Implementace překladače imperativního jazyka IFJ18.

Dokumentace k projektu IFJ/IAL

Tým 85 varianta 1

Kladňák Martin 25%

Urbánek Petr 25%

Valecký David 25%

Václavík Marek 25% - Vedoucí

Úvod

Tato dokumentace popisuje návrh a implementaci interpretu imperativního jazyka IFJ18 ,který je podmnožinou jazyka Ruby. Každá kapitola dokumentace popisuje způsob implementace dané problematiky. Dokumentace také zahrnuje způsob řešení algoritmů z pohledu předmětu IAL.

Práce v týmu

Jednou z nejtěžších věcí na projektu je práce v týmu. Náš tým se začal scházet od 23.10.2018 pravidelně každé úterý, kde se řešilo kdo, co a do kdy udělá. Dále jsme komunikovali přes sociální sítě a to hlavně pomocí aplikace messenger, kde jsme si posílali “screenshoty” co komu nejde a díky tomu mnohem rychleji odhalili problémy . Pro sdílení a jednoduchou kontrolu verzí jsme využili technologii Git na stránce Gitlab. Ze začátku jsme měli problémy s kolizemi ale postupem času jsme se naučili říkat si kdo na jakém souboru právě pracuje.

Lexikální analyzátor

Lexikální analýza je implementována pomocí konečného automatu. Lexikální analýzu zavoláme funkcí get_token. Ve funkci načítáme znak po znaku a podle aktuální stavu řetezce a znaku na vstupu se rozhode, jak budeme pokračovat dál. Rozhodnutí se děje podle switche který představuje konečný automat. Funkce vrací v parametru strukturu token se kterou potom pracuje syntaktický analyzátor.

Syntaktický analyzátor

Syntaktickou analýzu jsme implementovali rekurzivním sestupem pomocí LL gramatiky v jednom průchodu. Na syntaktické analýze se stal největším problémem objem kódu a rekurze. Tyto dva faktory způsobily, že debugování jakékoli chyby trvalo velice dlouho. Celá implementace zahrnovala spoustu pamatování a přemýšlení nad tím, kde jsem a co se vlastně teď může stát. I v posledních hodinách před odevzdáním se nám vloudila “chybička” do kódu. Celkově bychom syntaxi hodnotili jako nejtěžší část celého projektu.

Precedenční tabulka

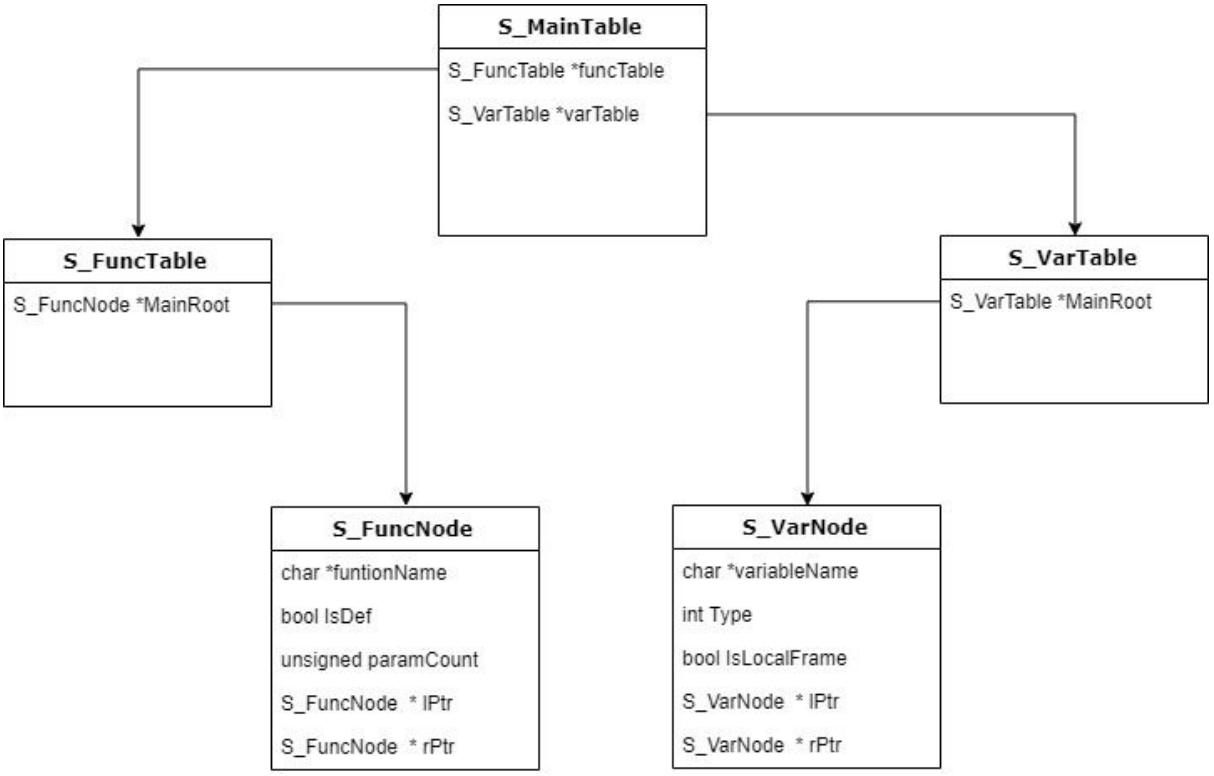
Precedenční tabulka slouží k vyhodnocování výrazů. Pokud pomocí rekurzivního sestupu nalezneme výraz, budeme volat funkci expression která nám pomocí precedenční tabulky řekne jak se daný výraz má vyhodnocovat. Při tvorbě pravého rozbor ho ihned vyhodnocuje sémantická analýza. Pokud se zjistí že daný výraz není možné rozložit podle pravého rozboru jedná se o chybu. Precedenční tabulku jsme si implementovali jako dvourozměrné pole kde hodnoty uvnitř znázorńují co se má z buńkou dělat.

LL - gramatika

Podle zadání vytvořit gramatiku bylo ze začátku dost problémové. Takže na tvorbě LL gramatiky jsme si dali vskutku hodně záležet. Zabrala spoustu času.
Ještě než se začalo programovat, chtěli jsme ji mít bez chyby, abychom pak nemuseli měnit kód, po případném zjištění nedeterminismu při implementaci. LL gramatiku jsme použili na zachycení syntaktických chyb. Enum nám slouží k odhalení o jakou chybu se jedná a vracíme její exit code pomocí error funkce, která vypisuje na stderr

Tabulka Symbolů

Rozhodli jsme se tabulku symbolů rozdělit do dvou tabulek, do tabulky funkcí a tabulku proměnných. Obě tabulky jsou ze zadání implementovány jako binární stromy, které nad sebou mají “abstraktní kořen” S_MainTable. Každý strom je řazen podle svého jména, to slouží jako klíč. Ten jsme vytvořili pro jednodušší předávání do funkcí. Pro vkládání do tabulek jsme využili nerekurzivní implementace, u vyhledávání naopak rekurzivní.



Sémantický analyzátor

Sémantiku jsme se rozhodli kontrolovat v rámci modulu parser. Zde se zkontrolují datové typy a v případě že se nevyskytne žádná chyba se vygeneruje nová instrukce a vloží se do seznamu instrukcí.

Generování kódu

Pro generování kódu IFJcode18 jsme se rozhodli využít tříadresného kódu, který ukládáme do jednovazného seznamu. Pokud nastala jakákoliv chyba, assembler se vůbec nedostal na povrch. Až teprve po úspěšné Syntaktické a Sémantické analýze jsme vypsali kód na stdout. Seznam byl koncipován na tři části, které se vytvořili pomocí funkcí BaseGen, BuildInGen a MainGen. Funkce BaseGen vytvořila základní rámec pro IFJcode18 a návěští pro chybu “dělení nulou”. BuildInGen vytvořila vestavěné funkce a MainGen udělala návěští kde začne samotný program a zbytek uživatelských instrukcí.t U všech funkcí jsme se snažili, aby se parametry předávali pomocí zásobníku a pomocných proměnných a konečný výsledek zanechali na vrcholu zásobníku.

Rozdělení práce

Martin Kladnak

- implementace lexikálního analyzátoru
- návrh konečného automatu pro lexikální analyzátor
- testování
- dokumentace

Marek Václavík

- implementace sémantické analýzy
- převod do tříadresného kódu

David Valecký

- návrh a implementace syntaktické analýzy, rekurzivní sestup i precedenční SA
- návrh LL gramatiky, precedenční tabulky
- testování

Petr Urbánek

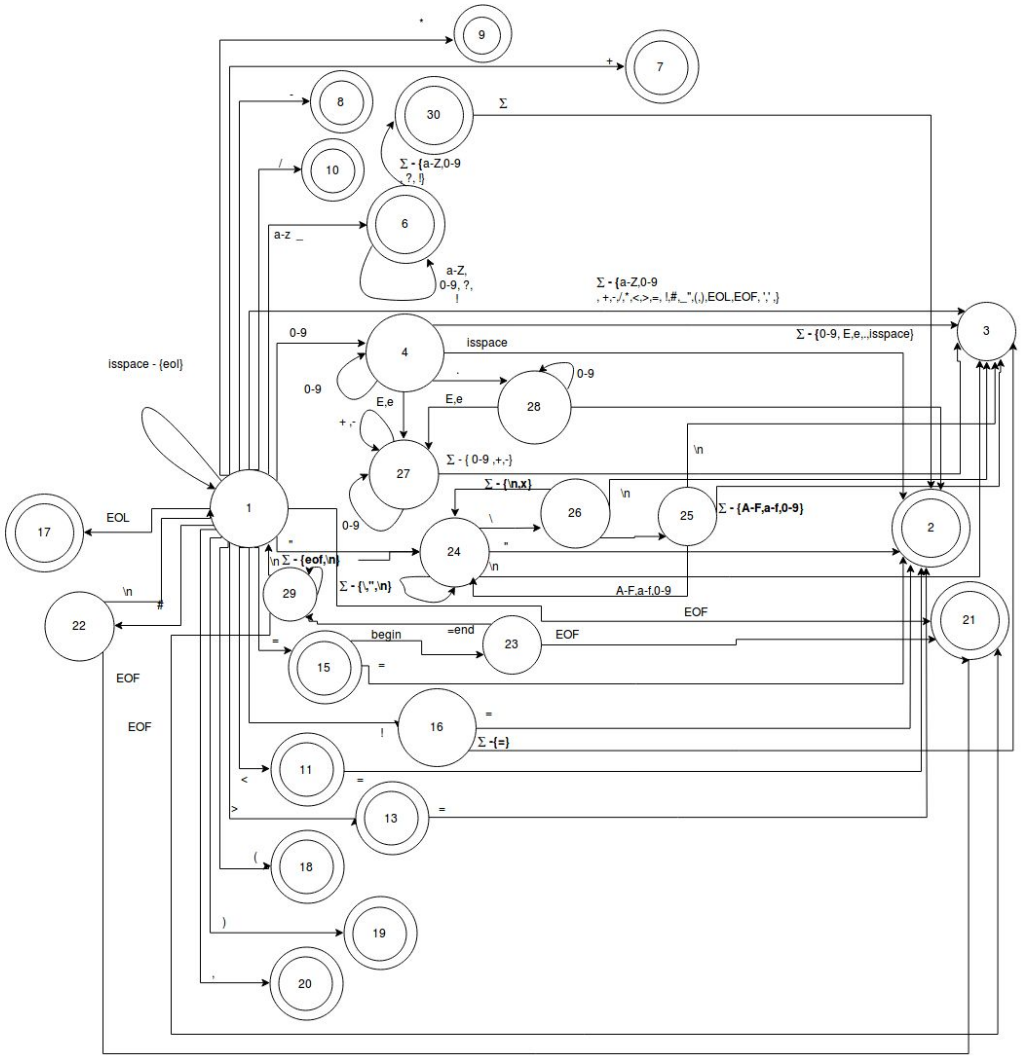
- návrh a implementace tříadresného kódu
- návrh a implementace tabulky symbolů
- Základní synchronizace (Makefile)
- dokumentace

Shrnutí

Mezi největší problémy na projektu patřila organizace času, ze začátku jsme ani nepřemýšleli nad tím, že by se projekt nedal stihnout, bohužel toto smýšlení nás obralo o drahocenný čas. Na konci Listopadu jsme zjistili, že času už moc nezbývá, a ze svěžích studentů se začali vstávat chodící zombie. Každý den jsme dělali do rána, abychom zprovoznili alespoň základní funkčnost, za kterou jsme brali alespoň základní přiřazení a počítání výrazů. Dnes v den odevzdání jsme zprovoznili základní generování kódu a s pomyslnou nadějí doufáme, že se příští rok uvidíme s přednášejícími IFJ jen na chodbách.

Přílohy

Konečný automat lexikální analýzy



S_START	1
S_END,	2
S_CHYBA,	3
S_INTEGER,	4
S_IDENTIFIKATOR,	6
S_PLUS,	7
S_MINUS,	8
S_KRAT,	9
S_DELENO,	10
S_MENSI,	11
S_VETSI,	13
S_ROVNO,	15
EOL	17
S_LEVA_ZAVORKA,	18
S_PRAVA_ZAVORKA,	19
S_CARKA,	20
S_END_OF_FILE,	21
S_COMENT,	22
S_BLOK_COMENT,	23
S_STRING,	24
S_X_STRING,	25
S_LOM_STRING,	26
S_EXPONENT,	27
S_DESETINY,	28
S_CEKEJ,	29
S_KEY_WORDS	30

LL-Gramatika

<prog>	->	<fun_def> <prog>
<prog>	->	<statement_list> <prog>
<prog>	->	EOL <prog>
<fun_def>	->	DEF ID (<param>) EOL <statement_list> END <prog>
<param>	->	ID <parameters>
<param>	->	eps
<parameters>	->	, ID <parameters>
<parameters>	->	eps
<statement>	->	WHILE <expression> DO EOL <statement_list> END
<statement>	->	IF <expression> THEN EOL <statement_list> ELSE EOL <statement_list> END
<statement>	->	ID <next>
<statement>	->	<expression> EOL
<statement_list>	->	<statement> <statement_list>
<next>	->	= <assign_value> EOL
<next>	->	<call_func>
<assign_value>	->	<expression>
<assign_value>	->	ID <assign_value2>
<assign_value2>	->	(<term>)
<assign_value2>	->	<term>
<call_func>	->	(<term>)
<call_func>	->	<term>
<term>	->	ID <terms>
<term>	->	eps
<terms>	->	, ID <terms>
<terms>	->	eps

LL-Tabulka

	DEF	WHILE	IF	EOL	ID	(	)	=	ELSE	END	,	$	expresion
<prog>	1	2	2	3	2								
<fun_def>	4												
<param>					5		6						
<parameters>						8					7		
<statement>		9	10		11								12
<statement_list>		13	13		13								
<next>				15	15	15		14					
<assign_value>					17								16
<assign_value2>				19		18	19						
<call_func>				20		20	21						
<term>				23	22		23						
<terms>				25			25				24		

Precedenční tabulka

*	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	<	>	<	>
+	<	<	>	>	>	>	<	>	<	>
-	<	<	>	>	>	>	<	>	<	>
R	<	<	<	<		>	<	>	<	>
R2	<	<	<	<	<					
(	<	<	<	<	<		<	=	<	
)	>	>	>	>	>			>		>
ID	>	>	>	>	>	>		>		>
$	<	<	<	<	<	<	<		<	

R: <,>,>=,<=
R2: ==,!=

1: E -> E + E
2: E -> E - E
3: E -> E * E
4: E -> E / E
5: E -> E > E
6: E -> E < E
7: E -> E >= E
8: E -> E <= E
9: E -> E == E
10: E -> E != E
11: E -> (E)
12: E -> ID