Dokumentace k projektu

2022/2023

Implementace překladače imperativního jazyka IFJ22

Tým xpejch08, varianta BVS

vedoucí týmu - Štěpán Pejchar (xpejch08) -	body(38)	6.12.2022
Ondřej Češka (xceska07) -	body(19)	
Adam Nieslanik(xniesl00) -	body(19)	
Marcin Sochacki(xsocha03) -	body(24)	

Obsah

1 Úvod

Cílem projektu bylo napsat program jazyce C jenž načítá zdrojový kód zapsaný v jazyce IFJ22 a přeloží ho do mezikódu IFJcode22. Jazyk IFJ22 je zjednodušený jazyk PHP. Program funguje jako aplikace v terminálu (tzv. filtr). Vstup programu se nachází na standardním vstupu a výstup buď na standardním výstupu nebo na chybovém výstupu stderr.

2 Implementace

Projekt jsme implementovali sami za pomoci znalostí získaných na přednáškách a demo cvičeních.

2.1 Lexikální analýza

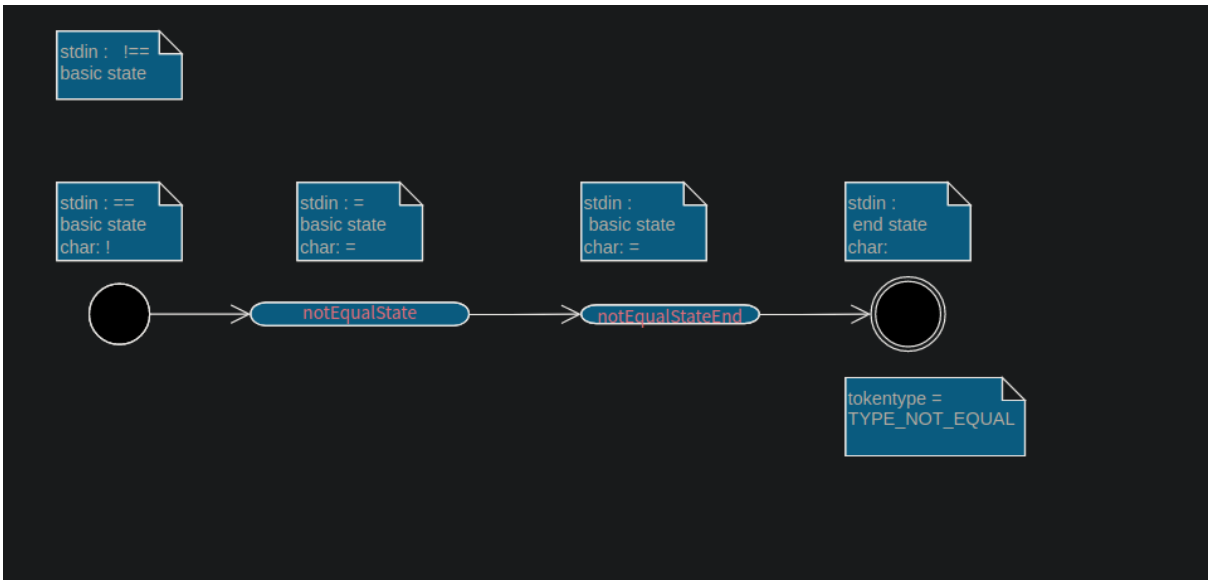
2.1.1 obecně

Lexikální analýza je programová část implementovaná v souborech lexical.c, lexical.h. Jejím úkolem je načítat standardní vstup a postupně generovat tzv. tokeny. Tokeny jsou řetězcové literály, které odpovídají pravidlům jazyka IFJ22. Lexikální analýzu jsme implementovali jako první.

2.1.2 hlavní funkce lexikální analýzy

Hlavní funkce lexikální analýzy je funkce getNextToken. Celá funkce je navrhnutá jako konečný automat. Jediný parametr funkce je “token *attr”. Attr je ukazatel na token, který přepisujeme a přeposíláme v syntaktické analýze. Struktura token je definovaná v souboru lexical.h a obsahuje strukturu tokenContent (také definovanou v souboru lexical.h) a integer type. Content obsahuje 3 dynamické string. Typ string je definován v souboru str.c a funkce pro práci s ním jsou implementované v souboru str.h. Type tokenu je nastaven vždy když se dostaneme do konečného stavu automatu, pokud lexikální analýza nevrací error. Typy, které může token nabývat jsou definované v hlavičce lexical.h (např. TYPE_ADDITION, KEYWORD_WHILE). Funkce getNextToken ignoruje whitespace a komentáře jak jednořádkové, tak blokové. Funkce getNextToken je implementovaná jako switch, jehož stavy jsou definovaná v souboru lexical.c. GetNextToken vždy zavolá první symbol ze standardního stupu a podle něj prostupuje automatem. Pokud lexikální analýza načte symbol, který do daného stavu nepatří vrací LEX_ERROR definovaný v hlavičce errors.h.

příklad načtení tokenu:



2.1.3 ostatní funkce lexikální analýzy

Lexikální analýza obsahuje další 2 funkce, které jsou potřeba aby správně fungovala. Funkce keywordCmp má parametr string* str a token* attr. V parametru str je posílán aktuální obsah tokenu attr. Parametr attr je posílán pro výsledný přepis. Funkce má za úkol porovnat obsah stringu v contentu tokenu a zjistit jestli není roven nějakému klíčovému slovu. Podle toho je token type nastavem na KEYWORD daného klíčového slova. Pokud se string žádnému klíčovému slovu nerovná je type tokenu nastaven na TYPE_IDENTIFIER. Funkce se volá ve funkci getNextToken ve stavu keywordOrIdentifierState

Poslední funkce lexikální analýzy je funkce prolog. Funkce je volána vždy na začátku celkového programu a kontroluje zda se na začátku zdrojového kódu IFJ22 vyskytuje povinný prolog ve správném formátu nebo ne. Pokud prolog ve zdrojovém kódu není, nebo má špatný formát, funkce vrátí SYN_ERROR definovaný s errors.h.

2.2 Syntaktická analýza

2.2.1 Parser Obecně

Syntaktická analýza je implementovaná v souboru parser.c a parser.h. Obsahuje 3 základní funkce: declrlist, statlist a parametrs. Účelem syntaktické analýzy je odhalit syntaktické chyby ve zdrojovém kódu IFJ22. Syntaktickou analýzu jsme implementovali metodou rekurzivního sestupu. Funkce Declrlist a Statlist se navzájem rekurzivně volají, a zanořují, dokud nějaká z nich nevrátí error nebo succes popsaný v souboru errors.h. Potom nastává výstup až do první funkce statlist volaná ve funkci parse. Funkce parse je volána ze souboru main, inicializuje všechny potřebné proměnné a volá první funkci statlist s prvním tokenem.

2.2.2 Funkce statlist

Funkce statlist má za úkol řešit všechny části zdrojového kódu, které nejsou built in konstrukce např.(if, while, funkce). Řeší části uvnitř složených záložek, popřípadě globální části kódu. Je implementovaná jako switch rozdělený podle typu aktivního tokenu. Funkce parse za pomoci globálních stromů implemetovaných v souborech symtable.c a symtable.h. Kontroluje, zda jsou použité proměnné a funkce již definované a vrací příslušné errory. Funkce statlist volá sama sebe rekurzivně, popřípadě volá funkci delcrlist pro kontrolu built in IFJ22 konstrukcí. Dále volá funkci parametrs, která kontroluje parametry funkcí, a funkci precedenceAction, která spouští precedenční analýzu.

2.2.3 Funkce declrlist

Funkce declrlist je velmi podobná funkci statlist. Funkce delcrlist je určená pro kontrolu volaných funkcí, jak built in tak deklarovaných, kontrolu ifů, elsů a whilů. Jejím účelem je odhalit syntaktické a sémantické chyby v built in konstrukcích programu. Funkce declrlist vrací success nebo příslušný error deklarovaný v hlavičkovém souboru errors.h. Funkce dále volá precedenční analýzu (funkci precednceAction) pro kontrolu výrazů v podmínkách if a while. Funkce také obstarává kontrolu ostatních IFJ22 keywordů a konstrukcí jako například keywordu RETURN voláním funkce parametrs.

2.2.4 Funkce parametrs

Funkce parametrs slouží kontroluje jestli všechno při deklaraci funkce, volání funkce, nebo při návratu funkce proběhlo syntakticky správně. V funkcí parametrs kontroluje se taky velká část sémantiky. Parametrs se volá rekurzivně kdykoliv dostaneme nějaký operátor. Pokud funkce proběhne bez problému vrátí SUCCES, pokud nevrátí příslušnou chybu.

2.3. Sémantická analýza

Sémantika programu se kontroluje v souborech parser.c a expression.c. V souboru parser.c v funkci parametrs se kontroluje správný typ návratu funkce, počet parametrů nebo třeba správny typ parametrů vestavěných funkcí. Ke kontrole se využívá tabulka symbolů implementována v souboru symtable.c. V souboru expression.c se během precedenční analýzý kontroluje jestlí proměná v výrazu byla dřív definovaná.

2.4.Precedenční syntaktická analýza

2.4.1 Precedenční syntaktická analýza obecně

Precedenční syntaktická analýza slouží výhradně ke zpracování výrazů, její chování se odvíjí od předem definované precedenční tabulky. Je implementovaná v souboru expression.c s headery expression.h a expstack.h a pro svoji realizaci využívá zásobník “stack”.

2.4.3 Zásobník

Zásobník pro precedenční analýzu je implementován pomocí souborů expstack.c a expstack.h, v kterých jsou definovány struktury a jednoduché funkce pro manipulaci se zásobníkem. V podstatě se jedná o jednosměrný vázaný seznam. Kromě klasických funkcí typu Pop, Push, Init, Dispose, getTop je pro zásobník definována funkce stackGetTopTerminal která vrací ukazatel na terminál, který je nejblíže vrcholu zásobníku. Funkce stackInsertShift, vloží před tento terminál symbol (využití pro akci SHIFT).

2.4.2 Precedenční tabulka - viz příloha

Precedenční tabulka je definovaná na začátku souboru podle tabulky priority a asociativity jednotlivých operátorů ze zadání. Jelikož některé operátory mají stejnou prioritu (asociativitu mají všichni levou), byla tabulka zjednodušená na méně souřadnic (PrtableCoordEnum), které odpovídají příslušným “skupinám” operátorů. V souřadnici osy X je vstupní symbol, na Y ose se nachází symbol terminálu na stacku, který je nejblíže jeho vrcholu. Pomocí těchto souřadnic se určí akce, kterou program vykoná.

2.4.3 Funkce precedenceAction

Precedenční analýza se v případě potřeby (výrazu), volá ze souboru parser.c pomocí funkce precedenceAction, která celý průběh precedenční analýzy řídí. Na začátku funkce se vloží symbol DOLLAR (značí dno zásobníku) na stack, v cyklu while poté následuje několik podmínek pro rychlejší přiřazení do proměnné nebo vyhodnocení podmínky, jedná-li se o jednoduchý, jednoprvkový výraz (např.: $x=5; , if(1)). Dále v tomto cyklu pomocí funkcí s přepínači na začátku souboru převede precedenceAction vstupní token sToken na symbol PrtableSymbolsEnum, zjistí jeho datový typ a poté z něj určí příslušnou X souřadnici PrtableCoordEnum. Podobný proces zopakuje i pro nejbližší terminál na stacku a získá tak souřadnici Y. Ze získaných souřadnic zjistí, která, na přednáškách probíraná akce se má provést. Podle parametru iforass se cyklus while vykonává do chvíle, než se v tokenu objeví “;” (přiřazení) nebo “{“ (if nebo while).

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

2.4.3 Funkce reduceExpression

Nejsložitější akcí je akce REDUCE, pro tuto akci se volá funkce reduceExpression, která pomocí pomocných funkcí zredukuje daný výraz podle pravidla precedenční analýzy. Podle typů operandů ve stacku vybere funkcí pickRule příslušné pravidlo, zjistí pomocí checkTypeForRule výsledný typ výrazu, smaže příslušný počet symbolů ze stacku (ten zjistí funkcí countSymbols) a vloží na stack neterminál s příslušným datovým typem. Kromě redukce výrazu zároveň vypíše příslušné instrukce v IFJcode2022. To zaručuje, že se instrukce vypíší ve správném pořadí.

2.5. Generování kódu

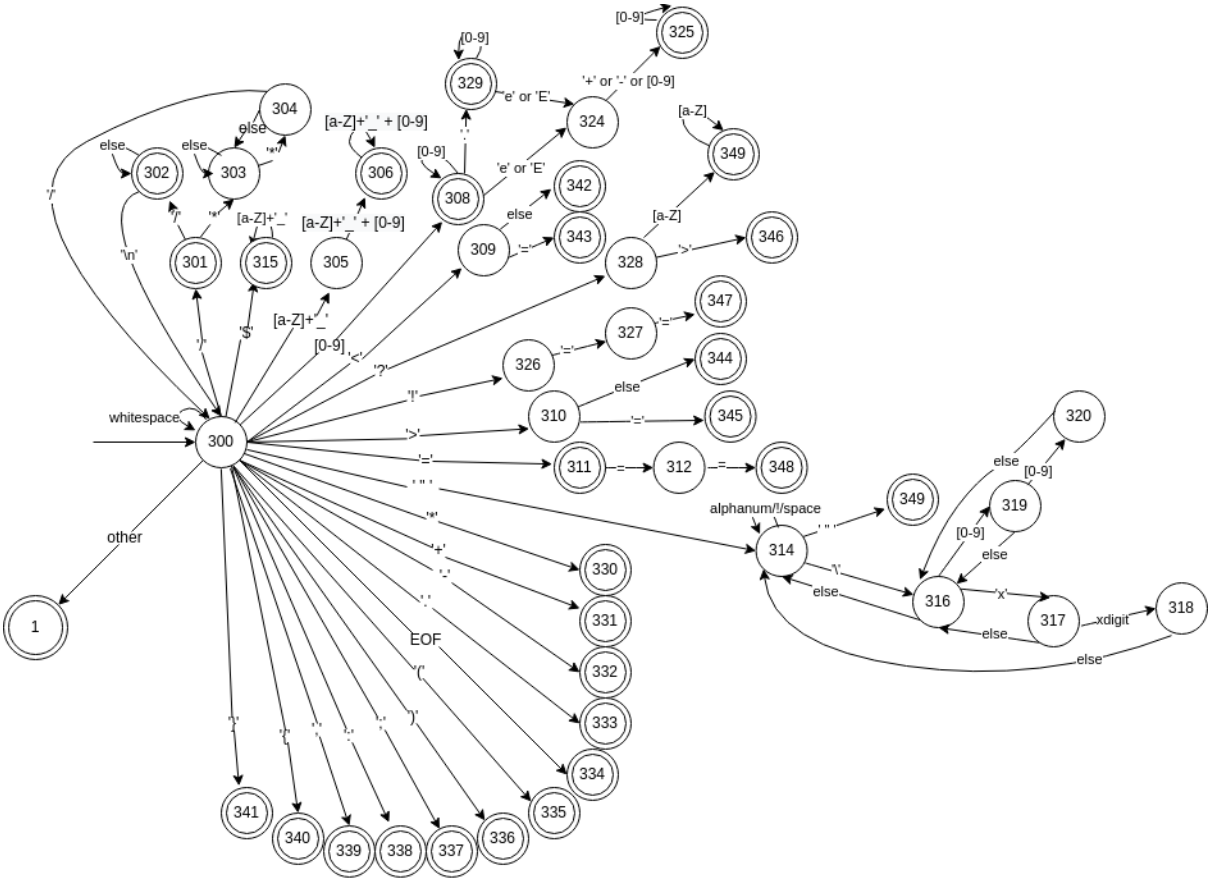
Generování kódu **IFJcode22** probíhá v souborech **parser.c** a **expressions.c.** Výsledný kód je generován zároveň se Syntaktickou Analýzou, tzn. při chybě ve vstupním programu se vypíše instrukce s příslušnou chybou. Celý finální kód se vypisuje na standardní výstup. Generátor kódu používá pomocné proměnné pro správné přiřazení na odpovídající proměnnou v programu. U generování kódu v souboru **expressions.c** se používají dvě pomocné proměnné **&expTmp1** a **&expTmp2** pro počítání výrazů řídících se precedenční tabulkou. V případech, kde se vyskytují vnořené podmínky, či vnořené cykly, se používají pomocné proměnné **uniqueIf** a **uniqueWhile**, pro správné pojmenování jednotlivých návěští. Generování instrukcí s následným přiřazením výsledného výrazu nebo návratové

hodnoty funkce na konkrétní výslednou proměnnou se provádí s pomocí pomocné proměnné **activeString**, na kterou se zpětně odkazuje daná instrukce. Jednotlivé instrukce jsou definovány v souboru **parser.h** jako globální proměnné.

2.6 Tabulka symbolů

Pro implementaci tabulky symbolů jsme zvolili binární vyhledávací strom, protože nám přišel snadnější na implementaci. Pro jeho implementaci jsme používali znalosti z předmětu IAL. Operace nad stromem jsme volali rekurzivně. V našem projektě jsme používali 3 binárni stromy, strom pro lokální proměnné, globální proměnné a funkce. Lokální proměnné se vždy po ukončení funkce odstraní. Kořeny stromů jsou uloženy v strukturách TRoot (pro proměnné) a TRoof (pro funkce). Jednotlivé uzly jsou uložené v struktuře TNode (pro proměnné) a TNodef (pro funkce). Struktura TNode obsahuje *content, který obsahuje název proměnné, *leftPtr a *rightPtr odkazují na pravého a levého syna, type obsahuje typ daná proměnné a declared nám sděluje, zda proměnná již byla definovaná. Struktura TNodef obsahuje *content, kde se nachází název funkce, *leftPtr a *rightPtr, které obsahují pravého a levého syna, proměnou parameters, která obsahuje počet argumentů dané funkce a proměnnou return_type, která obsahuje návratový typ funkce.

2.6 konečný automat lexikální analýzy:



stavy:

typeIdentifierState/type token	299	xponentPlusOrMinusState	324
basicState	300	endExponentState/exponent	325
possibleCommentState/token divide	301	notEqualState	326
oneLineCommentState	302	notEqualStateEnd	327
blockCommentState	303	epilogStateOrType	328
waitForBlockCommentEnd	304	decimalEndState/float number	329
keywordOrIdentifierStateBegin	305	token multiply	330
keywordOrIdentifierState	306	token addition	331
numberState/int number token	308	token subtraction	332
smallerThanOrEqualState	309	token decimal point	333
greaterThanOrEqualState	310	token EOF	334
assignOrEqualState/assign token	311	token left bracket	335
equalState	312	token right bracket	336
waitForStringEnd	314	token semicolon	337
variableRead	315	token colon	338
backslashState	316	token comma	339
escapeHexaState	317	token right vinculum	340
endHexaState	318	token left vinculum	341
escapeOctaState	319	token smaller than	342
waitOctaState	320	token smaller ot equal	343
decimalState	322	token string	349
token epilog	346	token not equal	347
token greater than	344	token greater or equal	345
token equal	348	Lexical error	1

pokud přijde symbol, který není definován v cestě, vrací se LEX ERROR

2.7. kdo pracoval na jakém souboru

kdo pracoval na čem:
lexical.c/lexical.h: xpejch08
str.c/str.h: *xpejch08*
parser.c/parser.h: *xpejch08, xsocha03, xniesl00*
main.c: xpejch08
makefile: xpejch08
expression.c/expression.h:xniesl00, xceska07
espstack.c/expstack.h: xpejch08, xceska07
symtable.c/symtable.h: xpejch08, xsocha03