Dokumentace k projektu pro předmět IFJ

Implementace interpretu imperativního jazyka IFJ15

13. prosince 2015

Tým 074, varianta β/1/II

xvelec07:	Jan Velecký (vedoucí)	30 %
xzahra24:	Tomáš Zahradníček	30 %
xpiece00:	Adam Piecek	20 %
xpavli78:	Jan Pavlica	20 %

Rozšíření:

```
BASE
WHILE
SIMPLE
BOOLOP
```


Fakulta Informačních Technologií
Vysoké Učení Technické v Brně

1 Obsah

1 Obsah ... 1
2 Úvod ... 2
3 Práce v týmu ... 2
4 Lexikální analyzátor ... 2
5 Analyzátor struktury programu ... 3
6 Tabulka symbolů ... 3
7 Analyzátor výrazů ... 4
8 Instrukční sada ... 5
9 Generátor instrukcí ... 6
10 Interpret ... 7
10.1 Práce s řetězci ... 7
11 Algoritmy IAL ... 8
11.1 Boyer-Mooreův algoritmus ... 8
11.2 QuickSort ... 8
11.3 Hashovací tabulka ... 8
12 Rozšíření ... 9
12.1 SIMPLE ... 9
12.2 WHILE ... 9
12.3 BASE ... 9
12.4 BOOLOP ... 9
13 Rozdělení práce ... 10
14 Metriky kódu ... 10
15 Příloha A: Konečný automat lexikálního analyzátoru ... 11
16 Příloha B: LL gramatika ... 12
17 Příloha C: Tabulka precedenční analýzy ... 13
18 Příloha D: Instrukční sada ... 14
18.1 Obecné instrukce: ... 14
18.2 Aritmetické/relační/logické instrukce: ... 14

2 Úvod

Tato dokumentace popisuje návrh a implementaci imperativního programovacího jazyka IFJ15, jakožto týmového projektu do předmětu Formální jazyky a překladače. Jazyk IFJ15 je podmnožinou jazyka C++. Projekt jsme si rozdělily do několika částí, kterým bude věnována samostatná kapitola:

- Lexikální analyzátor
- Syntaktická a sémantická analýza (bez výrazů) – Analyzátor struktury programu
- Syntaktická a sémantická analýza výrazů – Analyzátor výrazů
- Instrukční sada
- Interpret

3 Práce v týmu

Ještě před započetím práce na projektu jsme se dvakrát sešli zavčasu, abychom se seznámili, probrali postup řešení, domluvili se na společném komunikátoru a rozdělili si v práci týmu. Nadále jsme se již plánovaně nescházeli a začali pracovat na projektu, i když poměrně pozdě vzhledem k tomu, že práci jsme již měli dlouho rozdělenou brzy. Nicméně čekali jsme, až pokročí přednášky v předmětu IFJ. Původní koncepce se ukázala jako dobrá, protože ji nebylo potřeba nijak předělávat. Nicméně některé části projektu bylo i tak potřeba mnohokrát upravovat, aby fungovali správně. Pro komunikaci v týmu jsme používali komunikátor Skype a jako verzovací systém SVN na serveru assembla.com.

4 Lexikální analyzátor

Úkolem lexikálního analyzátoru je čtení zdrojového souboru v jazyce IFJ15 a následné zpracování na jednotlivé části – *lexémy*. Dalším úkolem je odstranění komentářů a vynechávání bílých znaků. Lexikální analyzátor je implementován formou konečného automatu. Diagram tohoto konečného automatu můžeme vidět v příloze.

Lexikální analyzátor je volán syntaktickým analyzátorem a to pokaždé, když požaduje nový lexém. V případě *bohatého lexému* (číslo, identifikátor, řetězec) vrací i dodatečné informace pomocí globálních proměnných. Při chybě způsobené nedodržením lexikálních pravidel je vrácena lexikální chyba. Při ostatních chybách (např. neúspěšná alokace) je vrácena interní chyba.

Samotný lexikální analyzátor čte vstupní soubor po znacích a dle daného znaku mění stav konečného automatu.

5 Analyzátor struktury programu

Analýza struktury programu je implementována metodou shora dolů rekurzivním sestupem. Vstupem této analýzy jsou tokeny, které vrací lexikální analyzátor. Výstupem je při chybě označení chyby, ke které při analýze došlo. Pokud nedošlo k žádné chybě, je výstupem instrukční páska pro interpret, jejíž posloupnost instrukcí odpovídá vstupnímu programu, a také globální tabulka symbolů obsahující informace o funkcích (např. místo potřebné pro uložení parametrů a proměnných funkce).

První funkcí analýzy je syntaktická analýza, která kontroluje správnou strukturu programu. Tato struktura je definovaná LL gramatikou. Další funkcí analýzy je sémantická analýza, ve které se s přístupem do tabulky symbolů kontroluje sémantická správnost programu, a následně se vygeneruje odpovídající kód do instrukční pásky.

Pokud analyzátor struktury programu očekává výraz (dostane se v LL gramatice k neterminálu EXPR), předává řízení analyzátoru výrazů. Po zpracování výrazu se řízení vrátí analýze struktury programu.

Při implementaci přiložené LL gramatiky se narazilo na 2 konflikty.

Prvním z nich je nejednoznačnost mezi pravidly 38 a 39. Pravidlo 39 očekává identifikátor, ten ale může být jako první obsažen i ve výrazu. Vyřešení této nejednoznačnosti je přístup do tabulky symbolů, a na základě sémantické hodnoty identifikátoru se rozhodne o použitém pravidlu. Pokud se jedná o identifikátor proměnné, musí se jednat o výraz a řízení je tedy předáno analýze výrazů (spolu s předáním již načteného tokenu id). Pokud se jedná o funkci, očekává se volání funkce, tedy pravidlo 39. Pokud identifikátor v tabulce symbolů není, jedná se v obou případech o chybu nedefinované proměnné nebo funkce.

Druhý z nich je nejednoznačnost mezi pravidlem 44 a 45, kde může existovat více derivačních stromů if/else struktur. Tato nejednoznačnost je vyřešena jednoduše použitím pravidla 44 pokud je to možné, takže se bude vždy else vztahovat k nejbližšímu předchozímu if.

6 Tabulka symbolů

Tabulka symbolů je implementována jako zásobníková struktura (ačkoliv přistupovat lze i k jiným položkám než jen na vrcholu zásobníku), kde jednotlivé položky jsou hashovací tabulky. Jednotlivé položky odpovídají vždy jednomu bloku ve zdrojovém kódu. Takto je vyřešena viditelnost proměnných ve vícenásobně vnořených blocích, protože tabulku symbolů lze prohledávat od nejvíce vnořeného bloku (hashovací tabulka na vrcholu zásobníku) a vrácené informace identifikátoru odpovídají první nalezené shodě.

První hashovací tabulka na zásobníku je globální tabulka symbolů (odpovídá bloku, kterým je celý program), která z definice jazyka může obsahovat jen funkce a k nim potřebné informace (jazyk nepodporuje globální proměnné). Ve všech dalších hashovacích tabulkách jsou jen informace potřebné pro proměnné (jazyk nepodporuje vnořené funkce). Po opuštění bloku při analýze struktury programu se odpovídající hashovací tabulka spolu se všemi informacemi o proměnných uvolní. Na konci analýzy programu je tedy jedinou hashovací tabulkou na zásobníku ta, která obsahuje informace o funkcích. K těmto informacím se přistupuje ještě v době interpretace programu.

7 Analyzátor výrazů

Analyzátor výrazů (AV) je volán analyzátorem programu (AP), pokud je v zápisu programu očekáván výraz. AV tedy přebírá řízení překladu od AP a vrací mu jej až po úspěšném zpracování celého výrazu. AV kontroluje správnost zápisu výrazu pomocí precedenční syntaktické analýzy, jíž simuluje tvorbu derivačního stromu. Dále je během analýzy výrazu kontrolováno, zda není sémanticky špatně – špatná kombinace datových typů operandů, použití nedeklarovaného identifikátoru a použití funkce jako proměnné. Během činnosti jsou samozřejmě také generovány příslušné instrukce odpovídající operacím provedeným ve výraze.

Při předání řízení překladu programu AV je mu předána také informace o výsledném datovém typu. V takovém případě AV zajistí i přetypování výsledku instrukce do správného datového typu. Předaný datový typ může být také speciální hodnota `NO_DATATYPE`. Toto použití AV slouží např. pro determinování datového typu proměnné `auto`.

Při zpracování příchozího lexému jsou převedeny všechny možné operandy (literály i proměnné) na terminál `i`, avšak je zachován příznak, zda se jedná o proměnnou, nebo literál. Lexémy, které nemůžou být ani operátorem ani operandem jsou převedeny na ukončující terminál `$`. Problémem je zde ukončující závorka ‚)‘, která může být jak součást výrazu, tak ukončující terminál, proto si AV při načítání lexémů jednoduchých závorek ukládá *aktuální hodnotu zanoření* a podle toho považuje ukončující závorku buď za operátor, nebo za `$`. Návratovou hodnotou AV je lexém, jenž je ukončujícím.

Precedenční analýza je tvořena precedenční tabulkou v. příloha – tabulka pro precedenční analýzu fungující nad jednoduchým zásobníkem, jehož položka je buď terminál, nebo neterminál a v obou případech sebou nese informaci o datovém typu, jelikož neterminál se v průběhu analýzy stává operandem. Neterminál jakožto operand může být proměnná, konstanta, nebo mezivýsledek, což je pouze dočasná, abstraktní proměnná použitá během vyhodnocování výrazu. V případě proměnné (i dočasné) si sebou nese i informaci o jejím indexu, v případě konstanty onu hodnotu.

Operace mezi dvěma konstantami je pak zderivována opět na neterminál typu konstanta. Dá se na to pohlížet jako na optimalizaci, nicméně je to nutné z hlediska instrukční sady. Výraz `a = 1 + 1;` tedy vygeneruje jedinou instrukci, která uloží do proměnné a hodnotu 2, výraz `a = 1 + 1 + a;` pak pouze jedinou instrukci sečtení (2 + a, resp. a + 2), nicméně výraz `a = a + 1 + 1;` z povahy zásobníkového řešení vygeneruje 2 instrukce, nikoliv jednu ((a + 1) + 1).

Sémantika operací je implementována v 3 (aritmetické, relační a logické operátory) souborech `*.inc.c` (sloužící jako víceřádkové makro), které jsou parametrizována konkrétním typem operace, jejím bajtkódem, dále tím zda je operace komutativní, jaký má (a jestli vůbec) neutrální a agresivní prvek a pro který prvek nemá definované chování (nula v děliteli). Tyto parametry slouží k dalším optimalizacím. Výraz `a = a * 1;` tak vede na přiřazení hodnoty 1. Při dělení konstantou nulou pak nedochází ke generování instrukce `DIV`, ale instrukce `ERROR`.

Výpočet výrazu interpretem simuluje zásobníkové zpracování podobně jako instrukce instrukční sady x87 FPU, nicméně je to výhodné, protože stejným způsobem funguje analýza výrazu. Nevýhodou je nutnost před započnutím výpočtu první operand načíst do paměťového místa vyhrazeného pro výpočet výrazu první operace na každé úrovni priority (dle priorit operátorů), jelikož adresa prvního operandu je zároveň místo, kde se ukládá výsledek operace. Pro výraz `(a+1)*(b+1)` tedy kromě 3

operací proběhnou i 2 načtení do dočasných proměnných. Pro výraz a+b+c+d by také proběhly 3 operace, avšak pouze jedno načtení.

Tabulky ilustrují průběh výpočtu v jednotlivých krocích pro 2 různé výrazy. Jeden řádek v tabulce je jeden krok a jeden sloupec jedna položka simulovaného zásobníku, tedy jedna dočasná proměnná. Modrá značí naplnění dočasné proměnné instrukcí LOAD, šedá nevyužité místo a oranžová výsledek, který je po dokončení výpočtu vždy na vrcholu zásobníku. Velikost zásobníku pro daný výraz je zjištěna během analýzy, při běhu tudíž tento imaginární zásobník představuje pouze vyhrazená část paměti výpočet jednoho konkrétního výrazu v rámci pro lokální proměnné prováděné funkce.

a
a+b
a+b+c
a+b+c+d

a		
a+1		
a+1	b	
a+1	b+1	
a+1	b+1	c
a+1	b+1	c+1
a+1	(b+1)*(c+1)	c+1
(a+1)*(b+1)*(c+1)	(b+1)*(c+1)	c+1

8 Instrukční sada

Bajtkódové instrukce Tinstruction pro náš interpret jsou 2operandové, podobně jako v instrukční sadě x86. První adresa značí obvykle první operand a určuje také cíl výsledku a druhá adresa druhý operand. Tinstruction má následující strukturu:

```
typedef struct {
    char      code;
    Toperand1  op1;
    Toperand   op2;
} Tinstruction;
```

```
typedef union {
    size_t index_size; // index proměnné
    void *        ptr;
    int       integer;
    double       real;
} Toperand;
```

První operand na rozdíl od druhého nemůže obsahovat přímou hodnotu typu double. V reálné instrukci stačí, když operace *pracuje pouze s jedním přímým operandem*, jelikož operace mezi dvěma konstantami je též konstanta. Je to tedy operace, která lze provést během elaborace výrazu a je zbytečné ji provádět až při interpretaci. Na 32bitových systémech nám to umožňuje mít menší[1] velikost instrukce a tím i menší velikost instrukční pásky. Naprostá většina instrukcí pak využívá oba operandy, zbývající alespoň jeden[2] – můžeme tedy hovořit, že bajtkód je *efektivně uložen v paměti*[3].

Jelikož může být přímý operand typu double pouze jako druhý operand instrukce, pro jednotnost napříč instrukcemi platí, že jakákoliv přímá hodnota je vždy v druhém operandu bez ohledu na datový typ. U nekomutativních aritmetických operací (odčítání a dělení) proto existuje ještě reverzní varianta instrukce, kde se operandy interpretují prohozeně. Zatímco výraz a/1.5 by byl převeden na normální instrukci, výraz 1.5/a na její reverzní variantu. U nekomutativního relačního operátoru je

[1] Velikost instrukce je 16 B na Windows x86 vs. 24 B na Linux x64.
[2] Kromě instrukce EOP, ta je však v každém programu jenom jedna.
[3] Délka instrukční pásky je pro referenční programy ze zadání: 10.1 faktoriál – 27, 10.2 rekurzivní faktoriál – 36, 10.3 vestavěné funkce – 39.

jeho reverzní varianta jiný relační operátor (`b > a` je totéž jako `a < b`), není tedy potřeba zvláštních reverzních instrukcí pro tyto operace.

Všechny instrukce s sebou v operačním znaku (kódu) nesou *informace o datových typech operandů,* čímž prakticky interpret odstiňují od rozhodování, nad jakými datovými typy pracuje. Také nese informaci o tom, zda je daný operand přímým operandem, nebo proměnnou. Důsledkem tohoto řešení je množství variant pro každou typovou instrukci. Zatímco instrukce CJMP pro podmíněný skok má 2 varianty (int a double; je tedy zakódována na 2 bitech); tak instrukce LOAD pro přesuny hodnot mezi proměnnými má 14 variant a je zakódována na 4 bitech (využívá tedy 14 z 16 možností) a je naší instrukcí s největším počtem variant.

Počet variant jsme nepatrně snížili tím, že instrukce s přímým operandem nesou informaci pouze o jednom datovém typu (výsledném) a konstanta je do správného typu případně zkonvertována během elaborace výrazu.[4] Celkově naše instrukční sada obsahuje 24 typů instrukcí a celkově 136 instrukcí se započtením všech variant, které jsou zakódovány na 1 B.

Seznam všech typů instrukcí a jejich zakódování jsou uvedeny v příloze instrukční sada.

8.1 Generátor instrukcí

V instrukční sadě interpretu je příliš mnoho instrukcí na to, aby analyzátor struktury programu a analyzátor výrazů vybírali správnou instrukci např. podle datových typů proměnných. Proto je implementován generátor instrukcí, který poskytuje abstrakci pro tyto analyzátory. Jeho vstupem je kromě základní instrukce a až 2 operandů také datové typy operandů a příznaky (např. zda se má nastavit příznak inicializace proměnné, nebo zda se má upravit počet referencí při práci s datovým typem string). Výstupem je zapsaná, potřebně modifikovaná instrukce na instrukční pásce. Další výhodou tohoto generátoru je, že řeší konverze (vygeneruje příslušnou instrukci, pokud je konverze potřeba), případně způsobí odpovídající chybu, pokud konverze není možná.

[4] Například konstanta 1 ve výrazu `double a = 1;` je zkonvertována na double již během elaborace. Z tohoto důvodu má např. instrukce LOAD jenom 14 variant, nikoliv 16.

9 Interpret

Interpret přímo provádí přeložený kód bez nutnosti kompilace. Je volán s těmito parametry: ukazatel do tabulky symbolů (funkce *main* při prvním zavolání), počet (celková velikost) rámce pro lokální proměnné (+ dočasné proměnné pro výrazy i návratové hodnoty funkcí) a odkaz do paměti pro navrácení návratové hodnoty. Lokální proměnné jsou tedy alokovány dynamicky na zásobníku a tato struktura umožňuje i rekurzivní volání funkcí, jelikož instrukce `CALL` je rekurzivní volání procedury interpretu s novými parametry. Tento rámec je implementačně pole obecného typu DOUBLEWORD (tedy pole dvouslov; int32_t prakticky), což je výhodnější než typ unie z pohledu paměťové náročnosti, protože proměnná typu int/bool bude zabírat pouze 4 B. Zároveň s tímto polem je ještě vygenerováno další pole bajtů stejné velikosti pro příznaky inicializace těchto proměnných.

Odkaz do tabulky symbolů je použit pro zjištění odkazu na první instrukci volané funkce, kterou je vždy instrukce `INIT`, která se ihned začne vykonávat bez dekódování. Operand instrukce obsahuje velikost předávaných parametrů, které jsou poté kopírovány z adresy pro návratovou hodnotu, kde volající funkce uložila předávané argumenty a je u nich nastaven příznak inicializace. U všech ostatních je naopak příznak inicializace nastaven na 0. Ve skutečnosti je tato instrukce relikt z raných verzí interpretu a nyní již není potřeba, jelikož se předává odkaz do tabulku symbolů.

Volání vestavěných funkcí je jednodušší a řešeno namapováním těchto funkcí na celé číslo a to je operandem instrukce `CALLBIN`. Zejména zde není nutné přesouvat proměnné mezi rámci funkcí.

Protože ARL (aritmetické, relační, logické) instrukce jsou implementovány zásobníkem, není nutné nikdy kontrolovat u prvního operandu, zda je inicializovaný, ani tento příznak nastavovat, neboť to již zajišťuje instrukce `LOAD`.

Jelikož je operační znak uložen na 8 B je možné výhodně dekódování instrukce implementovat jako LUT tabulku, což by se mělo pozitivně projevit na rychlosti a spolu s alokacemi lokálních proměnných na zásobníku, a typovostí instrukcí zajistit rychlé vykonávání instrukcí. Protože je však množství instrukcí značné, je pro ulehčení práce i zmenšení rizika chyby vystavěn interpret z větší části pomocí maker[5]. Část ARL instrukcí pak zcela.

9.1 Práce s řetězci

Datový typ string je implementován jako struktura Tstring, který de facto string Pascalovského typu, protože během všech operací se stringy známe jejich velikost. To umožňuje také používat funkce rodiny mem*() namísto str*() napříč projektem a opět by mělo vést k efektivnějšímu provádění programů během interpretace. Ve struktuře je také uložen počet referencí na daný řetězec a přiřazení mezi řetězců je řešeno inkrementaci/dekrementací počtu ukazatelů. Pokud je aktuální hodnota 1 a má se dekrementovat, dochází k uvolnění alokované paměti řetězcem. Interpret tedy neobsahuje žádnou centrální správu dynamické paměti a uvolňuje ji hned jak je to možné. Pro literály je tato hodnota 0 a inkrementace/dekrementace se u nich neprovádějí.

```
typedef struct {
    size_t len;
    unsigned refs;
    char str[];
} Tstring;
```

[5] Pravým programátorem interpretu je tedy preprocesor jazyka C ☺

10 Algoritmy IAL

10.1 Boyer-Mooreův algoritmus

Při zkoušení implementace algoritmu z Information Retrieval: Data Structures & Algorithms (součástí Dr. Dobb's Algorithms Collection) se vyskytly nejasnosti a poté implementovaný program nefungoval. Správné výsledky algoritmus neukazoval ani při implementaci námi upravené heuristiky Bad character rule. Algoritmus byl nakonec převzat ze studijní opory IAL a přepsána do jazyka C.

Oproti opory došlo k úpravě hlavní funkci "BMA". Jestliže víme, že poslední místo v tabulce MatchJump (Good suffix rule) bude mít vždy hodnotu skoku jedna, pak můžeme rovnou použít při porovnání posledního písmena vzoru s textem tabulku CharJump (Bad character rule), která v případě neshody použije pravděpodobně větší skok než jedna. Vyhneme se tak například podmínky "if (k < 0)" a volání funkce "max".

10.2 QuickSort

Quicksort je rychlou řadící metodou která pracuje rozdělováním velkých (a na seřazení složitých) polí postupně na menší části. Při rozdělování jsou prvky v poli rozděleny tak, že všechny prvky v pravém poli jsou větší než všechny prvky v levém poli (pro vzestupné seřazení). Jakmile tímto rozdělováním vzniknou jenom jednoprvková pole (jeden prvek je vždy seřazen v jednoprvkovém poli), je celé pole seřazeno (díky způsobu rozdělování).

Optimální by bylo, aby při rozdělování byl určen medián, podle kterého by se pole rozdělilo na 2 stejně velká pole (nebo rozdílná o 1 prvek). To je však výpočetně složité. Proto si zvolíme místo mediánu hodnotu prvku s prostředním indexem (tzv. pivot). Podle této hodnoty se většinou docílí rovnoměrného rozdělení polí.

Průměrná složitost tohoto algoritmu je linearitmická (n log(n)), avšak protože se na rozdělování nepoužívá reálný medián, ale pivot, nejhorší složitost je (n^2).

Existuje varianta nerekurzivní s využitím zásobníku a varianta rekurzivní. Pro implementaci jsme zvolili rekurzivní variantu, protože zápis je jednodušší a přehlednější a není potřeba implementovat speciálně pro tuto metodu zásobník.

10.3 Hashovací tabulka

Pro uchovávání informací o funkcích nebo proměnných je implementována hashovací tabulka (tabulka s rozptýlenými položkami) s explicitním zřetězením, tedy rozptylové pole obsahuje vždy ukazatel na na první položku v lineárním seznamu synonym. Jednotlivé položky obsahují kromě klíče (název proměnné nebo funkce) strukturu, která obsahuje informace o dané proměnné nebo funkci. Explicitní zřetězení bylo zvoleno, protože velikost rozptylového pole je implementačně pevně daná, ale počet identifikátorů může být neomezený. Velikost rozptylového pole byla zvolena tak, aby docházelo k minimálnímu počtu kolizí při obvyklém počtu identifikátorů ve zdrojovém programu, ale přesto nebyla zbytečně velká, protože může existovat několik hashovacích tabulek současně, podle vnoření bloků ve zdrojovém kódu. Rozptylová funkce byla převzata ze zadání programu hashovací tabulky do předmětu IJC (Jazyk C), kde byla využívána pro výpočet klíče anglických slov. Při vytvoření položky hashovací tabulka inicializuje některé složky struktury.

11 Rozšíření

11.1 SIMPLE

Interpret podporuje podmínku if bez části else a u podmíněného příkazu a cyklů lze použít jeden příkaz místo složeného. Implementace podmínky if bez části else vnesla do LL gramatiky konflikt, který je popsán v kapitole o analyzátoru struktury programu. Bylo potřeba v LL gramatice rozdělit podmíněný příkaz if/else na příkaz if s nepovinnou částí else. Podpora samostatného příkazu místo složeného u podmíněného příkazu a cyklů byla vyřešena jednoduše nahrazením na odpovídajících místech v LL gramatice { STMTS } samostatným neterminálem STMT, ze kterého se { STMTS } může vygenerovat.

11.2 WHILE

Interpret podporuje cykly typu while a do-while. Vzhledem k již implementovanému cyklu for, který je z těchto cyklů nejsložitější, nebyla implementace těchto dalších dvou cyklů problém. Oba nové cykly jsou si velmi podobné, liší se pouze umístěním podmínky, která při vyhodnocení false zapříčiní skok mimo cyklus. U obou cyklů se syntakticky vyskytují jednotlivé části v pořadí přesně tak, jak jsou vykonávány (na rozdíl od cyklu for, jehož inkrementační část nacházející se v hlavičce cyklu se vykonává až jako poslední), takže není potřeba generovat zbytečné skoky.

11.3 BASE

Rozšíření *BASE* se týká pouze lexikálního analyzátoru a je řešeno opět konečným automatem. Jako první je nutné rozlišit soustavu (BIN, OCT, HEX) a dále načítat znaky odpovídající dané soustavě. Při escape sekvenci v řetězci je dále důležité kontrolovat počet znaků (BIN – 8, OCT – 3, HEX – 2) – při nedodržení počtu znaků je vrácena lexikální chyba. Načtená hodnota je potom převedena pomocí funkce *strtol* do desítkové soustavy.

11.4 BOOLOP

Interpret podporuje datový typ bool, nové hodnoty true a false, a logické operace. Toto rozšíření se týkalo hlavně rozšíření precedenční tabulky v analýze výrazů o nové operátory. Dále bylo potřeba nový typ a nové operátory a literály přidat do lexikální analýzy. Datový typ bool se ve většině případů chová jako datový typ int (v aritmetických výrazech), na druhou stranu, výsledkem logických výrazů je vždy bool.

12 Rozdělení práce

Jan Velecký: analyzátor výrazů, interpret, generátor instrukcí , Quick-sort řadící algoritmus , testovací soubory

Tomáš Zahradníček: analyzátor programu, generátor instrukcí, tabulka symbolů, interpret, testovací soubory

Adam Piecek: Boyer-Moore vyhledávací algoritmus, testovací soubory, vstup a výstup, vestavěné funkce

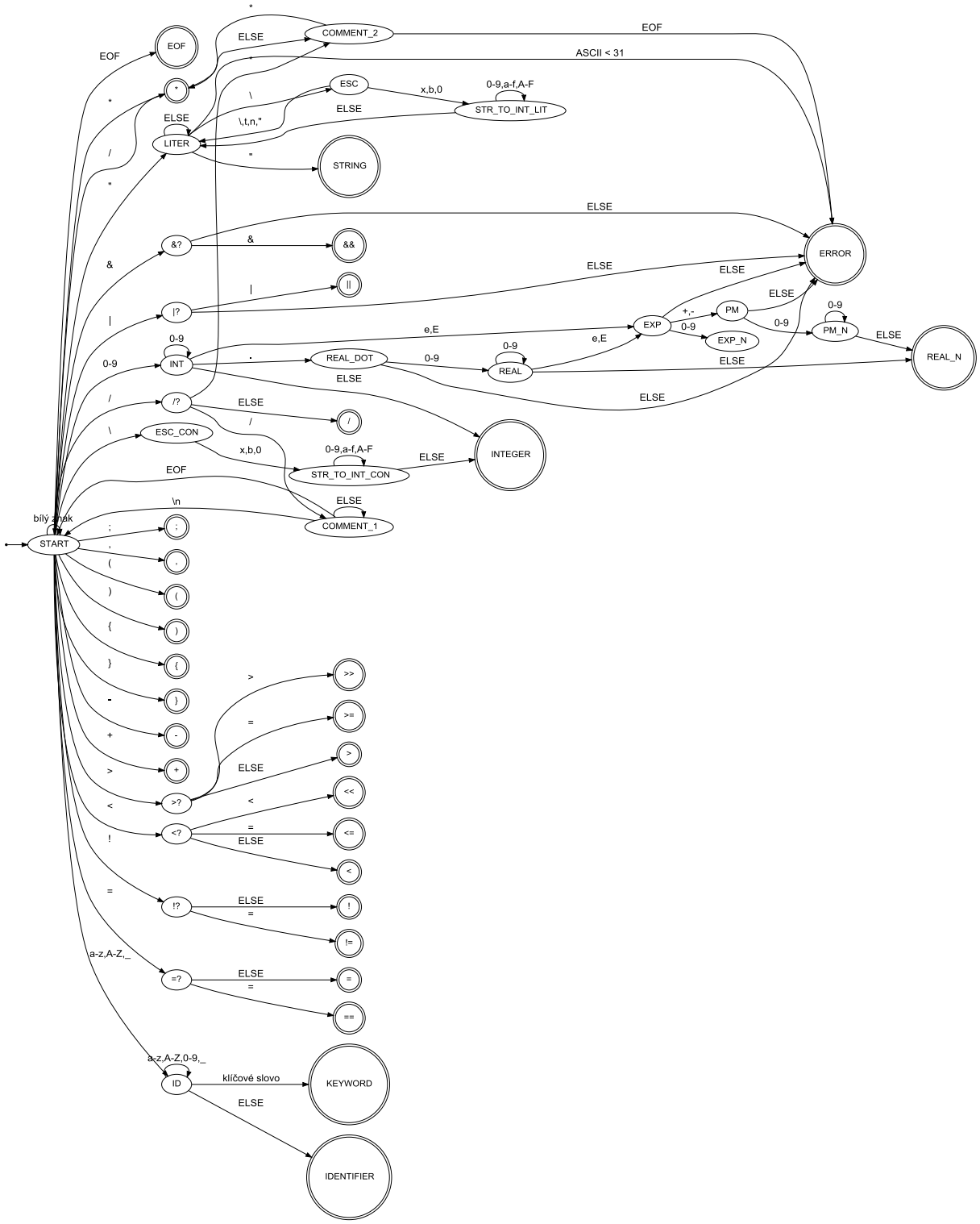
Jan Pavlica: lexikální analyzátor

13 Metriky kódu

Název	Počet souborů	Jména	Přibližná celková velikost [kB]
analyzátor programu	8	listparam, parser, strvars, symboltable	40
interpret	4	interpret, read	29
analyzátor výrazů	6	expression	25
lexikální analyzátor	2	lexer	17
generátor instrukcí	2	instruction	12
algoritmy	2	ial	10
vestavěné funkce	2	builtin_func	5
ostatní	4	symboltable, main, Makefile	15

Celkový počet souborů	32
Celková velikost	152 kB
Celkový počet commitů	212

14 Příloha A: Konečný automat lexikálního analyzátoru



15 Příloha B: LL gramatika

1: PROG → FDEF DEFS
2: DEFS → FDEF DEFS
3: DEFS → ε
4: FDEF→ TYPE id (FPARAMS) FDEFE
5: TYPE→ int
6: TYPE→ double
7: TYPE→ string
8: TYPE→ bool
9: FPARAMS → TYPE id FPARAMSL
10: FPARAMS → ε
11: FPARAMSL → , TYPE id FPARAMSL
12: FPARAMSL → ε
13: FDEFE → ;
14: FDEFE → { STMTS }
15: STMTS → STMT STMTS
16: STMTS →VDEF STMTS
17: STMTS → ε
18: VDEF → TYPEA id INIT
19: TYPEA → TYPE
20: TYPEA → auto
21: INIT → ;
22: INIT → = EXPR ;
23: STMT → { STMTS }
24: STMT → id = ASSIGN
25: STMT → if (EXPR) STMT ELSE
26: STMT → for (VDEF EXPR ; id = EXPR) STMT
27: STMT → while (EXPR) STMT
28: STMT → do STMT while (EXPR) ;
29: STMT → return EXPR ;
30: STMT → cin >> id CINL
31: STMT → cout << TERM COUTL
32: TERM → id
33: TERM → const
34: CINL → >> id
35: CINL → ;
36: COUTL → << TERM
37: COUTL → ;
38: ASSIGN → EXPR ;
39: ASSIGN → id (PARAMS) ;
40: PARAMS → EXPR PARAMSL
41: PARAMS → ε
42: PARAMSL → , EXPR PARAMSL
43: PARAMSL → ε
44: ELSE → else STMT
45: ELSE → ε

16 Příloha C: Tabulka precedenční analýzy

	!	+	-	*	/	<	>	<=	>=	==	!=	&&	||	(	)	i	$
!	<	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
+	<	>	>	<	<	>	>	>	>	>	>	>	>	<	>	<	>
-	<	>	>	<	<	>	>	>	>	>	>	>	>	<	>	<	>
*	<	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	<	>	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<	<	>	>	>	>	>	>	>	>	<	>	<	>
>	<	<	<	<	<	>	>	>	>	>	>	>	>	<	>	<	>
<=	<	<	<	<	<	>	>	>	>	>	>	>	>	<	>	<	>
>=	<	<	<	<	<	>	>	>	>	>	>	>	>	<	>	<	>
==	<	<	<	<	<	>	>	>	>	>	>	>	>	<	>	<	>
!=	<	<	<	<	<	>	>	>	>	>	>	>	>	<	>	<	>
&&	<	<	<	<	<	<	<	<	<	<	<	>	>	<	>	<	>
||	<	<	<	<	<	<	<	<	<	<	<	<	>	<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	E
)	E	>	>	>	>	>	>	>	>	>	>	>	>	E	>	E	>
i	E	>	>	>	>	>	>	>	>	>	>	>	>	E	>	E	>
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<	E	<	E

17 Příloha D: Instrukční sada

Instrukce jsou zapsány ve tvaru *<OPERAČNÍ ZNAK> [<TYP OPERANDU>] [,[<TYP OPERANDU>]]*. Datový typ je vyjádřen na 2 bitech, u číselných instrukcí pouze na 1 bitu.

17.1 Obecné instrukce:

CJMP INDEX, PTR — Conditional JuMP; 2 varianty
- [0] datový typ výsledku

JMP PTR — unconditional JuMP

CALL PTR, INDEX — user function CALL

CALLBIN INDEX, INDEX — Built-IN function CALL

INIT SIZE — INITiate called function

EOP — End Of Program

ERROR VALUE — Program in ERROR state

RETURN , INDEX/VALUE — RETURN from function; 10 variant
- pro čísla: [3:2] návratový datový typ, [1:0] typ navracené hodnoty
- pro řetězce: [2] příznak vracení literálu, [1:0] datový typ string

READ INDEX — READ from stdin; 4 varianty
- [1:0] datový typ čtené hodnoty

WRITE INDEX/VALUE — WRITE to stdout; 6 variant
- [2] příznak vypisování literálu, [1:0] datový typ vypisované hodnoty

LOAD INDEX, INDEX/VALUE — LOAD variable/value into variable; 14 variant
- pro číselné typy: [3:2] datový typ cíle
- pro řetězcové proměnné: [3] příznak inkrementace reference, [2] příznak dekrementace
- pro řetězcové literály: [3:2] datový typ string, [2] příznak dekrementace reference
- [1:0] datový typ zdrojové hodnoty

17.2 Aritmetické/relační/logické instrukce:

ADD INDEX, INDEX/VALUE — perform ADDition between two operands; 6 variant

MUL INDEX, INDEX/VALUE — perform MULtiplication between two operands; 6 variant

SUB INDEX, INDEX/VALUE — perform SUBtraction between two operands; 12 variant

DIV INDEX, INDEX/VALUE — perform DIVision between two operands; 12 variant
- pro dělení/odčítání: [3] příznak prohození operandů (cílový operand zůstává)
- [2] datový typ 1. operandu a cíle, [1] datový typ 2. operandu, [0] příznak konstanty

CMPEQ INDEX, INDEX/VALUE — CoMPare two operands on EQuality; každá 8 variant

CMPNE INDEX, INDEX/VALUE — CoMPare two operands on iNEquality

CMPL INDEX, INDEX/VALUE — CoMPare two operands on Less

CMPG INDEX, INDEX/VALUE — CoMPare two operands on Greater

CMPLE INDEX, INDEX/VALUE — CoMPare two operands on Less or Equal

CMPGE INDEX, INDEX/VALUE — CoMPare two operands on Greater or Equal
- [1] datový typ 1. operandu, [0] datový typ 2. operandu

AND INDEX, INDEX — perform logic AND between two operands; obě 4 varianty

OR INDEX, INDEX — perform logic OR between two operands
- [1] datový typ 1. operandu a cíle, [0] datový typ 2. operandu

NEG INDEX, INDEX — NEGate operand; 2 varianty
- [0] datový typ operandu