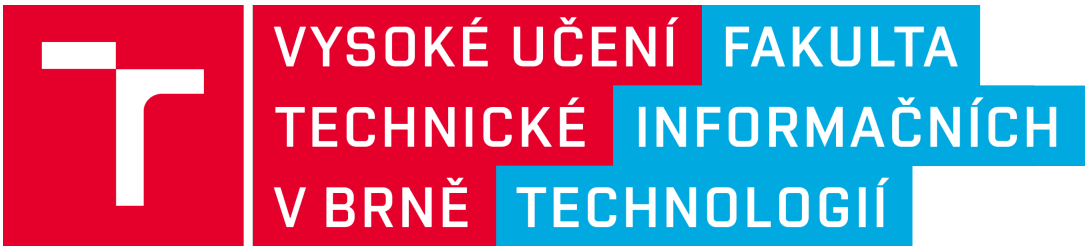


IFJ, IAL 2020 - Dokumentace

Implementace překladače jazyka IFJ20

Tým 48 - Varianta II

Implementovaná rozšíření: UNARY, FUNEXP

Marek Filip (**xfilip46**) – 27%
Vojtěch Bůbela (xbubel08) – 23%
Vojtěch Fiala (xfiala61) – 23%
Ondřej Míchal (xmicha80) – 27%

Obsah

1 Úvod

Tato dokumentace popisuje implementaci překladače imperativního jazyka IFJ20, který je založen na jazyce Go. Implementace překladače byla zadána jako projekt do předmětů IFJ a IAL.

2 Práce v týmu

2.1 Komunikace

Týmová komunikace probíhala převážně osobně a nebo skrz aplikaci Discord. Náš tým měl každý týden pravidelné schůzky, na kterých jsme probírali pokrok, jakého jsme za uplynulý týden dosáhli a určovali si milníky, kterých chceme dosáhnout do další schůzky.

2.2 Verzovací systém

Pro správu verzí projektu jsme použili verzovací systém git a platformu GitHub.

2.3 Rozdělení práce

Základní rozdělení práce na jednotlivých částech projektu proběhlo na prvním meetingu. Následně jsme si pak s jednotlivými problémy ještě navzájem vypomáhali. Konkrétní rozdělení práce popisuje následující tabulka:

Marek Filip (**xfilip46**)	Organizační záležitosti, syntaktická analýza, sémantická analýza, zásobníková struktura, testování, dokumentace
Vojtěch Bůbela (xbubel08)	Lexikální analýza, generování cílového kódu, testování, dokumentace
Vojtěch Fiala (xfiala61)	Lexikální analýza, generování cílového kódu, testování, dokumentace, prezentace
Ondřej Míchal (xmicha80)	Lexikální analýza, sémantická analýza, tabulka symbolů, testování, dokumentace, git podpora

Na mírně nerovnoměrném rozdělení bodů jsme se společně dohodli na základě toho, že mezi výkonností jednotlivých členů byly rozdíly a shodli jsme se, že je tímto způsobem dorovnáme.

3 Implementace jednotlivých částí projektu

3.1 Lexikální analýza

První krok při tvorbě překladače je lexikální analýza. Pro její implementaci jsme vytvořili deterministický konečný automat založený na diagramu 1.

Ze standardního vstupu jsou načítány jednotlivé znaky (v případě, že se jedná o bílý znak či komentář, je zahozen), které dále DKA analyzuje, přidává do struktury `string` [1] a skládá z nich tokeny. Token je reprezentován strukturou `token_t`, která se skládá z typu a atributu.

Typ tokenu může být buď identifikátor, literál, datový typ, operátor a nebo speciální typ – konec řádku, konec souboru a nebo neplatný typ. Jestliže se jedná o identifikátor, je jeho název dále porovnán s tabulkou klíčových slov a na základě výsledku tohoto srovnání se rozhodne, zda se jedná o klíčové slovo či název proměnné a nebo funkce.

Atribut tokenu je tvořen datovou strukturou `union` a v případě, že je typ tokenu literál, obsahuje jeho hodnotu.

Výsledný token je poté předán k dalšímu zpracování.

[1] Struktura `string` pochází z knihovny `str.h`, která je dostupná na stránkách projektu předmětu IFJ

4 Syntaktická analýza

Při návrhu jsme čerpali z populární mezinárodní knihy známé pod pseudonymem „Dragon Book“ [1]. Zádání nám určilo využití LL(1) gramatiky pro analýzu shora dolů a metodu precedenční tabulky pro analýzu zespodu nahoru. Jako první se sepsali pravidla LL(1) gramatiky. Pro správný postup byly použity instrukce získané z Dragon Book. Revize této gramatiky však probíhala několikrát, z důvodu neúplného pochopení zadání či špatného prvotního návrhu.

Sestavená gramatika (včetně gramatiky pro výrazy pro první nulté odevzdaní) pak byla využita pro sestavení LL tabulky. Pro sestavení tabulky jsme postupovali dle instrukcí získaných ze záznamu democvičení IFJ ze dne 18. 10. 2011. Správnost LL tabulky (pro účely implementace) jsme pak zkontrolovali pomocí webové aplikace autorů Radima Kocmana a Dušana Koláře[2].

Analýza zhora dolů je pak implementována metodou rekurzivního sestupu, kde se gramatická pravidla rozvíjí dle volání jiných funkcí nebo rekurzivního volání sebe sama. Tento postup se zdál jednodušší z důvodu banální implementace a absence zásobníku. Díky rozšíření FUNEXP jsme nemuseli řešit předávání kontroly nějakým ”hackem”a výsledný kód by měl proto působit „čistěji“.

Precedenční analýzu jsme započali po nultém pokusném odevzdání. V týmu jsme se dohodli na zvolení rozšíření FUNEXP a UNARY, takže se již od začátku počítalo se složitějším návrhem a následnou implementací. Rozšíření FUNEXP znamenalo hodně implementačních detailů a problému při různém stupni zanoření volání funkce a jejího správného vyhodnocení vzhledem na ostatní operátory včetně čárky, z pohledu tabulky to znamenalo přidat operátor `,` , který má nejmenší precedenci s tím, že na některých místech tabulky přibylo více -”operátorů.

Pro precedenční analýzu je použita struktura zásobníku (dále stack). Stack je implementoval obecně, díky chytrému využití maker pro abstrakci datového typu obsaženého ve stacku a díky shell skriptu generující nové soubory jsme mohli tento stack s lehkostí využít i pro potřeby sémantické analýzy a generování kódu. Na základu syntaktické analýzy je vybudována analýza sémantická.

[2] http://www.fit.vutbr.cz/~ikocman/llkptg/

5 Sémantická analýza

Implementace sémantické analýzy blízce následuje syntaktickou analýzu. V úvodní části se značným problém prokázala být kombinace analýzy zhora dolů a zdola nahoru. Důvodem byla tvrdá hranice mezi informacemi, které každá část analýzy má.

V rámci analýzy se ihned generuje tříadresný kód. Tento přístup byl zvolen s vidinou ulehčené práce. V průběhu implementace se ale objevila potřeba pro alespoň omezené použití abstraktního syntaktického stromu. Během analýzy se bohatě využívá několik datových struktur:

- tabulka symbolů uživatelských funkcí
- tabulka symbolů interních funkcí
- zásobník volaných funkcí
- zásobník dočasných identifikátorů
- zásobník proměnných na levé straně přiřazení
- zásobník typů, které by volaná funkce měla vrátit ve výrazu

V rámci vyhodnocování výrazů se každý výsledek redukce uložil do dočasné proměnné. Tento přistup umožnil už poměrně lehké propojení obou částí analýzy (zhora dolů a zdola nahoru). Zároveň ale tento přístup přináší značnou režii.

Rozšíření FUNEXP přineslo komplikaci ve formě nejasnosti vracených hodnot z volání funkcí. Tyto nejasnosti byly částečně vyřešeny sledováním ”hloubky” zanoření ve výrazu a na základě kontextu se pro funkci určí možná vrácená hodnota. I bez rozšíření FUNEXP je ale volání funkcí v ifj20 náročnější, protože nevyžaduje, aby funkce před voláním byla deklarována. Proto, kromě již zmíněného mechanizmu, se každé funkční volání přidává na zásobník. Po zdánlivě úspěšném zpracování analýzy se každé funkční volání porovná s tabulkami symbolů, kde jsou dostupné signatury interních i uživatelských funkcí.

Interně vyhodnocování výrazů analýza rozumí relačním operacím, ale neumožňuje práci s nimi mimo výrazy v hlavičkách prvků řídích běh programu (if, for).

Inspirace pro sémantickou analýzu: [2]

5.1 Generování cílového kódu

Generování cílového kódu IFJ20code je posledním krokem v běhu našeho překladače. Na začátku generování kódu dochází k vygenerování všech podporovaných vestavěných funkcí na začátek výstupu. Mezikód je generován na základě instrukcí v podobě tříadresného kódů, které jsou generátoru předány jako položky v lineárně vázaném seznamu. Při tvorbě struktury tříadresného kódů jsme čerpali z [3].

Argumenty tříadresného kódu jsou typově rozděleny - prefixy `i,f,s,d,b` značí, o jaký typ argumentu se jedná (`int,float,string,identifier, bool`). Na základě těchto prefixů jsou složky tříadresného kódu zpracovány a převedeny do formátu vhodného k vygenerování instrukce.

Následně se na základě jednotlivých instrukcí získaných z tříadresného kódů generuje výsledný mezikód.

6 Datové struktury

6.1 Tabulka symbolů

Dle zadání je implementace založena na tabulce s rozptýlenými položkami. Při implementaci se vycházelo z kódu, který byl vytvořen v rámci předmětu IJC.

Implementace tabulky symbolů probíhala v čtyřech iteracích, kdy se kromě funkcionality a implementačních detailů řešily problémy s komplexností API. Komplexnost se ve výsledku vyřešila tak, že veřejné API je v souboru `symtable.h` a zbytek je v interní `symtable-private.h`.

Tabulka symbolů je komplexní struktura, která funguje na několika úrovních:

- Funkční bloky oddělují symboly v jednotlivých funkcích. Slouží také pro reprezentaci globálního rámce. Jsou implementovány jako jednosměrný lineární seznam. Funkční bloky obsahují identifikátor funkce, parametry, návratové hodnoty a zásobník rámců.
- Zásobník rámců je lineárním seznam, kdy nejvrchnější prvek představuje nejhlouběji zanořený rámec. Struktura, reprezentující rámec, obsahuje odkaz na vnější rámec a samotnou tabulku symbolů.
- Tabulka symbolů je tabulka s rozptýlenými položkami, která obsahuje klíče, které jsou asociovány se symboly.
- Symbol je struktura, představující identifikátor, který obsahuje token (viz Lexikální analýza 3.1).

Lineární seznam byl pro většinu struktur zvolen pro lehkost implementace a nízkou paměťovou režií. Samotná hashovací tabulka je schopná se realokovat, pokud její kapacita začne být naplňována. K realokaci dojde, pokud lineární seznam v polích tabulky dosáhne zhruba 70% maximální „povolené“ délky.

Po vytvoření tabulka symbolů obsahuje vždy jeden funkční blok (globální), v něm zásobník rámců a v něm jeden rámec.

6.2 Lineární seznam - 3AC

Jednosměrný lineárně vázaný seznam je implementován ve 2 souborech – `three-address-code.c` a `three-address-code.h`. Je založen na domácí úloze č. 1 z předmětu IAL. Využíván je k předávání instrukcí tříadresného kódu. Skládá se z ukazatele na aktivní položku a na první položku v seznamu. Tělo jednotlivých položek obsahuje typ operandu, 3 argumenty (výsledek a 2 parametry operace) a ukazatel na další položku. Tělo tříadresného kódu je založeno na [3].

7 Závěr

Po zveřejnění zadání nás projekt zarazil svou rozsáhlostí a komplexitou. Projekt jsme vypracovávali od jeho zveřejnění, nicméně jeho kompletní dokončení se nám nepodařilo. Jednotlivé části jsme testovali za pomoci knihovny gtest[3] a větší celky pak za pomoci end-to-end testů a knihovny bats[4]. Při řešení projektu jsme využili znalosti získané na přednáškách předmětů IFJ a IAL, pro některé z nás to byla také první zkušenost s větším projektem. Jako tým jsme byli dohodnutí již předem, tedy k žádným problémům nedocházelo. Při řešení projektu se projevily rozdíly ve schopnostech jednotlivých členů týmu a tyto rozdíly jsme se tedy rozhodli kompenzovat úpravou finálního rozdělení.

[3] https://github.com/google/googletest
[4] https://github.com/bats-core/bats-core

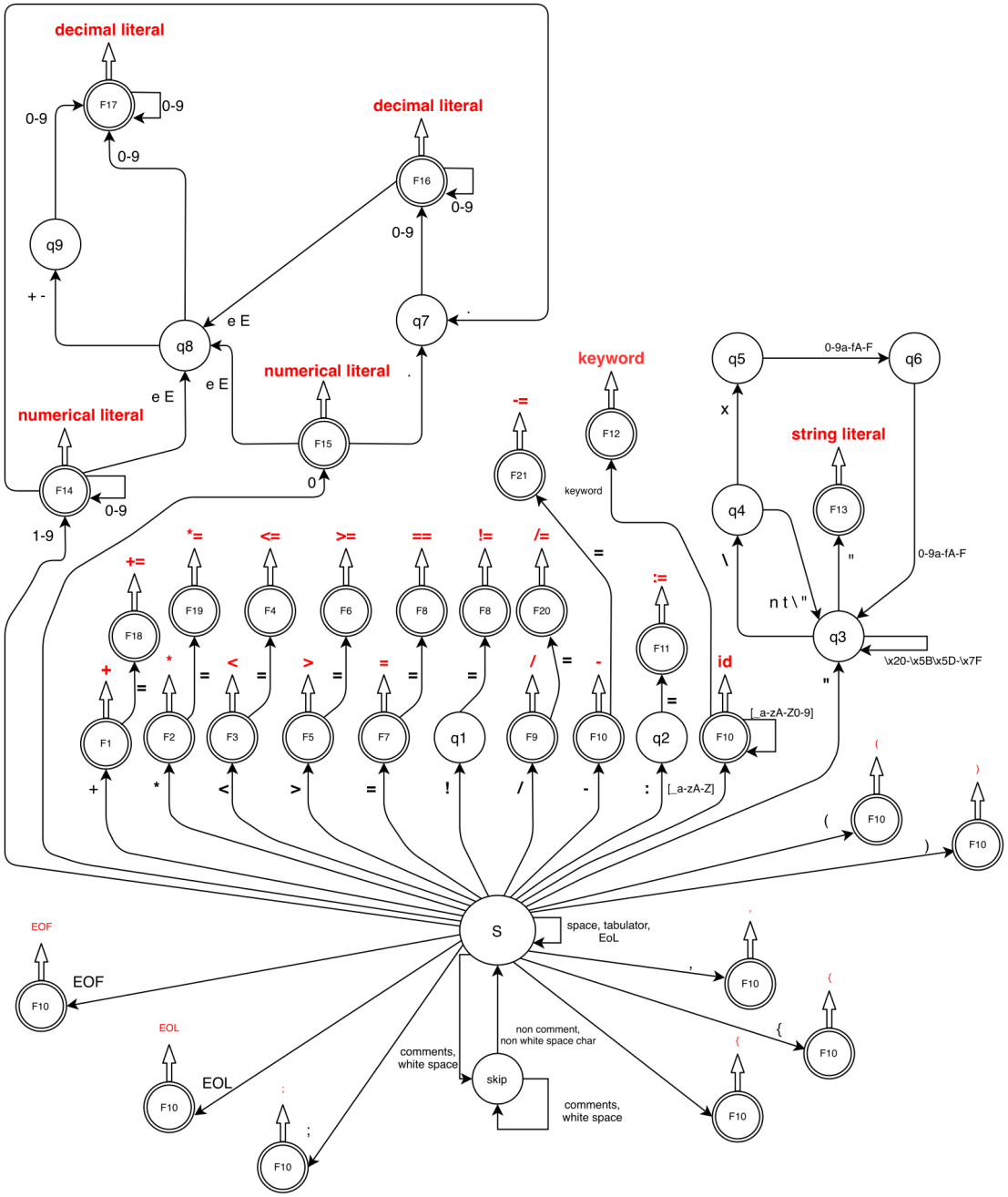
8 Literatura

[1] AHO, A. V.; LAM, M. S.; SETHI, R.; aj.: *Compilers: Principles, Techniques, and Tools*, kapitola Syntax Analysis. Boston: Addison Wesley, 2006, ISBN 978-0321486813, str. 191.

[2] AHO, A. V.; LAM, M. S.; SETHI, R.; aj.: *Compilers: Principles, Techniques, and Tools*, kapitola A simple Syntax-Directed translation. Boston: Addison Wesley, 2006, ISBN 978-0321486813, str. 64.

[3] AHO, A. V.; LAM, M. S.; SETHI, R.; aj.: *Compilers: Principles, Techniques, and Tools*, kapitola Code Generation. Boston: Addison Wesley, 2006, ISBN 978-0321486813, str. 366.

9 Příloha – Deterministický konečný automat pro lexikální analýzu



Obrázek 1: Deterministický konečný automat

10 Příloha – LL gramatika

Top-down grammar

1. **S** -> PROLOG FUNCS
2. **PROLOG** -> package id[‘main’]
3. **FUNCS ->** func <NO_EOL_START> id PARAMS R_PARAMS FUNCS
   | eps
4. **STMTS** -> STMT STMTS
   | eps
5. **STMT** -> id ID_FIRST
   | if <NO_EOL_START> EXPR BLOCK else <NO_EOL_START> BLOCK <EOL>
   | for <NO_EOL_START> OPTDEF ; EXPR ; OPTASSIGN BLOCK <EOL>
   | RETURN <EOL>
6. ID_FIRST -> DEF <EOL> | ASSIGN <EOL> | C_PARAMS <EOL> | += EXPR | -= EXPR | *= EXPR | /= EXPR

Help rules

7. TYPE -> int | float64 | string
8. BLOCK -> { <NO_EOL_END> STMTS }

Parameters definition

9. PARAMS -> (PARAMS_N
10. PARAMS_N -> PARAM)
    |)
11. PARAM -> id TYPE PARAM_N
12. PARAM_N -> , id TYPE PARAM_N
    | eps
13. R_PARAMS -> (R_PARAMS_N
    | eps
14. R_PARAMS_N -> R_PARAM)
    |)
15. R_PARAM -> TYPE R_PARAM_N
16. R_PARAM_N -> , TYPE R_PARAM_N
    | eps
17. C_PARAMS -> (C_PARAMS_N
18. C_PARAMS_N -> C_PARAM)
    |)
19. C_PARAM -> EXPR C_PARAM_N
20. C_PARAM_N -> , EXPR C_PARAM_N
    | eps

Var definition and assign

21. DEF -> := EXPR
22. ASSIGN -> ID_N = EXPRS

23.ID_N -> , id ID_N
| eps
24.EXPRS -> EXPR EXPR_N
25.EXPR_N -> , EXPR EXPR_N
| eps

Helper ‘for’ rules

26.OPTDEF -> id DEF
| eps
27.OPTASSIGN -> id ASSIGN
| eps

Return statement

28.OPTRETURN -> RETURN
| eps
29.RETURN -> return OPTEXPRS
30.OPTEXPRS -> EXPRS
| eps

Bottom-up grammar

1. EXPR -> E
2. E→id | lit
3. E → (E)
4. E→E REL E
5. E→E ADD E
6. E→(E)
7. E→ADD E
8. E→id ( )
9. E→id (E)
10. E→id (E, …)

Get Index depending on the operation needed here

11.REL -> < | <= | > | >= | == | !=
12.ADD -> + | -
13.MUL -> * | /

11 Příloha – LL tabulka

	package	id	func	if	expr	else	for	;	+=	-=	*=	/=	int	float64	string	{	}	)	,	(	:=	=	return	$
$	1																							
PROLOG	2																							
FUNCS			3																					4
PARAMS																22								
R_PARAMS																				28				
BLOCK																21								
STMTS		5		5			5										6						5	
STMT		7		8			9																10	
ID_FIRST									14	15	16	17							12	13	11	12		
OPTDEF		47						48																
OPTASSIGN		49						50																
RETURN																							53	
DEF																					40			
ASSIGN																			41			41		
C_PARAMS																				34				
TYPE													18	19	20									
PARAMS		23																24						
PARAM		25																						
PARAMS_N																		27	26					
R_PARAM																30			29					
R_PARAMS																33			32					
R_PARAMS													31	31	31									
C_PARAMS					35													36						
C_PARAMS_N					37																			
C_PARAM																		39	38					
ID_N																			42			43		
EXPRS					44																			
EXPR_N		46		46			46	46								46			45				46	
OPTRETURN																							51	
OPTEXPRS					54																			

12 Příloha – Precedenční tabulka

	,	REL	ADD	MUL	(	)	id	lit	$
,	=	<	<	<	<	=	<	<	
REL	>	>	<	<	<	>	<	<	>
ADD	>	>	>|<	<	<	>	<	<	>
MUL	>	>	>	>	<	>	<	<	>
(	=	<	<	<	<	=	<	<	
)	>	>	>	>		>			>
id	>	>	>	>	=	>			>
lit	>	>	>	>		>			>
$	<	<	<	<	<		<	<	

Grammar rules	
E → id | lit	BASIC
E → (E)	BASIC
E → E REL E	BASIC
E → E ADD E	BASIC
E → (E)	BASIC
E → ADD E	UNARY
E → id ( )	FUNC
E → id (E)	FUNC
E → id (E, …)	FUNC

>|<	UNARY symbol → buď > nebo <
REL	> < >= <= '==' !=
ADD	+ -
MUL	* /