Dokumentace k týmovému projektu IAL a IFJ

Implementace překladače imperativního jazyka IFJ18

Tým 079, varianta 1

Petr Bobčík	**xbobci02**	(27 %)
Jan Havlín	**xhavli47**	(27 %)
Jakub Karpíšek	**xkarpi06**	(27 %)
Jan Vavřina (Vedoucí)	**xvavri10**	(19 %)

5. prosince 2018

Obsah

1. Úvod

Tato dokumentace slouží k popisu naší implementace překladače imperativního jazyka **IFJ18**, který je zjednodušenou podmnožinou jazyka **Ruby 2.0**. Vybrali jsme si první variantu projektu, která nám určovala implementaci tabulky symbolů pomocí binárního vyhledávacího stromu.

Překladač lze rozdělit na několik modulů:

- Scanner – stará se o lexikální analýzu
- Parser – stará se o syntaktickou i sémantickou analýzu pomocí metody shora-dolů
- PSA – stará se o syntaktickou i sémantickou analýzu pomocí metody zdola-nahoru
- Generátor – slouží ke generování kódu výsledného jazyka IFJ18

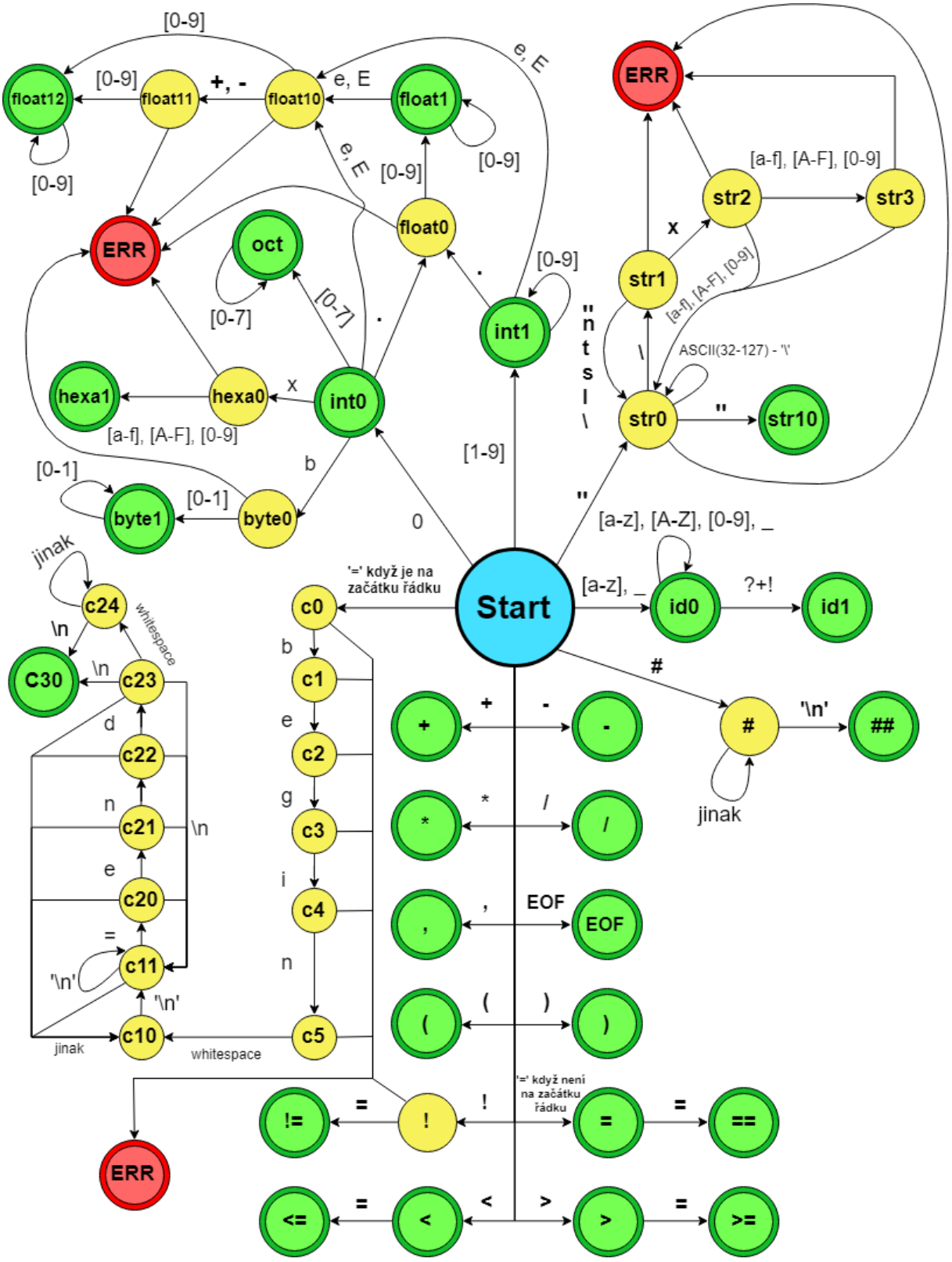
Tyto moduly jsou podrobněji popsány v časti Implementace.

2. Implementace

2.1 Scanner

Scanner se stará o lexikální analýzu. Na základě žádostí parseru a PSA načítá a rozděluje vstupní data na jednotlivé lexémy, určuje jejich typ a následně je, i s jejich daty, posílá k syntaktické a sémantické analýze. V naší implementaci se o většinu vnitřní logiky scanneru stará funkce *getToken()*, která načte token a uloží ho do struktury TToken, kterou vrátí jako návratovou hodnotu. Pokud se ve zdrojovém souboru vyskytne token, který není validní, překladač skončí a vrátí hodnotu 1 (lexikální chyba).

Konečný stavový automat pro lexikální analýzu



2.2 Parser

Stará se o syntaktickou a sémantickou analýzu pomocí metody shora-dolů, konkrétně metody rekurzívního sestupu. Parser je srdcem celého kompilátoru. Funkce *parse()* je volána na začátku a z ní je volána funkce *start()*, která reprezentuje neterminál <START>. Pro každý neterminál je v parseru jedna funkce stejného názvu a v nich jsou reprezentována jednotlivá pravidla **LL gramatiky**. Po každém načtení nového tokenu je provedena syntaktická kontrola. Po načtení tokenu typu TOK_ID jsou prováděny sémantické kontroly, se kterými nám pomáhá tabulka symbolů. Z ní lze zjistit, zda již byla proměnná definovaná a případně odhalit sémantickou chybu.

LL gramatika

<START>	→	<STAT> **EOL** S
<START>	→	<FUN-DEF> **EOL** S
<START>	→	**EOF**
<ST-LIST>	→	<STAT> **EOL** <ST-LIST>
<ST-LIST>	→	epsilon
<FUN-DEF>	→	def id (<P-LIST>) **EOL** <ST-LIST> end
<P-LIST>	→	<TERM P-LIST>
<P-LIST>	→	, <TERM P-LIST>
<P-LIST>	→	epsilon
<STAT>	→	id <ASS-OR-FUN>
<STAT>	→	if expr then **EOL** <ST-LIST> else **EOL** <ST-LIST> end
<STAT>	→	while expr do **EOL** <ST-LIST> end
<STAT>	→	epsilon
<ASS-OR-FUN>	→	= <ASSIGN>
<ASS-OR-FUN>	→	<P-BODY>
<ASSIGN>	→	id <P-BODY>
<ASSIGN>	→	expr
<P-BODY>	→	(<P-LIST >)
<P-BODY>	→	<P-LIST>
<TERM>	→	id
<TERM>	→	const
<TERM>	→	nil
<ASSIGN>	→	epsilon

2.3 PSA (precedenční syntaktická analýza)

Stará se o syntaktickou a sémantickou analýzu zdola-nahoru. Scanner na žádost PSA pošle token, který se následně s nejvyšším tokenem na zásobníku zpracuje a funkce lookUpPrecedenceTable() na základě těchto tokenů vrací pravidlo, které se má na výraz aplikovat (shift, reduce, equal, empty). PSA končí ve chvíli, kdy přijde pravidlo empty a je zásobník prázdný, pote skončí úspěšně a do bufferu scanneru vrací poslední načtený token, jinak končí neúspěšně.

Tabulka pro precedenční analýzu

	+	-	*	/	<	>	<=	>=	==	!=	(	)	i	$
+	>	>	<	<	>	>	>	>	>	>	<	>	<	>
-	>	>	<	<	>	>	>	>	>	>	<	>	<	>
*	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<=	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>=	<	<	<	<	>	>	>	>	>	>	<	>	<	>
==	<	<	<	<	<	<	<	>	>	>	<	>	<	>
!=	<	<	<	<	<	<	<	>	>	>	<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	=	<	ERR
)	>	>	>	>	>	>	>	>	>	>	ERR	>	ERR	>
i	>	>	>	>	>	>	>	>	>	>	ERR	>	ERR	>
$	<	<	<	<	<	<	<	<	<	<	<	ERR	<	ERR

2.4 Generátor

Generátor se skládá z funkcí, které volá parser a PSA. Tyto funkce vytváří instrukce v tříadresném kódu a vkládá je do dvojsměrně vázaného seznamu instrukcí. Každý prvek seznamu obsahuje jednu instrukci, uloženou ve struktuře TInst, která se skládá z operačního kódu a až tří adres. Každá instrukce TInst přesně odpovídá jedné instrukci cílového jazyka IFJcode18. Generátor se rovněž stará o veškerou typovou kompatibilitu operací. Generátor nepoužívá žádné zásobníkové instrukce, z tohoto důvodu je nutné vytvářet dočasné proměnné s unikátním názvem, což přidává práci parseru a PSA. Při výpisu seznamu se na začátek přidají vestavěné funkce jazyka IFJ18.

2.5 Tabulka symbolů

Jak již bylo zmíněno, tabulka symbolů je implementována jako binární vyhledávací strom. Volili jsme rekurzivní implementaci kvůli jednoduchosti. V naší implementaci byl velkou oporou druhý projekt předmětu IAL, který posloužil jako kostra našeho binárního stromu. Uzlem našeho stromu je struktura TsymItem, která obsahuje klíč, data a ukazatele na potomky. Uzel může být trojího typu: proměnná, funkce a klíčové slovo. Data TsymData obsahují podrobnější informace o daném uzlu (u proměnných například pořadí, u funkcí například počet parametrů).
Máme dva typy tabulek – globální a lokální. V globální tabulce uchováváme klíčová slova jazyka IFJ18 a definované funkce a v lokálních tabulkách uchováváme pouze definované lokální proměnné, přičemž každá funkce má svou vlastní lokální tabulku symbolů.

2.6 Nedostatky

- Uživatelem definované funkce nevrací návratovou hodnotu
- Nevyřešili jsme speciální případ, kdy definice funkce musí být před svým voláním

2.7 Rozšíření

- BASE

3. Práce v týmu

3.1 Organizace

Online komunikace probíhala pomocí sociální sítě **Facebook**, konkrétně ve skupinovém chatu, jelikož to byla nejjednodušší a nejpřístupnější metoda.
Měli jsme také nepravidelné týdenní meetingy, většinou ve fakultní knihovně. Nicméně ne vždy jsme se mohli setkat všichni v Brně, a proto jsme používali webovou platformu **Appear.com**, která slouží k uskutečňování skupinových videohovorů. Výhodou je, že uživatel se nemusí registrovat. Navíc podporuje sdílení obrazovek jednotlivých účastníků konverzace, což bylo velmi užitečné pro případnou revizi kódu.
Pro programovou část byl použit verzovací systém **Git**, přičemž pro uložiště našeho repozitáře jsme využili platformu **GitHub.**

3.2 Rozdělení práce

Petr Bobčík – PSA, zásobník, typová konverze

Jan Havlín – Generátor, scanner, testovací skript

Jakub Karpíšek – Parser, tabulka symbolů

Jan Vavřina – Dokumentace, testy

4. Závěr

Jelikož to byl náš první týmový projekt, vyskytly se určité problémy při komunikaci mezi členy týmu a při rozdělování práce. Nicméně jsme je postupem času vyřešili. Projekt byl taky hodně časově náročný, jedním z hlavních důvodů byla nutná příprava i do ostatních předmětů. I přes všechny tyto překážky jsme však dospěli alespoň k částečně úspěšné implementaci projektu a naučili jsme se hodně nových věcí.

4.1 Metriky kódu

- Počet zdrojových souborů: **21**
- Celkový počet řádků: **5042**
- Velikost spustitelného souboru: **142 KB**

4.2 Použité zdroje

- Podklady k předmětům IFJ a IAL
- Tabulka symbolů – Wikipedie. [online]. Dostupné z: https://cs.wikipedia.org/wiki/Tabulka_symbolů
- Lexikální analýza – Wikipedie. [online]. Dostupné z: https://cs.wikipedia.org/wiki/Lexikální_analýza