všetky testy uspejú na všetkých podporovaných verziách, v práci bude použitý nástroj Tox. Jedná sa o program, ktorý potrebuje iba jeden jednoduchý konfiguračný súbor, ktorý obsahuje spôsob ako sa spúšťajú testy a verzie Pythonov, pod ktorými chceme aby program fungoval. Následne zavolaním programu Tox sa testy spustia pod všetkými požadovanými verziami a prehľadný výpis oznámi, či všetky testy boli úspešné, prípadne ktorý test neuspel v ktorej verzii.

Kapitola 6

Výsledná aplikácia

Aplikácia bola vytvorená s ohľadom na zadanie bakalárskej práce. Boli dodržané všetky body zadania. Pri vývoji boli dodržané vlastnosti kódu a vývoja spomenuté v kapitole 5. Hneď po vytvorení prvej skutočne funkčnej časti kódu boli vytvárané automatické testy pre overenie funkcionality implementovaných častí a overenie, že počas pridávania ďalších vlastností neprišlo k regresii v inej časti programu.

Kód bol od začiatku publikovaný na GitHub-e s ohľadom na odporúčané praktiky pri využívaní verzovacieho systému Git. História je prehľadná a je možné jednoducho sa v nej vracať alebo zisťovať, kedy a prečo, ktorá časť kódu bola implementovaná.

Výsledný program bol dôkladne otestovaný na sade testov. Pre zoznam testov viď prílohu B. Všetky testy boli úspešné na všetkých požadovaných operačných systémoch ako aj verziách Pythonu.

6.1 Výsledná implementácia

V tejto časti textu je popis implementácie. Jedná sa o spojenie znalostí z kapitoly 3 o prekladačoch so skutočne použitými vlastnosťami a ich implementáciou v jazyku Python.

6.1.1 Lexikálny analyzátor

Ako prvá časť bol implementovaný lexikálny analyzátor. Jedná sa o samotnú triedu `Scanner`, ktorá v inicializácii berie názov súboru z programom v jazyku C. Následne je vstupný program načítaný a predspracovaný. Jedná sa hlavne o odstránenie komentárov a nahradenie skupín bielych znakov jednou medzerou. Tieto úpravy následne umožňujú jednoduchšie delenie kódu na tokeny. Trieda lexikálneho analyzátora poskytuje metódu `get_token`, ktorá vráti práve jeden token. Na delenie sa nevyužívajú regulárne výrazy, ale podľa prvého neprečítaného znaku sa scanner rozhoduje o aký token sa môže jednať. Následne po rozpoznaní tokenu, je vrátený token, ktorý je reprezentovaný jedným číslom - jedná sa o imitáciu `enum` pre jazyk Python. Konkrétna hodnota tokenu sa nevracia priamo a je možné ju získať volaním funkcie `get_value`. Scanner poskytuje aj metódu `unget_token`, ktorá vráti jeden token späť do vstupného reťazca. Nakoľko je ale scanner ako aj celý prekladač jednopriechodový, je táto funkcia implementovaná inštančnou premennou, ktorá drží vrátenú hodnotu. Výhodou je jednoduchá implementácia, teoretickou nevýhodou je schopnosť vrátiť iba jeden token, no v rámci implementácie tohto prekladača nebolo nikdy potrebné vrátiť viac za sebou idúcich tokenov.

6.1.2 Syntaktický analyzátor

Následne bol tvorený syntaktický analyzátor, ktorý je implementovaný ako trieda `Parser`. Jedná sa o syntaxou riadený preklad a preto je samotný analyzátor najdlhšia časť kódu. Napriek svojej veľkosti sa jedná o pomerne jednoduchý program. V princípe je celá analýza jeden while-cyklus, ktorý čaká až skončí vstupný súbor a `scanner` vráti `EOF`. V tomto cykle sa prečíta prvý token a na základe neho sa rozhodne o aký príkaz sa jedná. V tejto úrovni sa môže jednať iba o definíciu alebo deklaráciu funkcie alebo premennej. Spracovanie premennej je priamočiara záležitosť. Spracovanie funkcie obsluhuje funkcia `parse_function`, ktorá je podobne jeden while-cyklus, ktorý volá funkciu `parse_command` až pokiaľ nenarazí na ukončujúcu zloženú zátvorku. Je podporované spracovanie iba jednej funkcie a to prvej v súbore. Funkcia `parse_command` zavolá na základe prvého tokenu funkcie na spracovanie príkazu, napríklad `parse_if` alebo `parse_assignment`. Ak sa jedná o príkaz, ktorý vytvára ďalšiu úroveň vnorenia ako napríklad `if` alebo `while`, celá úroveň je spracovaná touto funkciou volania `parse_command` na spracovanie jednotlivých príkazov. Hovoríme teda o rekurzívnom vnáraní.

Ako bolo spomenuté v sekcii 3.7 ohľadom syntaxou riadenom preklade, počas syntaktickej analýzy sa vykonávajú aj ďalšie akcie. Medzi takéto akcie patrí napríklad vloženie premennej do tabuľky symbolov po nájdení novej premennej. Tabuľku symbolov vlastní trieda `Parser` a poskytuje sadu inštrukcií na prácu s ňou. Medzi ďalšie akcie patrí vygenerovanie inštrukcie vo vnútornom kóde po prečítaní celého príkazu.

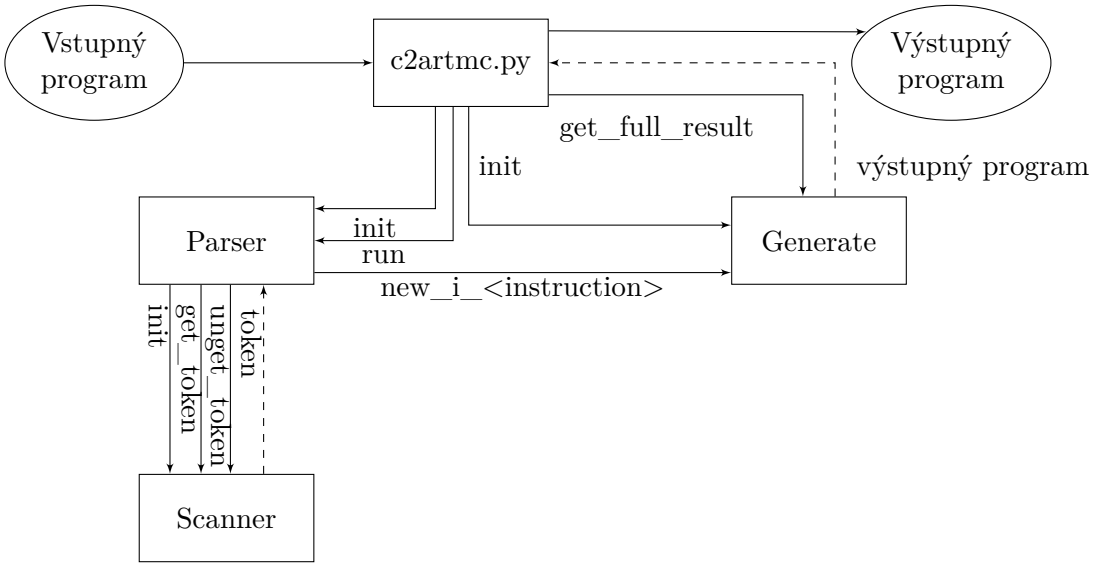
6.1.3 Sémantický analyzátor

Sémantický analyzátor nie je implicitne oddelený od zvyšku kódu ako je napríklad lexikálny alebo syntaktický analyzátor. Sémantická analýza sa vykonáva len v obmedzenej forme, a preto boli všetky potrebné kontroly zahrnuté priamo do parsera. Hlavný dôvod pre nízke množstvo kontrol súvisí s nepodporovaním dát v jazyku ARTMC a teda takmer všetky inštrukcie, kde je potrebná sémantická analýza sa nevyskytujú vo vstupnom súbore. Samozrejmosťou je ošetrenie správania sa v prípade výskytu takejto konštrukcie. Toto správanie je opísané v sekcii 6.5.

6.1.4 Generovanie kódu

Na spracovanie vnútorného kódu bola vytvorená trieda `Generate`. Ako názov napovedá, táto trieda nielen prijíma inštrukcie od parseru inštrukcie vo vnútornej reprezentácií, ale aj generuje cieľový kód. Trieda poskytuje celý rad funkcií pre vkladanie inštrukcií, ktoré začínajú prefixom `new_i_`. Okrem toho obsahuje funkciu `get_full_result`, ktorá má za úlohu vrátiť kód vo výstupnom jazyku pripravený na výpis alebo zápis do súboru. Táto funkcia volá všetky potrebné funkcie na generovanie a dokončovanie inštrukcií z vnútorného kódu do výsledného kódu.

Schéma fungovania výslednej implementácie je znázornená na obrázku 6.1.



Obr. 6.1: Schéma fungovania implementovaného prekladača

6.2 Vstupný program

Na vstupe sa očakáva platný program v jazyku C s určitými výnimkami. Výnimky môžme rozdeliť na štyri skupiny:

1. Nepodporované konštrukcie
2. Neštandardné podporované konštrukcie
3. Povinné položky
4. Nepovinné položky

6.2.1 Nepodporované konštrukcie

Ako bolo spomenuté v kapitole 2, nástroj ARTMC má obmedzenú sadu príkazov a preto nie je možné preložiť ľubovoľné konštrukcie. Preto bol prekladač navrhnutý tak, aby nebolo nutné program príliš upravovať. Je to dosiahnuté tým, že príkazy, ktoré nemenia používané premenné sú preskočené. Viac o tejto vlastnosti v podsekcii 6.5.1.

Z významných konštrukcií nie sú podporované:

- `enum`
- `for`
- `union`

Medzi významné nepodporované konštrukcie patrí aj obmedzenie práce s dátami. Tomuto obmedzeniu sa venuje sekcia 6.3.

6.2.2 Neštandardné podporované konštrukcie

Okrem validných konštrukcií pre jazyk C boli implementované aj rozšírenia, ktoré sú potrebné pre nástroj ARTMC. Jedná sa hlavne o podporu nedeterministického rozhodovania. V jazyku C je túto konštrukciu možné zapísať ako `if(*)` alebo `if(any)`. Obe tieto konštrukcie sú preložené na príkaz `if*`.

Bola implementovaná aj podpora pre kľúčové slovo `ERROR`, ktoré sa môže vyskytnúť v rámci príkazu `return`. Tento príkaz sa preloží na príkaz `goto`. Ako miesto skoku je riadok vyskytujúci sa v prostredí `env err_line`. Príkaz `return ERROR` sa môže chápať ako ukončenie s chybou, napríklad po neúspešnom alokovaní pamäte.

Posledná neštandardná vlastnosť pre implementovaný prekladač je podpora funkcie `random_alloc(void)`. Táto funkcia sa preloží na príkaz `random_position`. Využitím tejto funkcie je možné prinútiť nástroj ARTMC vybrať náhodný alokovaný uzol.

6.2.3 Povinné položky

Nástroj ARTMC implicitne predpokladá existenciu špecifickej štruktúry. Avšak počas návrhu prekladača bolo rozhodnuté, že prekladač nebude implicitne predpokladať existenciu žiadnej dátovej štruktúry a bude vyžadovať, aby vždy bola presne zadaná pre každý program. Preto vstupný program musí obsahovať presne definovanú štruktúru. Príkladom môže byť štruktúra v schéme 6.2. Za povšimnutie stojí existencia iba jednej neukazateľovej položky. Viac ako jedna dátová položka totižto nie je podporovaná zo strany nástroja ARTMC a teda takúto štruktúru prekladač odmietne.

```
typedef struct T1 {
    struct T1* next;
    struct T1* prev;
    int data;
}* T;
```

Obr. 6.2: Vzorový príklad štruktúry vo vstupnom programe

6.2.4 Nepovinné položky

Prekladač nepodporuje direktívu preprocesoru `#include`. Avšak takéto riadky sú pri preklade ignorované, takže je možné ich tam ponechať. Preto je riadok `#include <xxxxxx.h>` nepovinný aj napriek tomu, že v zdrojovom kóde sa môžu vyskytovať kľúčové slová a funkcie ktoré sú touto knižnicou podporované. V tabuľke 6.1 je výpis knižníc a príkazov, ktoré z nich sú podporované.

Knižnica	**Kľúčové slová a funkcie**
stdlib.h	NULL, malloc
assert.h	assert
stdbool.h	true, false

Tabuľka 6.1: Implicitne podporované príkazy

6.3 Práca s dátami

Vo vstupnom súbore nie sú podporované operácie nad dátovými premennými okrem priradenia a porovnania. Je možné len priame priradenie a nie je možné hodnotu meniť aritmetickými a reťazcovými operáciami. Na takúto situáciu prekladač upozorní. Viď tabuľku 6.2 pre lepšie porozumenie. Avšak je možné preložiť aj takýto program za pomoci ignorovania práce s dátami.

Príkaz	**Podpora v prekladači**
int x;	OK
int x = 5;	OK
x = 9;	OK
x = y;	OK
x = 14*3;	ERROR
x += 2;	ERROR

Tabuľka 6.2: Možnosti práce s dátami

6.3.1 Ignorovanie dát

Počas vývoja a testovania prekladača sa vyskytol problém pri práci s dátami. Na vstupe sa objavil validný program, ktorý by mohol byť korektne preložený, avšak kvôli prítomnosti príkazu manipulujúcemu s dátovou premennou nemohol byť tento program preložený. V mnohých prípadoch sa jednalo o prácu s premennou, ktorá nebola potrebná pre analýzu nástrojom ARTMC. Ako príklad uvádzam nasledujúci program:

```
x->data = 1;
...
numerické operácie s hodnotou x->data, napr. „x->data += 1;"
...
if (x->data < 0)
    x->next = y;
```

V hore uvedenom programe sa manipuluje s dátovou položkou štruktúry. Takéto operácie nie sú podporované zo strany nástroja ARTMC a preto by tento program nemohol byť preložený. Ak však používateľ prekladača uzná, že tieto manipulácie a podmienka nie sú kritickým rozhodovacím miestom a teda vynechanie zmien dátovej položky a nahradenie podmienky za nedeterministickú je možné, môže použiť prepínač `-i`, ktorý vynúti neprekladanie všetkých operácií, ktoré manipulujú s dátami a všetky podmienky obsahujúce porovnávanie dátové položky sa zmenia na nedeterministickú podmienku `if*`.

6.3.2 Využitie štandardných premenných

Okrem podpory priameho priraďovania dát do dátovej položky štruktúry je možné použiť priradenie aj zo štandardnej dátovej položky. Príklad použitia je znázornený na nasledujúcom kuse kódu:

```
int i = 0;
x->data = i;
```

Podobne je možné použiť štandardnú dátovú premennú aj v podmienke, ako je vidieť v nasledujúcej ukážke:

```
int i = 0;
x->data = 1;
if (x->data == i)
    return ERROR;
```

Avšak rovnako ako pri práci s dátami priamo, ani v tomto prípade nie je možné numericky a reťazcovo meniť hodnotu premennej (viď tabuľku 6.2).

6.4 Použitie aplikácie

Prekladač poskytuje viacero prepínačov, ktorými je možné upraviť implicitné správanie. Nasleduje výpis `c2artmc.py --help`, ktorý ukazuje možné použitie tohto nástroja.

```
usage: c2artmc.py [-h] [-o O] [-d D] [-i] INPUT_FILE

Converter from C to ARTMC.

positional arguments:
 INPUT_FILE  C source file to be converted

optional arguments:
  -h, --help  show this help message and exit
  -o O~Filepath to write the ARTMC file
  -d D        Initial pointer descriptor
  -i          Ignore data
```

6.5 Neštandardné vlastnosti

V tejto kapitole sú popísané vybrané vlastnosti prekladača, ktoré sú nad rámec zadania a implementujú užitočné vlastnosti.

6.5.1 Preskakovanie (častí) príkazov

Nakoľko nástroj ARTMC neobsahuje žiadnu podporu pre volanie funkcií, je nutná schopnosť adekvátne zareagovať na výskyt volania funkcie. Riešením je ohlásiť chybu pri nájdení volania funkcie. Počas testovania sa však ukázalo, že výhodnejšie je priamo takéto príkazy preskakovať. Preto bol na toto správanie prekladač upravený. Momentálne prekladač upozorní na fakt, že sa volanie funkcie preskakuje. Očakáva sa od používateľa, že na tieto varovania zareaguje a ak niektorá funkcia menila premenné podstatné pre analýzu, tak tieto príkazy adekvátne upraví.

Prekladač rovnako preskakuje aj časti príkazov, ak dopredu vie, že nasledujúcu informáciu nepotrebuje. Jedná sa napríklad o príkaz `return`. Ak sa za príkazom `return` nevyskytuje kľúčové slovo `ERROR`, tak môže prekladač zvyšok príkazu zahodiť. Podobné správanie je implementované aj pri funkciách `malloc` a `random_alloc`, ktorých argumenty nie sú prekladané.

6.5.2 Preklad viacnásobných pointrov

V prekladači je implementovaná podpora pre preklad viacnásobného prístupu cez ukazateľ, či sa už jedná o zápis alebo čítanie. Je teda možné preložiť príkaz typu:

```
x->next->next = y->prev->next->prev;
```

Nakoľko nástroj ARTMC nemá podporu pre takýto prístup, je nutné príkaz rozdeliť na viacero jednoduchých príkazov. Vyššie uvedený program by sa teda dal za pomoci pomocných premenných napísať ako:

```
tmp1 = x->next;
tmp2 = y->prev;
tmp3 = tmp2->next;
tmp4 = tmp3->prev;
tmp1->next=tmp4;
```

čo už je možné preložiť aj do nástroja ARTMC.

6.5.3 Vyhodnocovanie neúplných podmienok

V jazyku C sa ako nepravda v podmienke považuje číslica `0` rovnako ako aj ukazateľ `NULL`. Tento fakt sa často využíva pri zápise podmienok, kde podmienka typu `if (x)` je totožná s podmienkou `if (x != NULL)`. Podobne sa to dá využiť aj pri číslach a teda podmienka `if (!x)` je totožná s podmienkou `if (x == 0)`.

Podpora takéhoto zápisu bola implementovaná aj do vytváraného prekladača. Umožňuje to prekladať väčšiu škálu existujúcich programov v jazyku C bez ich upravovania.

Samotná implementácia využíva možnosť vrátenia prečítaného tokenu späť do zoznamu tokenov. Pri spracovaní podmienky kde je očakávané znamienko rovnosti alebo nerovnosti a namiesto toho je nájdená pravá zátvorka (znamená koniec podmienky), logický operátor alebo identifikátor sa tento prečítaný token vráti späť a vygeneruje sa podmienka podľa jej prvej časti.

6.5.4 Skratové vyhodnocovanie podmienok

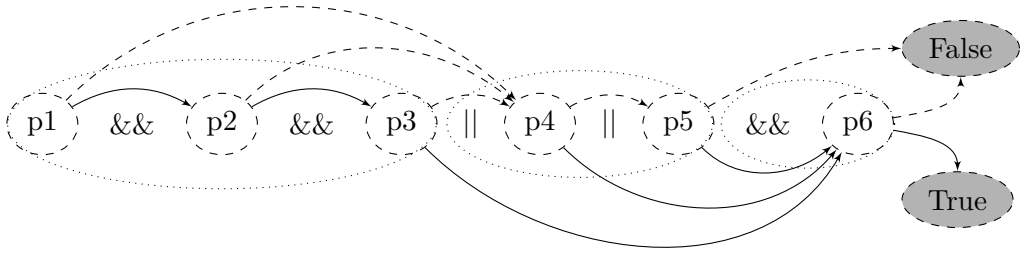
V jazyku C je zaužívané využívať skratové vyhodnocovanie podmienok pre jednoduchší zápis. Veľmi často sa tento prístup používa práve pri dynamicky viazaných dátových štruktúrach. Pre lepšie porozumenie uveďme príklad:

```
if (x && x->next){
    x->next->next = y;
}
```

Pri tomto zápise máme istotu, že nepristúpime alebo nezapíšeme do nealokovaného uzla, nakoľko sa najskôr overí, že je alokované `x` a v prípade úspechu overí aj dostupnosť `x->next`. A iba ak sú oba uzly alokované, môžme bezpečne zapísať aj do `x->next->next`.

Implementovaný prekladač dokáže takéto podmienky preložiť. Je k tomu použitý relatívne jednoduchý algoritmus, ktorý rozdelí výraz na podvýrazy a podvýrazy zhlukuje do skupín podľa logických operátorov. Následne sa prechádza cez skupiny `or`-ov a `and`-ov. Pre skupinu podvýrazov oddelených `||` platí, že po úspechu jednotlivých podvýrazov sa pokračuje za posledným podvýrazom v skupine a pri neúspechu nasledujúcim. Pri skupine spojenej logickým operátorom `&&` je to presne naopak. Tento algoritmus je znázornený

na obrázku 6.3, kde čiarkovane sú označené podvýrazy, bodkovane skupiny, plnou šípkou splnené podmienky a čiarkovanou šípkou nesplnené.



Obr. 6.3: Schéma fungovania prekladu skratového vyhodnocovania

6.6 Testy

Na overenie funkcionality prekladača boli vytvorené testovacie vstupné súbory. Jedná sa o sadu programov v jazyku C, ktoré obsahujú rôzne konštrukcie. Cieľom je overiť, že preklad vyprodukuje rovnaký kód, ako je očakávaný. Všetky očakávané programy v jazyku pre nástroj ARTMC sú taktiež súčasťou testov. Zoznam testov je uvedený v prílohe B.

Testy boli produkované postupne a pre každú dôležitú zmenu bol vytvorený test, ktorý testoval danú konštrukciu. Zároveň existujúce testy overili, že zmenou kódu neboli zanesené žiadne chyby.

Aby bolo možné testovať prenosieľnosť programu medzi rôznymi verziami Pythonu, bol využitý nástroj tox. Funkcionalita tohoto nástroja je popísaná v podsekcii 5.3.2.

6.7 Navrhovaná práca

Všetky požadované vlastnosti pre výslednú aplikáciu boli implementované. Avšak je možné, že časom si používanie ARTMC bude vyžadovať ďalšie funkcie, napríklad nové konštrukcie z jazyka C a teda bude potrebné daný kód doplniť.

Kapitola 7

Záver

Cieľom tejto práce bol návrh a implementácia prekladača z jazyka C do jazyka pre nástroj ARTMC. Text práce, ako aj samotná práca sa dá rozdeliť do dvoch častí, teoretickej a praktickej. V rámci praktickej časti bol naštudovaný nástroj ARTMC hlavne z pohľadu používania so zameraním na formát vstupu. Následne bola preštudovaná a analyzovaná tvorba prekladačov. Po výbere technológií a postupov pri tvorbe prekladača sa prešlo k časti praktickej - implementácii samotného prekladača.

Ako implementačný jazyk bol zvolený Python. Program sa vyznačuje objektovo orientovaným prístupom, čistotou kódu a nezávislosťou ako na verzii Pythonu tak na neštandardných balíčkoch. Samotný kód je rozdelený na viacero modulov podľa jednotlivých celkov prekladača (scanner, parser, generátor kódu). Do prekladača bolo implementovaných viacero neštandardných a pokročilých funkcionalít, ktoré robia prekladač ako aj nástroj ARTMC lepšie použiteľným.

Na overenie správnej funkcionality bolo vytvorených 16 komplexných testov. Každý test testuje celý priebeh prekladu. Testy je možné spúšťať naraz. Rovnako je možné spustiť všetky testy pre vybrané verzie Pythonu cez nástroj Tox.

Aplikácia spĺňa všetky požiadavky zadania a implementuje aj funkcionalitu navyše. Aplikáciu plánujem aj naďalej udržiavať a v prípade chyby alebo potreby pre ďalšiu funkcionalitu tieto zmeny vykonať. Aplikácia je verejne dostupná v službe GitHub a preto je možné, aby záujemcovia taktiež vykonávali zmeny.

Literatúra

[1] Aho, A. V.; Sethi, R.; Ullman, J. D.: *Compilers.* New Jersey: Prentice-Hall, druhé vydání, 1988, ISBN 0-201-10088-6, 796 s.

[2] Bouajjani, A.; Habermehl, P.; Rogalewicz, A.; aj.: Abstract Regular Tree Model Checking. In *Proceedings of the 7th International Workshop on Verification of Infinite-State Systems (INFINITY 2005)*, ročník 149, San Francisco, USA: Electronic Notes in Theoretical Computer Science, August 2005, ISSN 0909–3206, s. 15–24.

[3] Bouajjani, A.; Habermehl, P.; Rogalewicz, A.; aj.: Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Static Analysis*, Berlin: Springer Verlag, 2006, ISBN 978–3–540–37756–6, s. 52–70.

[4] Meduna, A.: *Elements of compiler design.* Boca Raton: Auerbach Publications, 2008, ISBN 1–4200–6323–5, 286 s.

Prílohy

Príloha A

Obsah CD

Priložené CD obsahuje:

- `c2artmc` - adresár so zdrojovými súbormi aplikácie
- `tests` - adresár s testovacími dátami
- `tex` - adresár so zdrojovými súbormi tejto práce vo formáte LaTeX
- `BP.pdf` - text tejto práce vo formáte pdf

Príloha B

Zoznam testov

Všetky testy sú dostupné na priloženom CD v adresári `c2artmc`. V tabuľke B.1 je ich zoznam so základnými údajmi. Na testy boli prevažne použité a upravené testy z nástroja `Predator`[1].

názov testu	**typ testu**	**riadky vstup**	**riadky výstup**
cdll	Kruhový DLL	48	36
complex_expression	Konktrola prekladu výrazov	20	37
dfs	Prehľadávanie do hĺbka	29	28
dll	DLL	38	31
dll_1	DLL	71	49
dll_concat	Spájanie DLL	78	61
dll_destroy	Rušenie DLL	58	44
dll_duplicate	Kópia DLL	45	36
dll_evenlength	DLL	53	40
dll_insert	Vkladanie do DLL	46	48
dll_insertsort	Triedenie DLL vkladaním	58	53
dsw	Deutsch-Schorr-Waite	52	51
link_leaves	4-násobne vizaný graf	38	36
sll_delete	Rušenie SLL	30	31
sll_reverse	Obrátenie SLL	18	22
tree_b	Binárny strom	100	86

Tabuľka B.1: Zoznam vytvorených testov

[1] http://www.fit.vutbr.cz/research/groups/verifit/tools/predator/