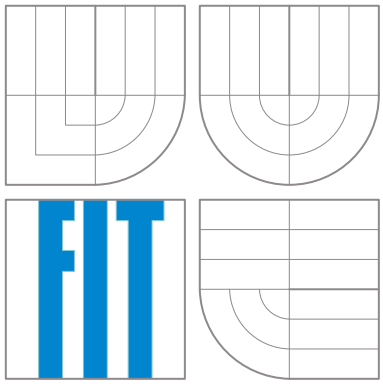


Dokumentace k projektu do předmětů IFJ a IAL

Implementace interpretu imperativního jazyka IFJ13.

15. prosince 2013

Varianta:	a/3/II
Rozšíření:	MINUS ELSEIF FOR
Vedoucí:	David Kovařík (xkovar66), 20%
Autoři:	Jonáš Holcner (xholcn01), 20%
	Marek Sychra (xsychr05), 20%
	Josef Karásek (xkaras27), 20%
	Benjamin Král (xkralb00), 20%

Obsah

1 Úvod

Jazyk IFJ13 je imperativní jazyk kombinující mnohá programovací paradigmata. Je podmnožinou jazyka PHP, přičemž některé vlastnosti byly z různých důvodů upraveny či omezeny. Tato dokumentace popisuje vývojový cyklus interpretu tohoto jazyka, včeteně popisu jednotlivých interních návrhů, mechanismů a rozhraní jednotlivých částí, ze kterých se celkový produkt sestává.

Jednotlivé části interpretu jsou popsány v pořadí, které odpovídá jejich logickému a časovému začlenění zhlediska struktury interpretu a průběhu jeho vývoje.

2 Analýza projektu

Dříve než bylo možné zahájit jakékoliv práce, bylo potřeba provést důkladné prostudování požadavků a složitosti zadání. Z hlediska počtu členů pracovního týmu se projekt řadí do kategorie středně velkých projektů. Bylo nám tedy již předem známo, že režie a plánování budou tvořit významnou část celkové práce.

Základním požadavkem tedy bylo, aby se všichni členové týmu důkladně seznámili se zadáním. Zde jsme narazili na první problém, neboť naše znalosti v této fázi vývoje pokrývaly minimum informací potřebných pro práci. Proto studování zadání mělo význam spíše pro vytvoření globální představy.

Byly stanoveny komunikační kanály a formáty jednotlivých dokumentů, jako například požadavek na opravu chyby, dokumentace kódu… Nicméně není vhodné považovat prvotní konvence za nějaká dogmata. Díky zvyšování povědomí o projektu, prohlubování znalostí a získávání zkušeností z dosavadního vývoje byly jednotlivé konvence postupně „optimalizovány“ pro aktuální požadavky.

3 Struktura projektu

Teoretický model rozděluje překladače až do šesti fází. My jsme si ovšem vybrali model více používáný v praxi, tedy **syntaxí řízený překlad**. Ten slučuje některé logické části do jedné a vytváří jednotný řídící uzel celého překladu.

3.1 Lexikální analyzátor – scanner

Úkolem této jednotky je zpracovat dokument po lexikálně stránce. Tedy rozpoznat jednotlivé lexémy daného jazyka a převést je na ekvivalentní vnitřní reprezentaci pomocí tokenů. Jádrem lexikálního analyzátoru je deterministický konečný automat, který načítá zdrojový soubor a rozpoznává lexémy. Schéma konečného automatu naleznete v příloze A.

Úkolem scanneru je také provádět expanze tzv. *escape sekvencí*. Jedná se o znaky, které kvůli jejich speciálnímu významu nelze zapsat do řetězce, ale musí být nejdříve uvozeny znakem „\“.

Průchod zdrojovým souborem je přerušen v případě, kdy se automat nachází v koncovém stavu a pro aktuálně načtený znak nemá definováno pravidlo přechodu. V takovém případě automat přijme načtený řetězec. Nesmí ovšem dojít k zahození znaku, který o přijetí řetězce rozhodnul, je tedy nutné tento znak vrátit zpět, aby byl při dalším čtení opět načten.

Lexikální analyzátor musí být schopen jednoznačně rozpoznat klíčová slova jazyka. Jelikož je jejich stavba identická se stavbou identifikátorů (ne identifikátorů proměnných) nelze je

pomocí čistého konečného automatu rozlišit. Proto jsme model konečného automatu museli částečně porušit. V momentě, kdy automat rozpozná identifikátor a je připraven jej přijmout, zkontroluje seznam klíčových slov, zda načtený řetězec neodpovídá některé z položek. Pokud ano, je automatu „vnucen“ stav odpovídající danému klíčovému slovu a řetězec je následně přijat. Podobný postup jsme použili také při expanizi *escape sekvencí* znaků zapsaných pomocí hexadecimálního čísla.

3.2 Syntaktická a sémantická analýza – parser

Pro syntaktickou analýzu jsme využili dva různé přístupy, v oblastech, kde přináší vyšší užitek daná metoda oproti jiným.

Tou je kontrola vícenásobné definice funkce. V interpretu existuje globální tabulka funkcí. Pokud parser při své práci narazí na definici funkce, zavede potřebné informace o ní do tabulky, včetně adresy, na které začíná na instrukční pásce. Pokud narazí na opětovnou definici téže funkce, zjistí, že ji již má v tabulce funkcí zavedenou, což nutně implikuje pokus o redefinici funkce.

Metoda rekurzivního sestupu je jednou z metod syntaktické analýzy shora dolů. Je vhodná pro zkoumání struktury programu. Metoda pracuje na základě LL gramatiky, která je také součástí příloh B. Kvůli nutnosti odstranění levé rekurze gramatiky však není vhodná pro zpracování výrazů. Pro úspěšnou kontrolu syntaxe programu je nutné, aby gramatika pokrývala všechny vlastnosti jazyka IFJ13.

Precedenční syntaktická analýza je metoda syntaktické analýzy zdola nahoru. U této metody je situace opačná, jelikož je vhodná spíše pro zpracování výrazů (levá rekurze nepředstavuje problém), nikoliv pro zkoumání kostry programu. Metoda se řídí statickou precedenční tabulkou, kterou jsme sestavili na základě znalostí o podporovaných operátorech a jiných elementech, které se ve výrazech mohou vyskytovat.

3.3 Sémantické akce

Myšlenkou syntaxí řízeného překladu je přidružení sémantických akcí k jednotlivým gramatickým pravidlům. V našem případě provádění sémantických akcí představuje generování tříadresného kódu, který je funkčně ekvivalentní ke zdrojovému programu.

Ze sémantických chyb jsme v době překladu schopni rozpoznat pouze vícenásobnou definici funkcí a to díky existenci globální tabulky funkcí 4.2. Ostatní chyby lze rozpoznat až v době běhu programu.

Vedle generování tříadresného kódu je neméně podstatné také rozpoznávání konstant a jejich závádění do vnitřních tabulky (pod unikátním názvem). Přestože se jedná o konstanty, jsou interně reprezentovány jako standardní proměnné. Většina instrukcí tedy očekává operandy typu řetězec. Ten obsahuje identifikátor, pod kterým v tabulce symbolů nalezneme požadovanou hodnotu. Tím jsme byli schopni redukovat instrukční sadu, ovšem za předpokladu častějšího vyhledávání v tabulce symbolů.

3.4 Instrukční sada

Tříadresný kód je implementován jako sekvence instrukcí z navržené instrukční sady. Pro vnitřní reprezentaci celé instrukční pásky jsme použili jednosměrně vázaný lineární seznam. Nicméně jsme museli tento model částečně degradovat. Linearita průchodu je narušena schopností seznamu nastavit svůj aktivní prvek (čtecí hlavu) na libovolnou instrukci kdekoliv na instrukční pásce. Tato vlastnost nám umožní rychle provádět instrukce skoku, které jsou klíčové pro implementaci volání funkcí, podmíněných příkazů a cyklů.

Míra abstrakce jednolivých instrukcí je značně nižší, než byla u elementárních jednotek zdrojového programu, nicméně je stále podstatně vyšší, než abstrakce instrukcí procesoru.

Instrukční sadu jsme navrhli až po důkladné analýze všech možných požadavků a jejich rozboru na elementární akce. Instrukce tedy představují právě tyto elementární akce. Z globálního hlediska byl dbán důraz na snadnou rozšiřitelnost instrukční sady v případě změny (rozšíření) požadavků.

4 Interpretace kódu

Interpretace kódu představuje průchod celou instrukční páskou a provádění jednotlivých instrukcí. Aktivní prvek seznamu instrukcí představuje aktuálně prováděnou instrukci tzv. *instruction pointer*. Po vykonání instrukce je *instruction pointer* posunut na další instrukci, která bezprostředně následuje za aktuální. U instrukcí skoku je následující instrukce určena jejich operandem.

Interpret je implementován nekonečným cyklem, který načítá a dekóduje jednotlivé instrukce. Každé instrukci pak odpovídá určitá sekvence operací v jazyce C (parametrizovaná operandy instrukcí). Cyklus interpretace je ukončen na základě charakteru provedené operace (například speciální instrukce značící konec instrukční pásky), nebo pokud při jejich provádění došlo k běhové chybě. V takovém případě návratová hodnota interpretu identifikuje chybu, ke které došlo.

4.1 Tabulka symbolů

Součástí interpetu je tabulka symbolů, do které jsou ukládány hodnoty proměnných podle jejich jména. Pro proměnné jsme vytvořili speciální datový typ, který zapoudřuje všechny datové typy (a speciální typ pro funkce), které IFJ13 podporuje, s potenciálem tento typ měnit.

Tabulku symbolů jsme implementovali pomocí *hashovací tabulky*. Rychlost vyhledávání je závislá na velikosti tabulky a kvalitě hashovací funkce, která převádí klíče záznamů na celá čísla, tedy indexy. Při vhodné hashovací funkci a velikosti je tedy možné vyhledávat v tabulce velmi efektivně. V ideálním případě, kdy je každý klíč převeden na unikátní index, je složitost vyhledávání konstantní (O(1)). V případě kolize indexů je vyhledávání degradováno na procházení lineárního seznamu (O(n)). Při vývoji jsme vypozorovali, že tabulka s velikostí prvočísla vykazuje lepší vlastnosti pro rozdělení indexů.

Rozhraní tabulky symbolů představují mimo jiné operace pro vkládání a vyhledávání prvků. Operace rušení prvků nebylo třeba implementovat. Pro potřeby vývoje a ladění jsme implementovali také operaci `foreach` pro průchod všech prvků tabulky.

4.2 Tabulka funkcí

Jedná se o speciální instanci tabulky symbolů. Její položky popisují pouze funkce, nikoliv proměnné. Položky uchovávají informace o počtu formálních parametrů, jejich názvech a ukazatel na první instrukci funkce. Tato tabulka je globální a je vytvářena již v době překladu. Pokud v ní některá z instrukcí vyhledává v době běhu a položku nenalezne, nebyla daná funkce definována.

4.3 Tabulka symbolů za běhu

Jazyk IFJ13 definuje veškeré proměnné jako lokální, čímž omezuje jejich rozsah platnosti. Není tedy možné využívat jedinou tabulku symbolů pro všechny úrovně zanoření. Aby interpret umožňoval volání uživatelem definovaných funkcí, včetně rukurze, implementovali jsme celý zásobník tabulek symbolů, který se dynamicky mění. Vrchol zásobníku pak představuje aktuální rozsah platnosti.

Na základě tohoto návrhu jsme značně omezili povědomí jednotlivých instrukcí o svém okolí. Instrukce tedy nemá žádnou možnost zjistit, v jakém rozsahu platnosti se pohybuje. Jediným bodem jejich interakce s okolím je tabulka na vrcholu zásobníku. Mohou v ní měnit hodnoty, nemají ovšem žádnou možnost měnit pořadí na zásobníku. Z toho vyplývá, že instrukce vyhledává své operandy právě na vrcholu tabulky zásobníků (pokud je operandem konstanta, hledá ji v samostatném bloku, ten je přístupný ze všech rozsahů platnosti).

4.4 Volání funkcí

Jediné dvě instrukce schopné měnit vrchol zásobníku tabulek symbolů jsou instrukce volání funkce a návratu z nich. Operandem instrukce volání funkce je identifikátor volané funkce. Na základě identifikátoru se v tabulce funkcí nalezne „objekt“ funkce. Ten uchovává informace o počtu a názvech formálních parametrů a ukazatel na první instrukci funkce na instrukční pásce.

Při volání funkcí je také potřeba zpracovat její argumenty. K tomu slouží fronta argumentů, do které se nahrají hodnoty všech argumentů, které se při volání funkce vyskytnuly. Pro vkládání do fronty slouží speciální instrukce. Na základě stanovené konvence po sobě vždy uklízí volaná funkce sama, nebylo tedy potřeba implementovat instrukci pro výběr z fronty. Abstraktní datový typ fronta jsme použili pro zachování pořadí argumentů.

Instrukce nyní vytvoří novou tabulku symbolů na vrcholu zásobníku a zavede do ní položky pro všechna jména formálních parametrů, kterým přiřadí hodnotu na základě fronty argumentů. Pokud počet prvků fronty je menší než počet formálních parametrů, dojde k chybě a ukončení interpretace. Je nutné, aby v tuto chvíli byla fronta prázdná (je společná pro všechna volání), proto je v případě nadbytečných argumentů vyprázdněna.

V tabulce symbolů vystupují dvě speciální položky. Do jedné z nich instrukce volání funkce uloží svou adresu na instrukční pásce, aby byla schopná vrátit řízení zpět. Další speciální položka je určena pro ukládání návratových hodnot dalších volaných funkcí.

Dále následuje samotné tělo funkce. Na jeho konci je vygenerována instrukce pro vrácení řízení volajícímu této funkce. Instrukce si načte návratovou hodnotu celé funkce. Odebere tabulku na vrcholu zásobníku. Pokud je zásobník prádný, nastal konec programu, jinak vloží návratovou hodnotu do tabulky, která se nyní nachází na vrcholu zásobníku. Nakonec provede nepodmíněný skok na instrukci, která bezprostředně následuje za instrukcí volání této funkce.

4.4.1 Vestavěné funkce

Vestavěné funkce vystupují jako černé skříňky, tvářící se jako uživatelské funkce. Dokonce přijímají argumenty stejným mechanismem. Ovšem namísto instrukce volání funkce je zavolána speciální instrukce korespondující dané vestavěné funkce.

Instrukce načte argumenty, vyprázdní frontu, provede požadovanou operaci a výsledek uloží do **aktuální** tabulky symbolů do položky pro návratové hodnoty. Tímto přístupem přesně simulujeme provádění funkce, ovšem bez nutnosti jediného skoku.

Funkce `sort_string` je vestavěná funkce určená pro řazení znaků v řetězci na základě ordinálních hodnot jednotlivých znaků pomocí algoritmu *shell sort*. Algoritmus vychází z principů *insertion sortu*. Rozdíl je v tom, že *shellsort* ve svém vnitřním cyklu neprochází polem sekvenčně, ale po skocích, jejichž délka je určena přírustkem h, který vypočteme $h = 3x + 1$, pro $h < delka\_pole$. Složitost algoritmu při použití takto vypočteného přírustku je v nejhorším případě $O(N^{\frac{3}{2}})$. Po vypočtení přírustku započne samotné řazení.

První navštívená hodnota v řazeném poli je hodnota s indexem h. Poté je spuštěn vnitřní cyklus, jenž prochází polem v opačném směru po skocích $j = j - h$ a provádí výměnu prvku s indexem h za prvek s menší hodnotou. Cyklus končí, jakmile je hodnota j menší než přírustek h. Po každém průchodu řazeným polem se vypočte nová hodnota přírustku $h = h/3$. Pokud platí, že $h < 1$, tak byly navštíveny všechny hodnoty řazeného pole a řazení tedy může skončit.

Funkce `find_string` je další vestavěnou funkcí určenou pro vyhledání podřetězce v zadaném řetězci. Pro samotné vyhledávání implementuje funkci Knuth-Morris-Prattův algoritmus.

Algoritmus pracuje na principu konečného automatu, jehož uzly představují úspěšné přečtení určité posloupnosti z hledaného řetězce. Pokud se automat dostane do koncového stavu, byl hledaný podřetězec nalezen. Hlavní výhoda se objevuje ve chvíli, kdy daný řetězec obsahuje určité opakující se sekvence znaků. Při čtení neodpovídajícího znaku není třeba, aby se automat vrátil do počátečního stavu.

Nejdříve se projde celý prohledávaný řetězec a vytvoří se pole indexů, které udává, do kterého stavu automat přejde v případě, že byl načten nevhodný znak. Tento úkon spočívá v procházení řetězce a porovnávání shodných částí. Dále se přejde k samotnému vyhledávání. Prohledávaný řetězec je načítán znak po znaku. V případě čtení hledané posloupnost se automat postupně přesunuje „směřem ke koncovému stavu“. Při přečtení nevhodného znaku se vrátí do nejvhodnějšího stavu „směrem zpět“. Do kterého stavu udává vektor indexů z první části algoritmu.

Celková složitost algoritmu je pak $O(m + n)$, kde m je složitost sestavení automatu a n samotného procházení..

4.5 Implementovaná rozšíření

4.5.1 Cyklus `for`

Hlavička cyklu se dělí na 3 sekce, přičemž každá z nich může být prázdná. Z hlediska generování instrukcí je hlavička zásadní, protože definuje návěští, na která se během iterací cyklu provádí skoky. Součástí těla cyklu mohou být také příkazy `break` (ukončí cyklus) a `continue` (okamžitě skočí na další iteraci). Gramatika jazyka umožňuje zápis těchto příkazů do libovolného bloku programu. Přípustný je ovšem jejich výskyt pouze v těle cyklu `while`. Pokud se tedy některý z nich nachází jinde, nastane (za běhu programu), sémantická chyba.

Jelikož mohou být cykly libovolně zanořeny do sebe, je nutné, aby byl každý cyklus schopen jednoznačně určit, na která návěští má provádět skok. Nesmí nastat situace, že by cyklus provedl skok na návěští nadřazeného cyklu. Tento problém je řešen pomocí tzv. zásobníku cyklů. Položky zásobníku uchovájí adresy návěští pro skok příkazů `break` a `continue`. Při vstupu (během překladu) se na zásobník přidá nová položka, při výstupu se vrchol odebere. Tento problém by bylo možné řešit také pomocí rekurzivního charakteru syntaktické analýzy. Tak je ovšem řešena konstrukce `if-elseif-else`, proto jsme v tomto případě zvolili jiný postup.

4.5.2 Rozšířený podmíněný příkaz

Rozšířený podmíněný příkaz umoňuje libovolný počet výskytů speciálních bloků `elseif` a případnou absenci bloku `else` na konci podmíněného příkazu. Tím pádem může při vícenásobném vnoření těchto příkazů může vzniknout sémantická nejednoznačnost, ke kterému bloku `if` patří blok `else`. Logicky blok `else` náleží „předcházejícímu“ bloku `if`. Pomocí zásobníku jsme schopni jednoznačně rozlišit aktuálně zpracovávány blok a instrukcím skoku přiřazovat správné adresy návěští. Zásobník ovšem není nutné implementovat, využili jsme rekurzivního charakteru metody syntaktické analýzy, která zásobník simuluje.

4.5.3 Unární minus

Operace unární minus je z naší množiny aritmetických operací jediná unární a pravě asociativní s nejvšší prioritou. Výrazy jsou zpracovávány precedenční syntaktickou analýzou řízenou precedenční tabulkou. Bylo tedy potřeba přidat do tabulky řádek a sloupec pro tento operátor a správně doplnit položky tabulky.

Lexikálně jsou unární a binární minus identické. Je tedy potřeba, na základě kontextu, rozpoznat, o jaký operátor se jedná. Unární minus se od binárního liší svou pozicí, tedy pokud se před lexémem minus **nenachází** operand nebo prává uzavírací závorka, jedná se o unární minus. Pro jeho implementaci byla rozšířena instrukční sada interpretu o instrukci znaménkové negace. Je velmi důležité, aby instrukce v případě změny znaménka proměnné neměnila hodnotu přímo, ale vytvořila pomocnou proměnnou, se kterou se dále pracuje. Proměnné mohou měnit svou hodnotu pouze během operace přiřazení a jiný postup by měl za následek neočekávané chování při vyhodnocování výrazů.

5 Vývojový cyklus

Jak již bylo zmíněno, není možné na začátku vývojového cyklu odhalit všechna uskalí, na která tým při své práci narazí, proto jsme první pracovní dny nevěnovali přímé práci na částech interpretu, ale na doprovodných, však neméně důležitých, činnostech.

První kroky vedly směrem k „nastavení pracovního prostředí“. Tím je myšleno stanovení komunikačních prostředků pro spolupráci celého týmu, výběr systému pro správu verzí a stanovení konvencí během vývoje. Tyto konvence se vztahovaly například ke komentování (v českém jazyce) zdrojového kódu v optimálním množství tak, aby libovolný člen, který se k danému úkolu připojí, nebo jej převezme, byl schopen v nejkratším možné čase maximálně porozumět již implementovanému nebo rozpracovanému řešení.

Jakmile bylo „pracovní prostředí“ nastaveno, došlo již k samotnému programování. Nejdříve jsme ovšem vytvořili pomocné modulů, u kterých bylo již na začátku zřejmé, že budou potřeba.

Jednalo se například o modul pro práci s textovými řetězci, zpracování chybových stavů programu a modul pro správu paměti umožňující lokální a globální alokaci bez potřeby explictního uvolnění.

Následně jsme se dostali do fáze, kdy již bylo téměř vše připraveno pro práci, ale neměli jsme dostatečné teoretické znalosti. Jelikož úkolem vedoucího bylo navrhnout a předložit celkovou myšlenku, návrh a plán produktu, bylo potřeba, aby nejdřív podrobněji nastudoval danou problematiku. Následně navrhl potřebné součásti dané sekce, seznámil tým se základními principy (případně vazbou na budoucí části). Dál již byla týmu ponechán prostor pro projevení vlastní kreativity a originality při řešení. Nicméně dodržení plánu a návrhu bylo klíčové pro snadné propojování s již existujícími komponentami. Jako příklad je možné uvést navržení modelu konečného automatu, vytvoření gramatiky jazyka, sestavení precedenční tabulky nebo návrh instrukční sady včetně sémantiky jednotlivých instrukcí.

V průběhu vývoje jsme následně narazili na problém koroordinování komunikace více členů, kteří si aktivně podílí na samotném vývoji. Snížili jsme tedy počet členů aktivních ve vývoji ze čtyř na tři. Uvolněný člen byl pověřen důkladným testování. Současně byla členům testovácí části týmu přidělena pravomoc zasahovat do samotného jádra programu a provádět jednoduché a zřetelné změny. Těmito kroky jsme výrazně snížili režii nutnou na koordinaci komunikace členů a navíc jsme zkrátili čas nutný k opravě drobných chyb. Těch průběžné testování totiž odhalilo velké množství a jejich hlášení a čekání na odezvu by pak bylo příliš neefektivní.

6 Orientační rozdělení úkolů

- David Kovařík (vedoucí)
  - Plánování a koordinace týmu, dokumentování práce
  - Návrh lexikálního analyzáru, gramatiky jazyka, instrukční sady, sémantických akcí . . .
  - Implementace syntaktické analýzy struktury programu, cyklus `for`
- Jonáš Holcner
  - Zpracování výrazů (syntaktické i sémantické), unární minus
  - Implementace sémantických kontrol a generování kódu a vybraných instrukcí
- Marek Sychra
  - Implementace interpretu, většiny instrukcí a rozšířený podmíněný příkaz
  - Další sémantické akce a kontroly
- Josef Karásek
  - Vytvoření automatizované sady testů, kontrola integrity a kvality
  - Implementace pomocných abstraktních datových typů, testování, ladění
- Benjamin Král
  - Lexikální analyzátor
  - Aktivní testování a hlášení chyb

7 Závěr

Projekt svým rozsahem pokrývá celé spektrum činností, které je pro úspěšné řešení projektů v praxi klíčové. Navíc bylo nutné zvládnout velké množství teoretických informací a vhodně je převést do praxe. Kvůli počtu členů týmu bylo důležité svědomitě provádět (a respektovat) manažerské práce jako rozdávání úkolů, přebírání výsledků a plánování. Tím ovšem režie nutná pro vývoj nekončí. Jelikož tým představuje živý organismus, je nezbytně nutné, aby si členové vzájemně naslouchali a případně upravovali již stanovené plány. V tak citlivém odvětví jako je vývoj softwaru většinou není autokratický přístup vedoucího příliš efektivní. Také není možné, aby členové týmu dogmaticky prováděli zadané úkoly, aniž by nebyli schopni jej v opodstatněných případech připomínkovat a upravit.

Přes všechna tato úskalí jsme se společně, jako tým, dostali a na výstupu naší práce byl funkční interpret jazyka IFJ13. Díky dobrému časovému zvládnutí projektu jsme byli schopní implementovat několik rozšíření, která syntaxi a sémantiku jazyka uvedla téměř na úroveň jazyků běžně používaných. Nejvýznamější ovšem byly zkušenosti, které všichni členové týmu získali.

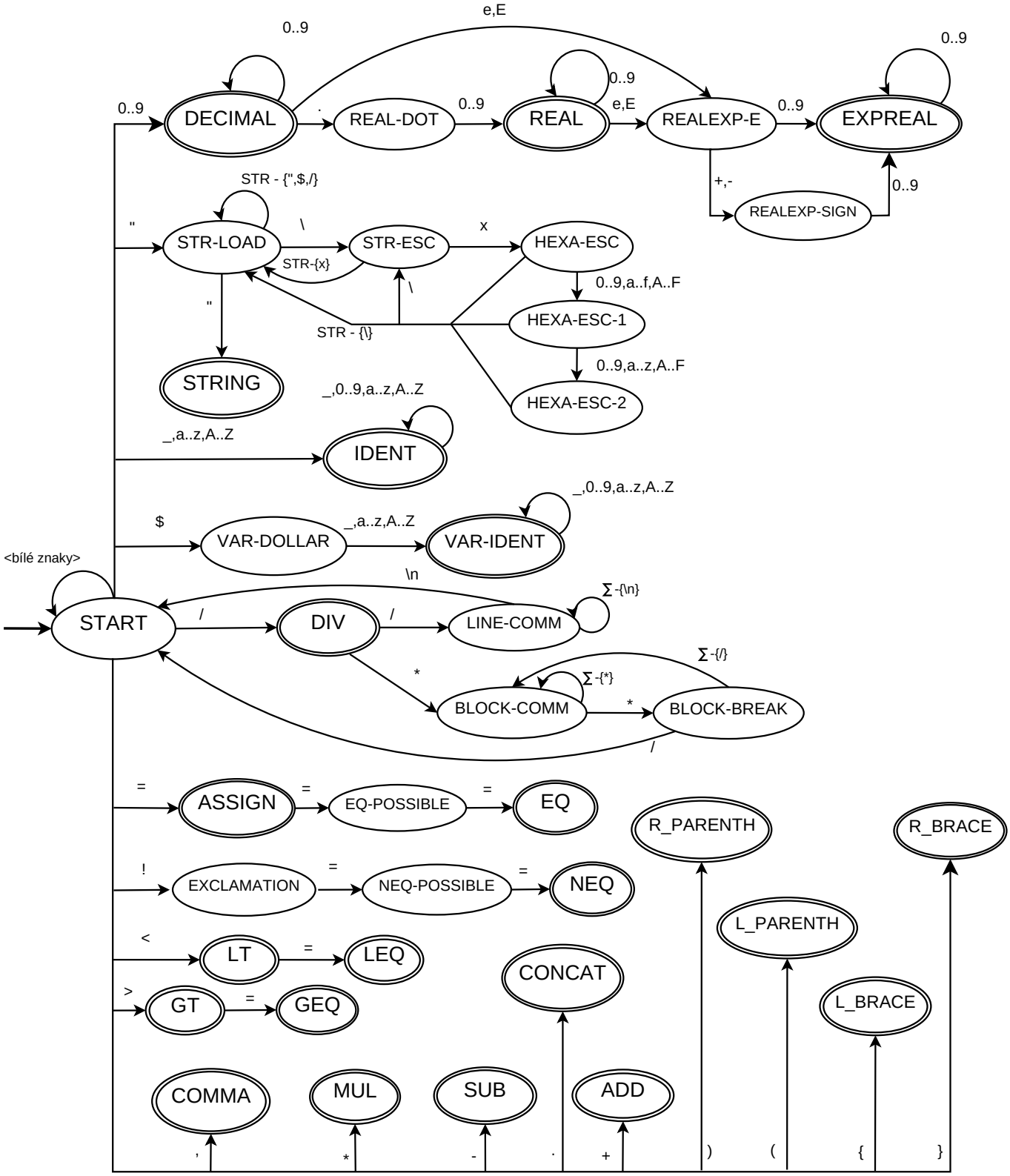
Reference

[1] *PHP Manual* [online], URL: <http://www.php.net/manual/> [cit. 15. prosince 2013]

[2] *MEDUNA, Alexander* Elements of compiler design. Boca Raton: Auerbach Publications, 2008, xiii, 286 s. ISBN 1-4200-6323-5.

[3] HONZÍK, J. M.: *Opora IAL* [online]. Brno: Fakulta informačních technologií VUT v Brně, 2012, rev. 12-L.

A Konečný automat lexikálního analyzátoru



Množina **STR** znační množinu všech znaků, které se mohou vyskytovat v řetězci.

Obrázek 1: Konečný automat lexikálního analyzátoru

B LL gramatika

1: <program> ⟶ **<?php** <program-body> **eof**
2: <program-body> ⟶ $\mathcal{E}$
3: <program-body> ⟶<statement> <program-body>
4: <program-body> ⟶<function> <program-body>

5: <function> ⟶ **function id (**<params> **) { **<inner-block> **}**
6: <params> ⟶ **var-id** <params-next>
7: <params> ⟶ $\mathcal{E}$
8: <params-next> ⟶ **,** **var-id**<params-next>
9: <params-next> ⟶ $\mathcal{E}$

10: <inner-block> ⟶<statement><inner-block>
11: <inner-block> ⟶ $\mathcal{E}$
12: <statement> ⟶ **if (E) {** <inner-block> **}** <if-block>
13: <statement> ⟶ **while (E) {**<inner-block> **}**
14: <statement> ⟶ **return E**
15: <statement> ⟶ **var-id =**<value> **;**
16: <statement> ⟶ **for (**<for-assign> **;** <for-cond> **;**<for-assign> **) {**<inner-block> **}**
17: <statement> ⟶ **break ;**
18: <statement> ⟶ **continue ;**

19: <for-assign> ⟶ **var-id = E**
20: <for-assign> ⟶ $\mathcal{E}$
21: <for-cond> ⟶ **E**
22: <for-cond> ⟶ $\mathcal{E}$

23: <if-block> ⟶ **elseif (E) {**<inner-block> **}**<if-block>
24: <if-block> ⟶ **else {** <inner-block> **}**
25: <if-block> ⟶ $\mathcal{E}$

26: <value> ⟶ **E**
27: <value> ⟶ **function (**<args> **)**

28: <args> ⟶<term><args-next>
29: <args> ⟶ $\mathcal{E}$
30: <args-next> ⟶ **,**<term> <args-next>
31: <args-next> ⟶ $\mathcal{E}$

32: <term> ⟶ **dec**
33: <term> ⟶ **real**
34: <term> ⟶ **realexp**
35: <term> ⟶ **string**
36: <term> ⟶ **true**
37: <term> ⟶ **false**
38: <term> ⟶ **null**
39: <term> ⟶ **var-id**

Speciální terminál **E** značí výraz. Ty jsou zpracovávány precedenční syntaktickou analýzou a nejsou tedy součástí LL gramatiky

C LL tabulka

	<?php	function	ident	(	)	{	}	var-id	,	if	expr	while	return	;	=	for	break	continue	elseif	else	ident	dec	real	realexp	string	true	false	null	$
<program>	1																												
<program-body>		4						3		3		3	3			3	3	3											2
<statement>								15		12		13	14			16	17	18											
<function>		5																											
<params>					7			6																					
<inner-block>							11	10		10		10	10			10	10	10											
<params-next>					9				8																				
<if-block>		25					25	25		25		25	25			25	25	25	23	24									
<value>											26										27								
<for-assign>					20			19						20															
<for-cond>											21			22															
<args>					29			28														28	28	28	28	28	28	28	
<term>								39														32	33	34	35	36	37	38	
<args-next>					31				30																				