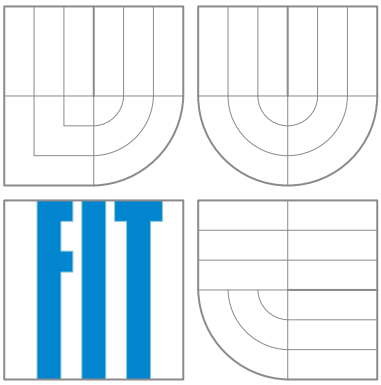


Fakulta informačních technologií
Vysoké učení technické v Brně

Dokumentace projektu

pro předměty IFJ a IAL

Identifikace:	Tým 55, varianta a/4/I		
Vedoucí:	Vendula Poncová	`xponco00`	29%
Řešitelé:	Marek Salát	`xsalat00`	29%
	Tomáš Trkal	`xtrkal00`	29%
	Patrik Hronský	`xhrons00`	13%
Rozšíření:	REPEAT		
	LOCALEXPR		

11. prosince 2011

Obsah

1 Úvod

V této zprávě dokumentujeme náš návrh a implementaci interpretu imperativního jazyka IFJ11 a způsob, jakým jsme postupovali.

V první řadě jsme se zaměřili na návrh datových struktur (kap. 2), neboť na návrhu tabulky symbolů stojí značná část implementace interpretu. Strukturu interpretu jsme rozdělili do čtyř modulů, které společně obstarávají všechny logické fáze interpretu, a ty implementovali (kap. 3). Také jsme implementovali zadané algoritmy pro vyhledávání podřetězce v řetězci a řazení znaků v řetězci (kap. 4).

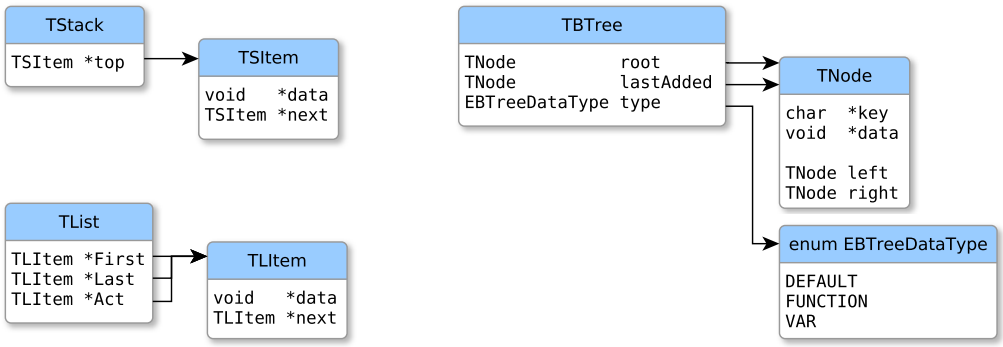
Vycházeli jsme z přednášek a materiálů předmětů IFJ a IAL.

2 Datové typy a struktury

Tato kapitola popisuje návrh datových typů a struktur a jejich implementaci. Jsou zde také definovány některé pojmy používané v dalších kapitolách. Návrhu jsme věnovali hodně pozornosti, neboť je na něm postavena celá implementace interpretu.

2.1 Abstraktní datové typy

Implementovali jsme abstraktní datové typy: seznam, zásobník a binární vyhledávací strom. Abstraktní datové struktury všech těchto typů (obr. 1) obsahují položku ukazatel na data typu void. Díky tomu jsou tyto datové typy zcela obecné a nezávislé na dalších částech programu. V případě použití některého datového typu bylo nutné v daném modulu implementovat funkce pro práci s daty.



Obrázek 1: Zásobník, seznam, binární vyhledávací strom.

Seznam Seznam je lineární, jednosměrný, s ukazatelem na konec. Umožňuje uložit ukazatel na aktivní prvek seznamu a nastavit daný prvek jako aktivní. V rámci optimalizace jsme některé funkce přepsali na makra.

Binární vyhledávací strom Abstraktní datovou strukturu binárního vyhledávacího stromu jsme přizpůsobili tomu, že je kostrou tabulky symbolů. Strom je definován samostatnou strukturou s položkami: kořen stromu, typ stromu a poslední přidaný uzel. Poslední přidaný uzel umožňuje rychlý přístup k poslednímu přidanému prvku.

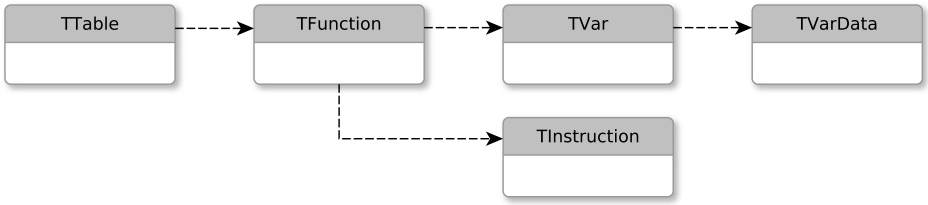
Operace nad binárním stromem jsou implementovány nerekurzivně s výjimkou operace smazání stromu. Nerekurzivní řešení jsme zvolili kvůli optimalizaci interpretu. Funkce pro smazání je implementovaná, ale nepoužívá se, neboť se strom maže v rámci tabulky symbolů.

Původně jsme implementovali AVL strom, neboť je výškově vyvážený a vyhledávání v něm má ve všech případech logaritmickou časovou složitost, zatímco binární vyhledávací strom má nejhorší časovou složitost lineární. Nebyla nám ale taková modifikace dovolena.

2.2 Tabulka symbolů

Tabulka symbolů je sestavena z abstraktních datových typů popsaných v kapitole 2.1 a dat typu TFunction, TVar, TVarData a TInstruction. (obr. 2)

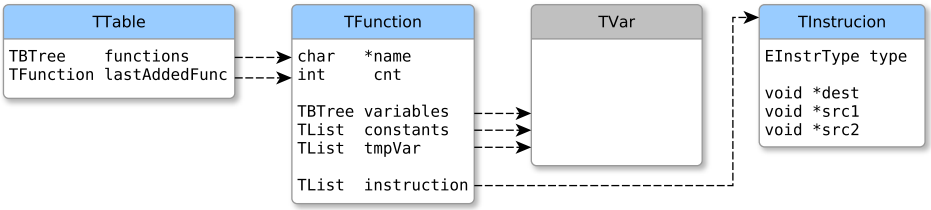
Tabulka symbolů Tabulka symbolů je reprezentovaná datovou strukturou TTable. Položkou v této struktuře je strom funkcí, tj. binární vyhledávací strom s daty TFunction. Položkami struktury TFunction jsou: strom proměnných, seznam konstant, seznam pomocných proměnných a seznam instrukcí. Strom proměnných je binární vyhledávací strom s daty typu TVar. Seznam konstant a seznam pomocných proměnných jsou jednosměrné seznamy s daty typu TVar. Seznam instrukcí je jednosměrný seznam s daty typu TInstruction. Proměnné, pomocné proměnné a konstanty (tedy datové struktury TVar) mají svoji hodnotu a typ uložené v datech, tj. datových strukturách typu TVarData. (obr. 2 a 3)



Obrázek 2: Model tabulky symbolů.

Funkce Funkce jsou jednoznačně určené svým názvem, podle názvu jsou funkce řazeny ve stromu funkcí. Součástí funkce je počítadlo, které určuje, kolikrát je daná funkce aktuálně volaná. Neboť se toto počítadlo používá jako index pole, je počáteční stav počítadla -1. (obr. 3)

Instrukce V rámci interpretu generujeme tříadresný kód. Ten je pro každou funkci reprezentovnán seznamem instrukcí. Instrukce obsahují položky: typ instrukce, adresa výsledku, adresa prvního operandu, adresa druhého operandu. (obr. 3)



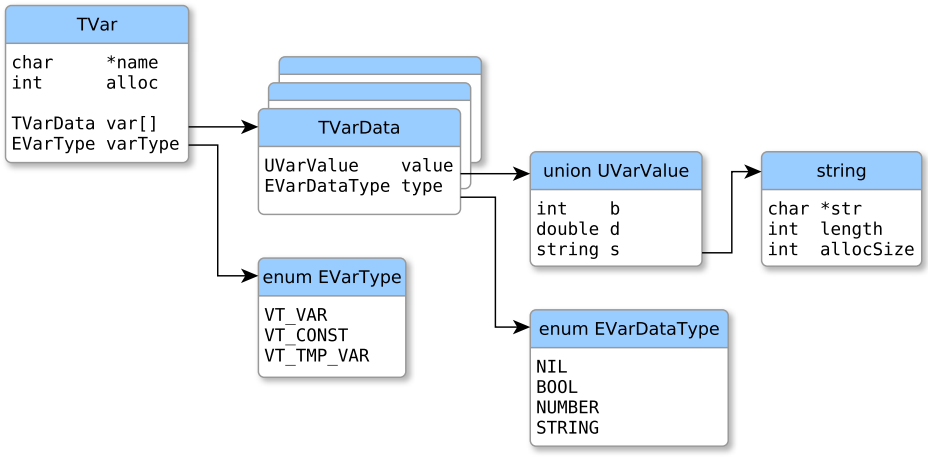
Obrázek 3: Datové struktury pro tabulku symbolů, funkce, instrukce.

Proměnné, konstanty a pomocné proměnné Proměnnou je myšlena uživatelem deklarovaná proměnná, jsou to tedy lokální proměnné a formální parametry funkcí. Proměnné jsou jednoznačně určené svým názvem. Konstanty jsou pomocné proměnné, které nesou hodnotu literálu. Pomocné proměnné používáme pro ukládání mezivýsledků při zpracování výrazů. (obr. 4)

Konstanty a pomocné proměnné nejsou součástí stromu proměnných, aby jej nezatěžovaly při vyhledávání. Seznam pomocných proměnných není součástí seznamu konstant, neboť se se seznamem pomocných proměnných pracuje odlišným způsobem. Vycházeli jsme z předpokladu, že když se při zpracování výrazu použije určitý počet pomocných proměnných, tak po vyhodnocení výrazu nebude jejich hodnota už nikdy potřeba. Proto pomocné proměnné tzv.recyklujeme: když požádáme o novou pomocnou proměnnou, použije se již vygenerovaná, pokud jsme již všechny v daném výrazu použili, vygeneruje se nová. Toto chování jsme velmi snadno implementovali pomocí aktivity seznamu pomocných proměnných.

Data proměnných Z proměnné přistupujeme na její data pomocí pole prvků typu TVarData. Index do pole dat je určen počítadlem volání příslušné funkce. Je tím vyřešen problém rekurzivního volání funkce. Pole jsme zvolili kvůli rychlému přístupu k datům, na druhou stranu při příliš velkém rekurzivním zanoření je nutná realokace. Pomocné proměnné a konstanty mají jen jednu hodnotu, proto v jejich případě přistupujeme vždy na nultý prvek pole hodnot. (obr. 4)

Hodnota proměnné je implementovaná jako unie. Jedním z datových typů, které unie může uchovávat, je string. String je datová struktura, která obsahuje řetězec, délku řetězce a velikost alokovaného řetězce. Nad tímto typem je implementovaná knihovna str, kterou jsme převzali ze vzorového interpretu IFJ a rozšířili.



Obrázek 4: Datové struktury pro proměnné a jejich data.

3 Popis implementace

Činnost našeho překladače obstarávají čtyři moduly: scanner, parser, expression a interpret. Scanner provádí lexikální analýzu. Parser zajišťuje syntaktickou analýzu shora dolů, sémantickou analýzu a generování vnitřního kódu. Expression se stará o syntaktickou a sémantickou analýzu výrazů a generuje matematicko-logické instrukce. Překlad je tedy řízen syntaxí. Interpret interpretuje vnitřní kód a kontroluje sémantiku.

Moduly pracují s datovými strukturami popsanými v kapitole 2.

3.1 Modul scanner

Modul scanner provádí lexikální analýzu zadaného vstupního souboru a dodává tokeny modulu parser.

Lexikální analýza Pokud je modul scanner požádán o další token, přečte ze vstupního souboru jednu lexému a vrátí příslušný token. Atributy tokenu (literály a identifikátory) se ukládájí do proměnné typu string, atribut je pak zpracován modulem parser.

Lexikální analýzu jsme implementovali podle našeho návrhu konečného automatu (příloha A). Konečný automat je reprezentován while cyklem, který čte znaky ze vstupu, a jedním příkazem switch se všemi stavy automatu. Mezi stavy se přechází podle přečteného znaku. Pokud se přečte neočekávaný znak a nejsme v konečném stavu, dojde k lexikální chybě, pokud jsme v konečném stavu, vrátí se znak na vstup a je vrácen příslušný token. Pro rozpoznání tokenu identifikátor je přečtený řetězec porovnán s klíčovými a rezervovanými slovy. Pokud žádnému z nich neodpovídá, jedná se o identifikátor.

Modul scanner obsluhuje čítač řádků zdrojového souboru. Hodnota čítače se vypisuje v případě chyby v době překladu a slouží k usnadnění hledání chyby ve zdrojovém souboru.

3.2 Modul parser

Modul parser žádá o tokeny modul scanner. Na základě tokenů pracuje s tabulkou symbolů a vkládá do ní funkce, proměnné, konstanty a instrukce.

Syntaktická analýza shora dolů Syntaxi jazyka IFJ11 jsme popsali LL-gramatikou a na základě této gramatiky jsme pro potřeby implementace vytvořili LL-tabulku. (příloha B)

Syntaktický analyzátor je implementován podle metody rekurzivního sestupu. Pro každý neterminál je vytvořena funkce, která analyzuje všechna jemu příslušná pravidla. Pokud se v pravidlu objeví terminál *exp*, znamená to, že součástí pravidla je výraz a je zavolán syntaktický analyzátor výrazů z modulu expression.

Sémantická analýza Z hlediska sémantické analýzy parser kontroluje podle zadání deklarace proměnných a funkcí. Vyhledává je v tabulce symbolů.

Generování vnitřního kódu Syntaktický analyzátor generuje tříadresný kód přímým způsobem. Nové instrukce se ukládají na konec seznamu instrukcí příslušné funkce.

3.3 Modul expression

Modul expression je volán modulem parser, pokud je třeba provést analýzu výrazu.

Syntaktická analýza zdola nahoru Syntaktická analýza výrazů probíhá zdola nahoru a využívá k analýze precedenční tabulku a pomocný zásobník tokenů.

Operace nad zásobníkem jsme se snažili co nejvíce zjednodušit. Na zásobník se nevkládají tzv. zarážky. Při vkládání proměnných a konstant na zásobník rovnou uplatňujeme pravidla $E \rightarrow id$ a $E \rightarrow const$, a vkládáme je jako neterminál. Hlídáme pak případ, kdy token, který chceme vložit na zásobník je proměnná nebo konstanta a na vrcholu zásobníku je neterminál, neboť dojde k syntaktické chybě.

Problém určení konce výrazu jsme vyřešili tak, že první chybný token považujeme za konec výrazu a další tokeny nenačítáme. Chybný token je pak dále zpracováván modulem parser. Tento způsob umožnil implementovat modul expression nezávisle na modulu parser a jeho syntaktické analýze.

Sémantická analýza Sémantika výrazů se kontroluje jen u konstant, neboť proměnné nemají v době překladu definované datové typy. Ověřuje se, zda jsou datové typy operandů kompatibilní s příslušnou matematickou nebo logickou operací. Kontrola se provádí pomocí tabulky, kterou pro kontrolu sémantiky matematicko-logických instrukcí používá i interpret.

Generování vnitřního kódu Instrukce se generují jen pro pravidla typu $E \longrightarrow E\ op\ E$. Pro výsledek operace se generuje pomocná proměnná. Ukazatel na výsledek výrazu se předává modulu parser.

3.4 Modul interpret

Interpretace vnitřního kódu probíhá v modulu interpret. K interpretaci dojde jen v případě, že předchozí překlad proběhl bez chyb.

Interpretace Modul interpret se zavolá pro funkci *main*, a pokud jsou v něm volány další funkce, volá rekurzivně sám sebe. Interpretace seznamu instrukcí funguje na principu aktivity seznamu, kdy aktivním prvkem seznamu je aktuálně prováděná instrukce. Na základě typu instrukce se pak provede příslušná akce.

Modul interpret používá zásobník hodnot proměnných, tj. zásobník dat typu TVarData, pomocí kterého se předávají parametry funkcí a návratová hodnota. Parametry jsou na zásobník ukládány v opačném pořadí. Vždy platí, že pokud nad prázdným zásobníkem hodnot provedeme instrukci POP, do dané proměnné se uloží hodnota nil. Pokud byla funkce zavolaná s více parametry než očekávala, zásobník s nepotřebnými hodnotami se vyprázdní.

Sémantická analýza Sémantická kontrola se provádí dle zadání všude, kde ji nešlo provést během překladu.

4 Algoritmy

V této kapitole je popsána naše implementace algoritmů požadovnaných v rámci předmětu IAL.

4.1 Knuth-Moris-Prattův algoritmus

Pro vyhledávání podřetězce v řetězci jsme implementovali Knuth-Moris-Prattův algoritmus (zkráceně KMP). Tento algoritmus je optimalizací triviálního způsobu vyhledávání a jeho výhodou je, že se v prohledávaném řetězci nevrací. Používá konečný automat, který je reprezentovaný vyhledávaným vzorkem a vektorem přechodů. Stavy jsou pak indexy těchto polí. Pro stav i platí, že znak na i-tém indexu vzorku se bude porovnávat s aktuálním znakem prohledávaného řetězce, a pokud bude výsledek porovnávní neuspěšný, přejde se do stavu definovaného na i-tém indexu vektoru přechodů. Pořáteční stav je -1, konečný stav je délka řetězce.

KMP má dvě fáze: neprve se pro zadaný vzorek vytvoří vektor přechodů, a pak se pomocí vzorku a vektoru vyhledává v prohledávaném řetězci. Při generování vektoru přechodů se postupuje od počátečního stavu po konečný. Přechod pro následující stav se určuje s pomocí již vytvořené části automatu. Hledání probíhá tak, že pokud se nacházíme v počátečním stavu nebo je porovnání aktuálního vstupního znaku se znakem ve stavu úspěšné, přesuneme se na následující stav a ze vstupu načteme nový znak. Jinak zůstáváme na aktuálním vstupním znaku a pomocí vektoru přechodů přecházíme na některý z předchozích stavů. Opakujeme, dokud nedojdeme na konec vstupního řetězce. Pokud dosáhneme konečného stavu, je vzorek ve vstupním řetězci nalezen.

4.2 Merge sort

Merge sort je nestabilní, nepřirozený řadící algoritmus. Je to metoda sekvenční, využívá přímý přístup k prvkům pole. Funguje na principu slévání setříděných posloupností. Postupuje zdrojovým polem zleva i zprava, a proti sobě jdoucí neklesající posloupnosti ukládá do cílového pole. Po každém kroku se ověří, zda je posloupnost znaků již seřazena, pokud ne, prohodí se role zdrojového a cílového pole a proces se opakuje. Krokem se rozumí vložení všech prvků zdrojového pole do pole cílového.

V naší implementaci využíváme jedno pole o velikosti dvojnásobku délky vstupního řetězce. Do první části našeho řadícího pole je překopírován vstupní řetězec bez ukončovacího znaku \0, který je pro účely řazení ignorován. Pomocí tohoto pole se třídění uskutečňuje kopírováním znaků ze zdrojové části pole do cílové části. Nakonec je setříděný řetězec překopírován zpět do původního řetězce.

Merge sort má ve všech případech linearitmickou časovou složitost. Jeho nevýhodou je nutnost alokace pole dvojnásobné délky než je vstupní posloupnost znaků.

5 Vývoj a práce v týmu

Tato kapitola popisuje, jakým způsobem náš tým pracoval a kdo měl kterou část projektu na starosti.

5.1 Způsob práce v týmu

Na projektu jsme začali pracovat co nejdříve, neboť jsme chtěli využít možnosti pokusného odevzdání. Nejprve jsme implementovali části, pro které nebyly nutné znalosti z předmětu IFJ. Pak jsme postupně vyvíjeli jednotlivé moduly podle toho, co bylo probíráno na přednáškách. Pro nastudování látky jsme využívali i záznamy přednášek z minulých let. V době pokusného odevzdání byl již interpret zcela funkční, ale kvůli nesplnění formálních požadavků na odevzdané soubory, selhal překlad. Do oficiálního odevzdání jsme ladili, testovali a psali dokumentaci.

Pro vzájemnou komunikaci jsme nejčastěji využívali internetové prostředky (ICQ, diskuze). Osobně jsme se pravidelně scházeli jednou týdně v prostorách školy. Na schůzkách jsme se informovali o současném stavu projektu, řešili aktuální problémy a domluvili se, na čem se bude pracovat další týden.

Zdrojové soubory jsme sdíleli prostřednictvím GIT repozitáře umístěného na stránkách *https://bitbucket.org*. Projekt jsme vyvíjeli na operačních systémech Ubuntu a Windows 7 a pravidelně testovali na školním serveru merlin. Pro testování a ladění jsme hojně využívali program valgrind. Kód jsme optimalizovali pomocí profileru gprof. Dokumentace je napsaná v LaTeXu.

5.2 Rozdělení práce

Procentuální ohodnocení členů týmu je úměrné práci, kterou odvedli, a času, který projektu věnovali.

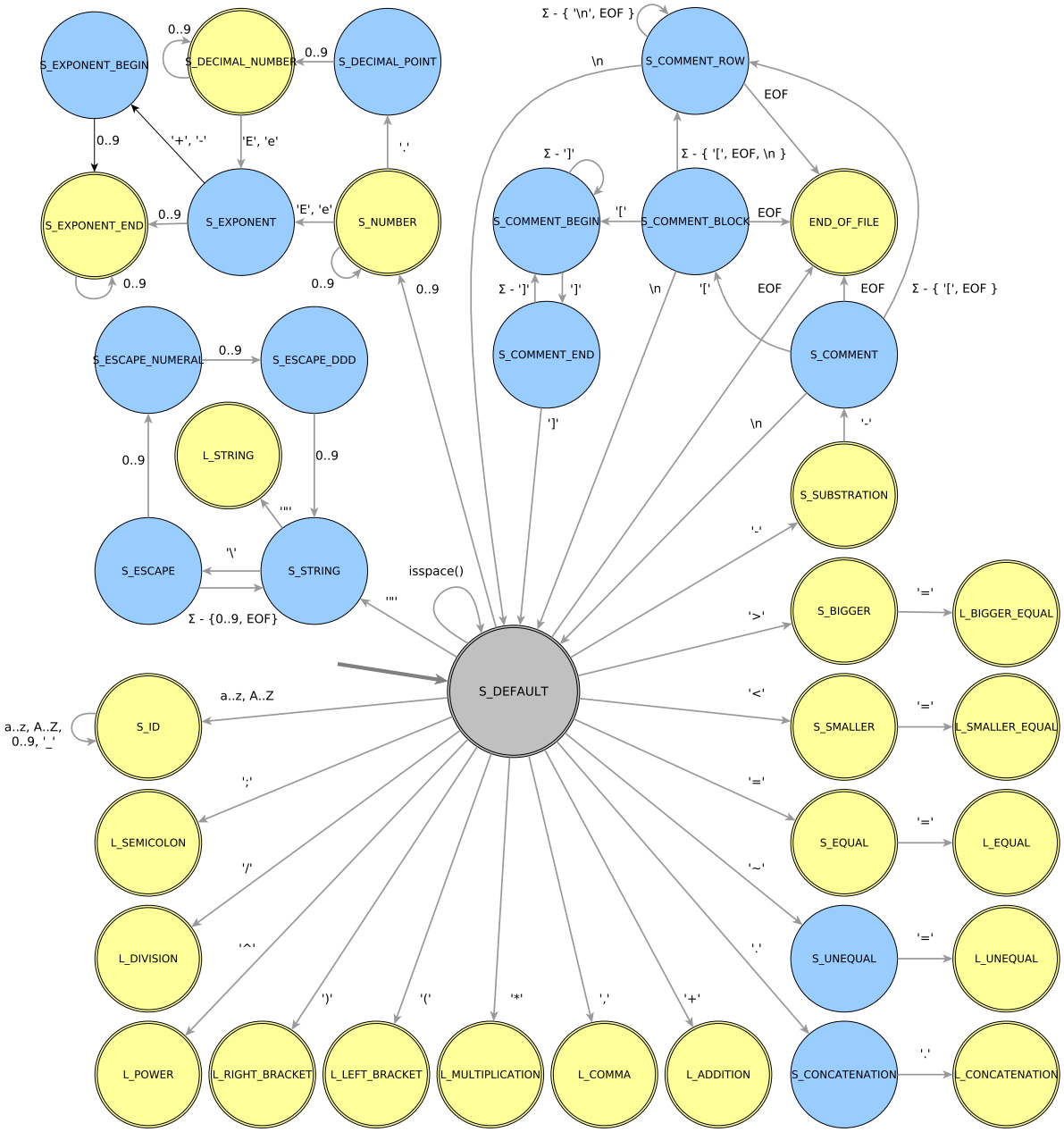
Vendula Poncová	abstraktní datové typy zásobník a seznam Knuth-Moris-Prattův algoritmus moduly expression a library úprava a rozšíření knihovny pro práci s datovým typem string sepsání a úprava dokumentace
Marek Salát	abstraktní datový typ binární strom návrh a implementace tabulky symbolů návrh instrukční sady tříadresných instrukcí modul parser testování
Tomáš Trkal	návrh konečného automatu lexikální analýzy moduly scanner a interpret testování
Patrik Hronský	implementace merge sortu testování

6 Závěr

Implementovali jsme interpret jazyka IFJ11 dle specifikace v zadání a upřesnění na fóru. Při návrhu a implementaci jsme vycházeli z poznatků z předmětů IFJ a IAL a kladli důraz na úsporu času a paměti při běhu programu. Interpret má všechny části funkční, bylo na něm úspěšně otestováno 305 testovacích zdrojových souborů.

Na projektu jsme si vyzkoušeli implementaci některých zajímavých algoritmů, teorii formálních jazyků v praxi a spolupráci v malém týmu.

A Konečný automat



Obrázek 5: Graf konečného automatu pro lexikální analýzu.

B LL-gramatika

G = (N, T, P, S), kde:

N = { <program>, <def_func>, <params>, <params_n>, <stat>, <def_var>, <init>, <lit>, <stat_list>, <command>, <expression_n>, <assign>, <var_params>, <var>, <var_n>}

T = { function, main, (,), *;* , id, end, *,* , local, =, if, then, else, while, do, return, write, read, repeat, until }

S = <program>

P = {

<program>	⟶	function <def_func>
<def_func>	⟶	main ( ) <stat>end ; <EOF>
<def_func>	⟶	id (<params>) <stat>end <program>
<params>	⟶	eps
<params>	⟶	id <params_n>
<params_n>	⟶	eps
<params_n>	⟶	, id <params_n>
<stat>	⟶	<def_var><stat_list>
<def_var>	⟶	eps
<def_var>	⟶	local id <init>; <def_var>
<init>	⟶	eps
<init>	⟶	= exp
<lit>	⟶	literal
<stat_list>	⟶	eps
<stat_list>	⟶	<commad>; <stat_list>
<command>	⟶	if exp then <stat_list>else <stat_list>end
<command>	⟶	while exp do <stat_list>end
<command>	⟶	return exp
<command>	⟶	write (exp <expression_n>)
<expression_n>	⟶	eps
<expression_n>	⟶	, exp <expression_n>
<command>	⟶	id = <assign>
<assign>	⟶	exp
<assign>	⟶	read (<lit>)
<assign>	⟶	id (<params>)
<var_params>	⟶	eps
<var_params>	⟶	<var><var_n>
<var>	⟶	<lit>
<var>	⟶	id
<var_n>	⟶	eps
<var_n>	⟶	, <var><var_n>
<command>	⟶	repeat <stal_list>until exp

}

	function	main	(	)	end	;	id	,	local	=	literal	if	exp	then	else	while	return	write	read	repeat	until
<program>	1																				
<def_func>		2					3														
<stat>					8		8		8			8				8	8	8		8	
<params>				4			5														
<params_n>				6				7													
<def_var>					9		9		10			9				9	9	9		9	
<stat_list>					14		15					15			14	15	15	15		15	14
<init>						11				12											
<lit>											13										
<command>							22					16				17	18	19		32	
<expression_n>				20				21													
<assign>							25						23						24		
<var_params>				26			27				27										
<var>							19				28										
<var_n>				30				31													

Tabulka 1: Tabulka LL gramatiky.

C Metriky kódu

Počet zdrojových souborů: 23 souborů

Počet řádků zdrojového textu: 5440 řádků

Procentuální podíl komentářů: 33%

Velikost statických dat: 160 789 B

Velikost spustitelného souboru: 127 032 B (školní server merlin, bez ladících informací)