Střední průmyslová škola elektrotechnická Ječná
Ječná 30, 120 00, Praha 2

DLOUHODOBÁ MATURITNÍ PRÁCE

Vlastní překladač programovacího jazyka

Autor
Adam Barborík

Vedoucí práce
Mgr. Alena Reichlová

Datum
5. 4. 2023

Oponent práce
Ing. Jan Šváb

Adam Barborík

barborikadam@gmail.com

Tato práce je odevzdávána jako školní projekt.

Čestné prohlášení

Prohlašuji, že jsem celou práci na téma „Vlastní překladač programovacího jazyka“ vypracoval zcela samostatně a uvádím veškeré zdroje, které jsem použil.

V Praze dne 5. 4. 2023 .....

vlastnoruční podpis

Poděkování

Chtěl bych poděkovat vedoucí mé práce, Mgr. Aleně Reichlové, za vstřícnost a poskytnutí cenných rad při zpracování této práce.

Anotace

V této práci se věnuji vývoji překladače vlastního programovacího jazyka, za použití pouze standardní knihovny. Práce slouží především pro rozšíření vlastní sady znalostí a vědomostí v oboru programování.

Annotation

In this work I am developing a custom programming language compiler completely from scratch, using only the standard library. The work primarily serves to expand my own set of knowledge and understanding in the field of programming.

Klíčová slova

teorie programovacích jazyků, vývoj programovacího jazyka, lexikální analýza, syntaktická analýza, x86 assembly

Keywords

programming language theory, programming language development, lexical analysis, syntax analysis, x86 assembly

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Práci na toto téma jsem se rozhodl realizovat z důvodu svého zájmu o nízkoúrovňové programování. Myšlenku implementace vlastního překladače mám už delší dobu a dlouhodobou maturitní práci považuji za vhodnou motivaci.

Můj zájem o programování začal již na základní škole, kde jsem psal jednoduché statické HTML stránky. Netrvalo mi dlouho, než jsem se přes tvorbu primitivních webů dostal k reálnému programování, konkrétně k jazyku Python. Výběr mého prvního programovacího jazyka byl zcela založen na poznatcích z internetu, kde mi Python byl prezentován jako výborný startovní bod pro začátečníky, avšak na rozdíl od ostatních, je reálně využívaný v oboru. Retrospektivně musím říct, že se rozhodně nestavím na stranu lidí, doporučující Python jako výukový nástroj. Až na střední škole jsem pořádně pochopil koncepty programování a vše z mých předchozích projektů mi začalo být jasnější. Přibližně 2 roky nazpět, ke konci 2. ročníku, jsem započal samostudium jazyka C a hned na to nízkoúrovňového programování obecně. Vyzkoušel jsem si psaní x86 assembly programů jak na úrovní operačního systému, tak pod ním. Naprogramoval jsem jednoduchý primární zavaděč pro spuštění programu na té nejnižší možné úrovni, jaké lze ze softwarového pohledu dosáhnout. Při vývoji tohoto programu jsem narazil na netradiční problém, žádný moderní překladač jazyka C, jako např. GCC nebo MSVC, nepodporuje generování instrukcí pro 16-bitový reálný režim x86 procesorů. Byl jsem tak nucen použít starší, ne tolik populární, ale stále udržovaný překladač OpenWatcom. V tom se mi zrodila myšlenka vývoje vlastního překladače, který by vývoj takových programů usnadnil.

1. Cíl práce

Cílem práce je vytvořit překladač vlastního dialektu jazyka C do symbolického zápisu instrukcí Intel x86, pro 64-bitové verze systémů Windows a Linux a také pro podporu 16-bitového reálného režimu x86 procesorů. Práce řeší vše od syntaktické analýzy jazykových tokenů, přes stavbu syntaktického stromu do konečné generace assembly kódu. Vše za použití pouze standardní knihovny.

Před začátkem implementace překladače musíme promyslet následující body:

- Výběr programovacího jazyka, ve kterém bude překladač napsán.
- Výběr prostředí, ve kterém budeme překladač vyvíjet tak, abychom měli co nejmenší potíže při řešení chyb.
- Návrh změn gramatiky pro náš dialekt programovacího jazyka C.

2. Software

Na počátku práce jsem se musel rozhodnout a vybrat si programovací jazyk, ve kterém bude překladač vyvíjen. Existuje spoustu různých programovacích jazyků, avšak já mám ve výberu jasno. Vybral jsem si programovací jazyk C pro jeho rychlost a spolehlivost, na které má kvůli absenci garbage collectoru zásluhu i programátor samotný.

Dále jsem se musel rozhodnout pro vývojové prostředí, což není v případě jazyka C vůbec jednoduché. Nakonec jsem si zvolil neintegrované řešení - textový editor, kompilátor a debugger samostatně. Kompilátor i debugger jsem zvolil ze sady GCC, tedy kompilátor CC a debugger GDB. Pro debugování výstupu z mého překladače používám interaktivní disassembler IDA64.

V poslední řadě uvádím, že jsem k vývoji použil jak operační systém Windows, tak Linux. Podstatné je to z důvodu změn v ABI těchto operačních systémů, které musí můj překladač podporovat pro jeho správnou funkci interoperability se standardní knihovnou jazyka C a jiných funkcí, které nejsou nativně psané v mém programovacím jazyce.

3. Filozofie

Má filozofie je založena na jednoduchosti a použitelnosti. Věřím, že tyto dva základní principy by měl respektovat každý programátor a měly by se stát standardem v oboru informačních technologií. Realita je ale taková, že obor stále spěje ke složitému, chybovému a pomalému softwaru.

Mnoho programátorů zastává názor, že více kódu znamená lepší softwarové řešení a zároveň to považují za důkaz svých vysokých schopností a dovedností. Ve výsledku jim pak nejde o kvalitu kódu, ale o to, udělat co nejrychleji něco, "co prostě nějak funguje".

Technologický pokrok ve vývoji hardwaru už skoro nic neznamená. Dovoluje akorát takovým programátorům schovat nedostatky jejich softwaru za rychlost nových procesorů a grafických karet. Výsledek je takový, že ač hardware se žene kupředu, software se zdá být stále pomalejší a méně spolehlivý.

4. Styl psaní kódu

Nejdůležitějším aspektem čitelného a úhledného kódu je konzistence. Té dosáhnu pomocí následujících pravidel.

Struktura hlavičkových souborů

1. Ochranný kód `#ifndef #define`
2. `#include "includes.h"`
3. Makra
4. Externí proměnné
5. Uživatelské typy
6. Statické proměnné, enumerátory, atd.
7. Deklarace funkcí

Struktura zdrojových souborů

1. `#include "includes.h"`
2. Globální proměnné
3. Definice funkcí

Bloky

- Deklarace proměnných na začátku bloku
- Složené závorky na vlastním řádku
- Pro bloky bez závorek je následující příkaz vždy psán na stejném řádku

Klíčová slova

- Mezera mezi `if`, `for`, `while`, switch a následující závorkou
- Žádné mezery za začínající závorkou a před končící závorkou
- Závorky v kombinaci se `sizeof`

Uživatelské typy

- Bez přípony `_t`
- Název v CamelCase

Komentáře

- Vždy za použití `/* */` nikoliv `//`

Procedury

- List argumentů obsahuje klíčové slovo `void`, nikoliv prázdné závorky

5. Gramatika

Nejdříve si musíme definovat dva různé koncepty, příkazy (statements) a výrazy (expressions). Příkaz je část kódu, která něco vykonává. Výraz, je speciálním případem příkazu, který něco vyhodnocuje. Každý příkaz musí být ukončen středníkem, aby byl platný. Zvláštním případem jsou příkazy blokové, ty nejsou ukončeny středníkem, ale jsou jasně označeny složenými závorkami.

Překladač používá jednoduchou LL(1) gramatiku. Jedná se o typ gramatiky, která umožňuje určit význam příkazu po přečtení maximálně jednoho dalšího terminálního tokenu po neurčitém počtu neterminálních tokenů. V praxi to pak znamená, že pokud se na vstupu funkce, která určuje o jaký příkaz se jedná, objeví token datového typu, víme, že se jedná o deklaraci proměnné, nebo funkce. Dále víme, že musí následovat token identifikátoru, což je další neterminální token. Hned za ním následuje token terminální, který rozhodne o finálním rozhodnutí algoritmu. V případě deklarace funkce by se jednalo o otevírací závorku, vše ostatní můžeme vyhodnotit jako deklaraci proměnné.

Gramatika dialektu překladače je silně inspirována jazykem HolyC. Hlavním rozdílem od standardního C jsou datové typy, ty mají přímo v názvu uvedenou svou velikost.

signed	unsigned	C ekvivalent
I0	U0	void
I8	U8	char
I16	U16	short
I32	U32	int
I64	U64	long
F32	N/A	float
F64	N/A	double

Tabulka č. 1: Datové typy a jejich ekvivalenty v jazyce C

Dále je nutno zmínit, že hlavičky funkcí připomínají standard C89 (ANSI C). Pokud je funkce procedurou, tj. nepřijímá žádné parametry, musí být deklarována nebo definována následovně:

```
I32 procedura(U0)
...
```

Zápis prázdných závorek, ač v jazyku C znamená, že funkce může brát nespecifikovaný počet parametrů a je kompletně validní `|1|`, je v mém jazyku neplatný a překladač jej nezpracuje.

6. Preprocesor

Preprocesorem rozumíme program, v tomto případě část programu, který zpracovává vstup ve formě zdrojového kódu pro další kroky kompilačního procesu. S kódem je manipulováno pouze v paměti, neprobíhají žádné permanentní změny ve zdrojovém souboru na disku.

Preprocesor provádí jednoduchou záměnu textu na základě direktiv ve zdrojovém kódu. Direktivy jsou řádky započaté znakem mřížky (`#`) a fungují jako speciální příkazy pro generování odpovídajícího textového výstupu. Mimo direktiv se také stará o odstranění komentářů ze zdrojového kódu.

Podporované direktivy:

`#include` - Načte textový soubor referencovaný mezi znaky `<` a `>` pro standardní soubor, nebo uvozovkami pro soubor uživatelský a vloží jeho obsah namísto řádky s direktivou.

`#define` - Definuje symbol pro záměnu textu později v kódu.

`#ifdef` - Podmíněné překládání, nechá nebo odstraní text pod direktivou (až do konce podmínky) na základě toho, zda je symbol preprocesoru definován, či nikoliv.

`#ifndef` - Negovaná obdoba `#ifdef`.

`#endif` - Konec podmínky.

7. Lexikální analýza

Lexikální analýzou nazýváme proces vykonávaný takzvaným scannerem popřípadě lexerem. Je to první ze tří hlavních kroků překladače. Dochází v něm ke zpracování zdrojového kódu na takzvané tokeny, nebo také lexémy. Tokeny jsou základní jednotkou programovacího jazyka a tvoří jeho gramatiku. Jsou to jednotlivá slova, symboly a čísla, které tvoří kód programu. |4|

V překladači je token reprezentován následující strukturou:

```
typedef struct
{
    int token;
    union
    {
        long   i;
        double f;
        Sym   *s;
    } val;
    int line;
} Tok;
```

`Token`, myšleno vlastností struktury tokenu, tvoří hodnoty výčtového typu, který obsahuje veškeré validní jazykové tokeny. Unie `val` nabývá hodnot `i` a `f` v případě, že je token literálem. Pokud se jedná o řetězec, nebo identifikátor, kterému ještě nebyl přiřazen symbol, hodnota `i` obsahuje index do struktury `uniq`, která ukládá pouze unikátní řetězce. Vlastnost `line`, je pomocnou vlastností, se kterou překladač nijak nepracuje, má uživateli pouze usnadnit hledání chyb ve zdrojovém kódu při chybě v překladu.

7.1. Klíčová slova

Klíčová slova jsou slova, která jsou rezervována překladačem a mají speciální význam. Slouží k definici různých typů dat, cyklů, funkcí a dalších konstrukcí. Nelze je používat jako názvy funkcí a proměnných.

U0	U8	U16	U32	U64	extern	goto	return
I0	I8	I16	I32	I64	enum	break	sizeof
F32	F64	if	else	while	for	continue	typedef

Tabulka č. 2: Klíčová slova

7.2. Identifikátory

Identifikátory jsou názvy proměnných, funkcí a jiných uživatelsky definovaných jazykových konstrukcí. Všechny identifikátory mají svůj obor platnosti, což znamená, že jsou viditelné (referencovatelné) pouze v určité části kódu, jinak také nazývané jako blok. Blokem rozumíme část kódu mezi složenými závorkami. Platí, že identifikátor je viditelný v bloku ve kterém byl deklarovaný a všech blocích pod ním. Nemůžeme však referencovat identifikátor deklarovaný v bloku který se hierarchicky nachází níže.

Hierarchicky nejvýše se nachází obor globální. Identifikátory deklarované v tomto oboru jsou viditelné všem funkcím jejichž deklarace se nachází pod deklarací identifikátoru. Globální obor je také jediný obor, ve kterém je možno definovat a deklarovat funkce. Všechny funkce tudíž musí být globální a nemohou být vnořené.

7.3. Operátory

Operátory jsou symboly, popřípadě sekvence symbolů, které slouží k provádění různých aritmetických, logických, relačních, bitových a jiných operací. Operátory mají různou prioritu a asociativitu, což určuje pořadí, v němž jsou operace vyhodnocovány. Kromě toho operátory rozlišujeme na unární (mají pouze jeden operand) a binární (mají dva operandy).

Priorita	**Operátor**	**Popis**	**Typ**	**Asociativita**
1	`++ --`	Inkrement a dekrement	Unární - přípona	Zleva doprava
	`[]`	Přístup do pole		
	`()`	Volání funkce		
2	`++ --`	Inkrement a dekrement	Unární - předpona	Zprava doleva
	`+ -`	Unární plus a minus		
	`! ~`	Logický a bitový NOT		
	`(`*`typ`*`)`	Převod typu		
	`&`	Adresa		
	`*`	Dereference		
	`sizeof()`	Velikost typu nebo výrazu		
3	`* / %`	Násobení, dělení a zbytek	Binární	Zleva doprava
4	`+ -`	Sčítání a odčítání		
5	`<< >>`	Bitový posun vlevo a vpravo		

6	`< <=`
	`> >=`
	`== !=`
		Menší než, menší nebo rovno, větší než, větší nebo rovno, rovnost a nerovnost		

7	`& | ^`	Bitový AND, OR a XOR		
8	`&& ||`	Logický AND a OR		

9	`= += -=`
	`*= /= %=`
	`&= |= ^=`
		Jednoduché přiřazení a přiřazení po operaci (přiřazení součtem, rozdílem atd.)		Zprava doleva

10	`,`	čárka		Zleva doprava

Tabulka č. 3: Operátory, jejich priorita a asociativita

7.4. Literály

Literály, nebo také doslovné hodnoty, označují přímý zápis konstant v kódu. Patří mezi ně celá čísla, desetinná čísla, znaky a řetězce. Mohou mít vícero zápisů.

Podporované alternativní zápisy celých čísel jsou zápisy hexadecimální a binární. Hexadecimální zápis musí obsahovat předponu `0x`, tedy desítkovou hodnotu `255` bychom zapsali jako `0xff`. Binární zápis musí být také s příponou, tentokrát `0b`, číslo `255` by tedy bylo vyjádřeno jako `0b11111111`.

Rozlišení znaků a řetězců probíhá pomocí uvozovek. Znak se píše výhradně mezi uvozovky jednoduché a řetězec mezi uvozovky dvojité. Pokud jediný znak napíšeme mezi dvojité uvozovky, bude uložen jako řetězec s nulovým znakem na konci a bude s ním tak nakládáno.

Znaky a řetězce mohou obsahovat takzvané escape sekvence. To jsou speciální sekvence dvou znaků, kdy prvním znakem je vždy zpětné lomítko. Primárně se používají pro vyjádření znaků, které nazýváme jako řídící, nebo netisknutelné, nemají žádný symbol, který by je reprezentoval a tím pádem se nenachází na klávesnici. Historicky to jsou všechny ASCII znaky s hodnotou menší než 32. Dále se používají v případě, že v řetězci chceme použít znak, který má v jeho kontextu už nějaký význam. Například uvozovka, která by normálně řetězec ukončila, pokud před ni však dáme zpětné lomítko, znak bude ignorován a stane se součástí řetězce.

Escape sekvence	**Hexadecimální ASCII hodnota**	**Reprezentující znak**
`\0`	0x00	Nulový znak
`\a`	0x07	Zvonek
`\b`	0x08	Zpětný skok
`\e`	0x1B	Escape znak
`\f`	0x0C	Konec stránky
`\n`	0x0A	Nový řádek
`\r`	0x0D	Návrat vozíku
`\t`	0x09	Tabulátor
`\v`	0x0B	Vertikální tabulátor
`\\`	0x5C	Zpětné lomítko
`\'`	0x27	Jednoduchá uvozovka
`\"`	0x22	Dvojitá uvozovka

Tabulka č. 4: Escape sekvence

8. Syntaktická analýza

Syntaktická analýza je částí procesu překladu, při které se z jednotlivých tokenů tvoří příkazy a výrazy. Při jejich tvorbě zároveň dochází k mapování tokenů lexikálních, z předchozího kroku, na tokeny syntaktické. Tokeny tak nabývají významu, syntaktické tokeny nepopisují token jednoduše pomocí znaku kterým jsou reprezentovány, ale operací, kterou vykonávají. Z jednoduchého ampersandu (&) tak může vzniknout token pro bitový AND, nebo token pro operátor získání adresy, v závislosti na kontextu, ve kterém se token nachází. |5|

V případě, že při syntaktické analýze dojde k chybě, například chybějící středník, nebo algoritmus narazí na token, který v daném kontextu nedává smysl, veškerý běh překladače se ukončí a chyba je formou standardního výstupu hlášena uživateli.

8.1. Symboly

Pokud algoritmus syntaktické analýzy narazí na token identifikátoru, přiřadí mu symbol. Symbol je strukturou obsahující podpůrné informace a reprezentuje proměnnou či funkci. Symbol je vytvořen při deklaraci proměnné nebo funkce a na základě jeho třídy je uložen ve struktuře `glob`, nebo `local`.

Překladač rozeznává následující třídy:

C_GLOB	-	Globální symbol
C_LOCL	-	Lokální symbol
C_EXTN	-	Externí symbol
C_DATA	-	Konstantní symbol nebo řetězec

8.2. Výrazy

Výrazy, od jednoduchých volání funkcí či procedur po složité sekvence operandů a operátorů se symboly, konstantami a modifikátory, tvoří obrovskou část zdrojového kódu. Pro správnou funkčnost překladače je nutné je správně zpracovávat společně s prioritou a asociativitou operátorů.

Jedním z algoritmů pro implementaci této funkcionality je takzvaný Prattův parser. Algoritmus čte ještě nezpracované lexikální tokeny zleva doprava a pro každý token se dívá na jeden předchozí a *n* následujících tokenů rekurzivně, aby určil, jak parsovat aktuální token. Pokud je priorita aktuálního tokenu vyšší než předchozího tokenu, spojí aktuální token s následujícím výrazem. Pokud je priorita nižší, spojí předchozí token s předcházejícím výrazem. Tato vlastnost čtení binárního výrazu jako dvojici operandů a operátorů umožňuje velmi snadnou implementaci unárních operátorů. |9|

8.3. Syntaktický strom

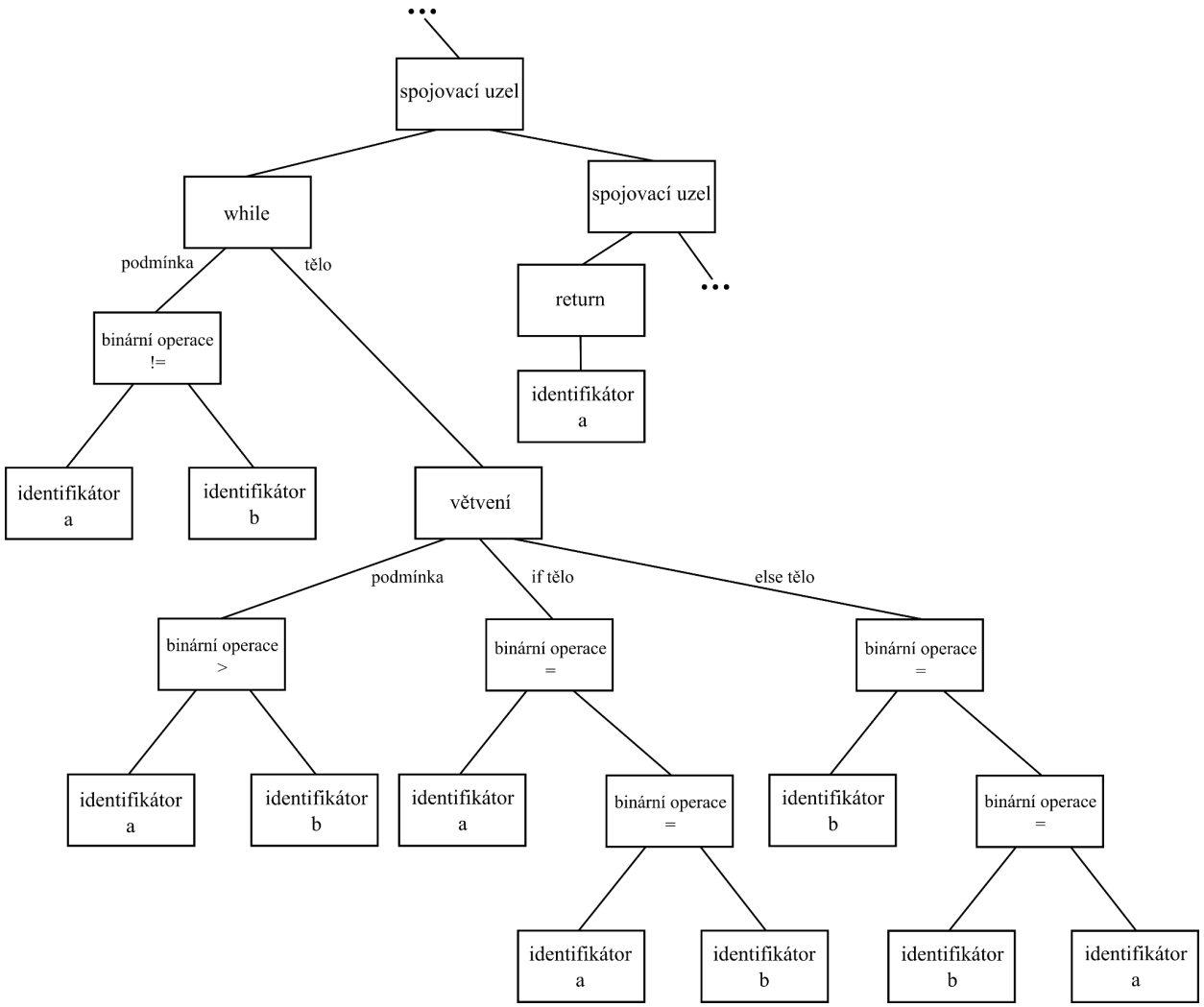
Abstraktní syntaktický strom (AST) je stromová reprezentace syntaktické struktury zdrojového kódu. Je produktem syntaktické analýzy. Jeho účelem je přehledně a systematicky reprezentovat vztahy mezi syntaktickými tokeny. Skládá se z uzlů, které představují příkazy a listy jsou jejich operandy. Neobsahuje určité gramatické prvky jazyka (závorky, středníky atd.), ty jsou použity k jeho sestrojení a jsou tak ve stromu pouze implikovány.

Každý podstrom daného syntaktického stromu je samostatnou jednotkou a lze ho přeložit (nebo interpretovat), za podmínky, že jsou všechny identifikátory v něm referencované platné, kompletně bez kontextu. Speciálním případem jsou pak uzly spojovací, které spojují příkazy, jenž spolu nijak nesouvisí, pouze mají být sekvenčně provedeny za sebou. |6|

Mějme následující kus kódu možné implementace Euklidova algoritmu:

```
...
while (a != b)
{
	if (a > b)
	{
		a = a - b;
	}
	else
	{
		b = b - a;
	}
}
return a;
...
```

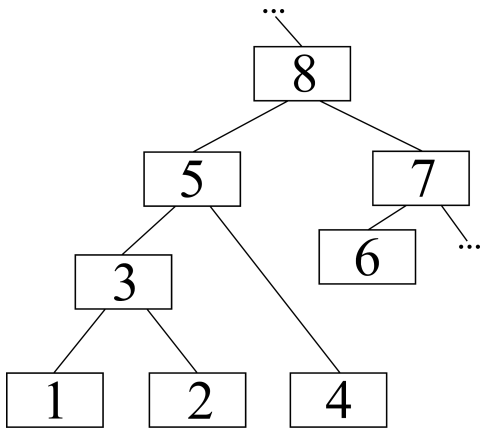
Syntaktický strom pro takový kód by vypadal následovně:



Obrázek č. 1: Syntaktický strom pro možnou implementaci Euklidova algoritmu

9. Generování assembly

Generování assembly kódu je posledním krokem procesu překládání. Pokud se překladač dostane do této fáze překládání je už jisté, že bude celý proces úspěšně dokončen. Proces překládání syntaktických tokenů na assembly funguje na základě procházení syntaktického stromu do hloubky. Generační algoritmus prochází strom od kořene do hierarchicky nejnižšího listu nejvíce vlevo, listy jsou vždy operandy, takže v případě identifikátoru načte jeho hodnotu z paměti a v případě konstanty pouze alokuje registr a načte do něj hodnotu. Nyní vygeneruje assembly reprezentaci operace, popsané rodičovským uzlem obou listů a na stejném principu pokračuje v generaci dále. |2|



Obrázek č. 2: Pořadí uzlů stromu při procházení do hloubky

9.2. Konvence volání funkcí

Před zavoláním funkce, tedy skokem do jiné části kódu za pomocí instrukce `call`, je nutné uložit její argumenty tak, aby se k ním později dostala. Na 32-bitových systémech se tento problém standardně řeší uložením argumentů na zásobník. To se později ukázalo být pomalé a tak se v dnešních 64-bitových systémech používají registry. Pokud počet argumentů přesahuje ABI definovaný počet registrů pro předávání argumentů, je zbytek uložen na zásobník, stejně jako na 32-bitových systémech. |7|

ABI	Typ	1.	2.	3.	4.	5.	6.	7.	8	9. a výše
SystemV	*i*	rdi	rsi	rdx	rcx	r8	r9	zásobník		
SystemV	*f*	xmm0	xmm1	xmm2	xmm3	xmm4	xmm5	xmm6	xmm7	zásobník
Microsoft x64	*i*	rcx	rdx	r8	r9	zásobník				
Microsoft x64	*f*	xmm0	xmm1	xmm2	xmm3	zásobník				

Tabulka č. 5: Pořadí registrů konvencí volání funkcí pro ABI SystemV a Microsoft x64

i - celočíselný typ nebo ukazatel
f - typ s pohyblivou řádovou čárkou

Návratová hodnota je vždy uložena v registru rax a obě ABI vyžadují zarovnání zásobníku při volání na 16 bajtů. |7|

9.2. Složitější jazykové konstrukce

Většina syntaktických tokenů vyžaduje generaci jednoho řádku assembly na token. To neplatí pro blokové konstrukce, ty mají vždy složitější mapování už jen z principu, že musí ukládat celý podstrom kódu v složených závorkách.

9.2.1. Podmínka if - else

Assembly konstrukci podmínky `if - else` lze v přirozeném jazyce vyjádřit následovně:

```
        Vyhodnocení podmínky
        Porovnání s nulou
        Skok na L1 pokud bylo porovnání rovno
        Tělo if
        Skok na L2
L1:
        Tělo else
L2:
        …
```

9.2.2. Cyklus while

Konstrukce cyklu while je velmi podobná konstrukci podmínky `if - else`, pouze bez části else.

```
L1:
        Vyhodnocení podmínky
        Porovnání s nulou
        Skok na L2 pokud bylo porovnání rovno
        Tělo while
        Skok na L1
L2:
        …
```

9.2.3. Cyklus for

Znovu lze vidět podobnost, tentokrát s předchozím cyklem `while`, pouze s přidanou řídící proměnnou.

```
        Alokování místa pro řídící proměnnou
        Nastavení počáteční hodnoty řídící proměnné
L1:
        Vyhodnocení podmínky
        Porovnání s nulou
        Skok na L2 pokud bylo porovnání rovno
        Tělo for
        Inkrementace řídící proměnné
        Skok na L1
L2:
        …
```

10. Kompilace a použití

10.1. Kompilace

Celý program se zkompiluje velmi jednoduše, jedinou jeho závislostí je kompletní sada GCC, obsahující mimo kompilátor a linker také `make`. Pro kompilaci navigujte do kořenového adresáře projektu, ve kterém se nachází soubor `Makefile`. V tomto adresáři spusťte příkaz `make`, kompletně bez argumentů, a po dokončení kompilace se v podadresáři `bin` objeví zkompilovaný binární soubor.

10.2. Použití

Použití programu je velmi jednoduché, pokud je program spuštěn bez argumentů, nebo s přepínačem `-h`, potažmo `--help`, vypíše se následující pomocný text:

```
$ ./abcc [flags] <input file(s)> [-o output file(s)]

flags:
      -h, --help display this help message

      -m mode
         64       64-bit mode
         16       16-bit mode

      -cc calling convention
          sysv    SystemV (Linux, most Unix variants)
          ms64    Microsoft x64 (Windows)
```

11. Licencování

Zdrojový kód programu je licencován pod licencí ABL - Adam Barborík's license, tedy Licence Adama Barboríka. Ta dává absolutní svobodu pro soukromé účely, modifikace a nekomerční distribuci. Pro komerční či výukové účely je nutný písemný souhlas autora zdrojového kódu. V případě, že byl již hotový program stažen v binární podobě, je nutný nejen písemný souhlas autora zdrojového kódu, ale také všech osob, které se podílely na jeho kompilaci.

Závěr

Netvrdím, že program, v podobě v jaké byl odevzdán, je kompletně bez chyb a uživatelsky přívětivý. Hodlám však ve vývoji překladače pokračovat i nadále a pokud ho uznám za dostatečně pohodlný, také v něm vyvíjet i jiné projekty. Rád bych v něm jednou napsal mikrojádro pro vlastní pseudo-operační systém, jak už jsem zmiňoval v úvodu práce.

Dialekt jazyka překladače jsem nazval ABC - Adam Barborík C, spíše než kvůli vlastnímu egu, se mi líbila abecední posloupnost písmen.

Projekt je hostovaný na mém github profilu ve veřejném repozitáři:
https://github.com/barborik/abcc

Na úplný závěr cítím povinnost poděkovat Warrenovi Toomeyovi, který mě svým podrobným návodem vývoje překladače velmi inspiroval, ale také pomohl v raných fázích vývoje, kdy jsem si nebyl jistý kde a jak mám začít. Jeho gitový repozitář je uveden mezi zdroji.

Slovník zkratek a pojmů

Zavaděč - První program, který se spouští při zapnutí počítače. Jeho úkolem je načíst operační systém do paměti a inicializovat hardware. Dále může sloužit k výběru mezi více nainstalovanými operačními systémy na počítači.

Mikrojádro - Typ architektury operačního systému, kde jádro obsahuje pouze základní funkce, jako jsou správa procesů a paměti, zatímco všechny ostatní funkce, jako jsou ovladače zařízení a souborové systémy, jsou umístěny mimo jádro.

AST - Abstract Syntax Tree, viz 8.3. Syntaktický strom.

ABI - Application Binary Interface, nízkoúrovňová obdoba API.

Uzel - Základní stavební element stromové struktury, který obsahuje data a ukazatele na své potomky. Každý uzel může mít libovolný počet potomků (nebo podle typu stromu, binární - max. 2, ternární - max. 3), ale pouze jednoho předka.

List - Speciální případ uzlu, který nemá potomky.

Kořen - Speciální případ uzlu, který nemá předka.

Řetězec - Sekvence znaků.

ASCII - Standard pro kódování znaků.

Zásobník - Paměťová oblast programu převážně pro lokální proměnné, ale také pro návratovou adresu při skoku do funkce nebo argumenty funkce. Jeho funkce je založena na LIFO (Last-In-First-Out) principu, což znamená, že poslední vložená položka do zásobníku bude jako první odebrána.

Zdroje

Literatura

|1| KERNIGHAN, Brian W. a Dennis M. RITCHIE. *Programovací jazyk C*. 2. vydání. Přeložil Zbyněk ŠÁVA. Brno: Computer Press, 2019. ISBN 978-80-251-4965-2.

|2| WRÓBLEWSKI, Piotr. *Algoritmy*. Brno: Computer Press, 2015. ISBN 978-80-251-4126-7.

Elektronické zdroje

|3| A Language Design Analysis of HolyC [online], Harrison Totty. Dostupné z:
https://harrison.totty.dev/p/a-lang-design-analysis-of-holyc

|4| Lexikální analýza [online], Wikipedie. Dostupné z:
https://cs.wikipedia.org/wiki/Lexik%C3%A1ln%C3%AD_anal%C3%BDza

|5| Syntaktická analýza [online], Wikipedie. Dostupné z:
https://cs.wikipedia.org/wiki/Syntaktick%C3%A1_anal%C3%BDza

|6| Syntaktický strom [online], Wikipedie. Dostupné z:
https://cs.wikipedia.org/wiki/Syntaktick%C3%BD_strom

|7| Calling Conventions [online], OsDev. Dostupné z:
https://wiki.osdev.org/Calling_Conventions

|8| A Compiler Writing Journey [online], Warren Toomey. Dostupné z:
https://github.com/DoctorWkt/acwj

|9| Simple but Powerful Pratt Parsing [online], Alexander Kladov. Dostupné z:
https://matklad.github.io/2020/04/13/simple-but-powerful-pratt-parsing.html

Přílohy

Příloha č. 1: Licenční text ABL licence

The Adam Barborík's License (hereinafter referred to as "License")
grants the user the right to use, modify and distribute the
source code of the program and compiled binaries provided under
this license in a non-commercial setting.

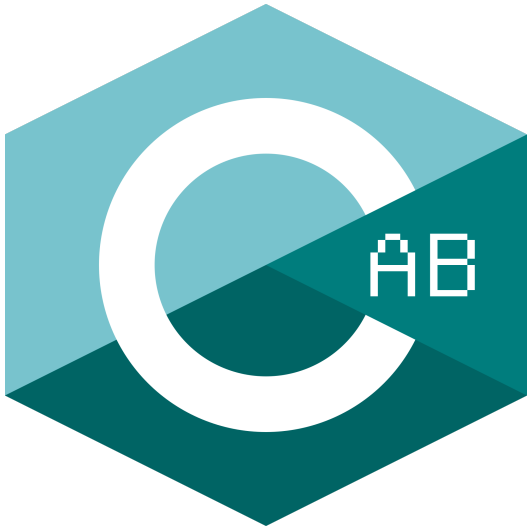
The use of this program for commercial or educational purposes is only
allowed with the author's written consent, who must also grant permission
for the use of the source code. Apart from the written consent, which can
be revoked at any time at the authors will, the author must also be informed
every single time the program or its source code is to be used.

If the program has already been downloaded in binary form, written consent
is required not only from the author of the source code but also from all
people involved in its compilation for further use.
The author is not responsible for any damages caused by the use of the program.

The license is not tied to any specific product.

For more information, please contact the author of this license
via email at barborikadam<at>gmail<dot>com.

Příloha č. 2: Logo jazyka ABC



Následující příklady byly psány v rané fázi vývoje překladače, některé příkazy mohou být zjednodušeny.

Příloha č. 3: Program pro výpočet prvních deseti Fibonacciho čísel

```
extern U0 printf(I8* fmt, ...);

U0 fib(I32 i)
{
    I32 j;
    j = 0;

    I32 a;
    I32 b;
    I32 c;

    a = 0;
    b = 1;
    c = 0;

    while (j < i)
    {
        printf("%d\n", a);

        c = a + b;
        a = b;
        b = c;

        j = j + 1;
    }
    return;
}

I32 main(U0)
{
    fib(10);
    return 0;
}
```

Příloha č. 4: Program pro výpočet faktoriálu čísla, které je programu předáno ve formě parametru

```
extern U0 printf(I8* fmt, ...);
extern I32 atoi(I8* str);

I32 fac(I32 i)
{
    if (i == 0)
    {
        return 1;
    }
    return i * fac(i - 1);
}

I32 main(I32 argc, I8** argv)
{
    printf("%d\n", fac(atoi(argv[1])));
    return 0;
}
```