Dokumentace týmového projektu IFJ2019

Team 116, varianta II

08. 12. 2019

Ondrej Kondek	xkonde04
Marek Fiala	xfiala60
Daniel Kamenický	xkamen21
Tomáš Lisický	xlisic01

Obsah

1. Úvod

Naším úkolem bylo vypracovat společný projekt do dvou předmětů, a to IAL a IFJ. Zadáním bylo vypracovat/naprogramovat překladač v programovacím jazyce C. Tento překladač pracoval se vstupním zdrojovým kódem psaným v jazyce IFJ19. IFJ19 je zjednodušená podmnožina programovacího jazyka Python 3, což je dynamicky tipovaný imperativní (objektově-orientovaný) jazyk s funkcionálními prvky.

2. Lexikální analýza

Prvým krokom pri implementácii prekladača bola lexikálna analýza. Úloha lexikálnej analýzy je defragmentovať časti zdrojového súboru na tzv. tokeny. Rôzne tokeny nadobúdajú rôzne atributy, ktoré sú ďalej využívané pri implementácii. Token môže nadobúdať práve jeden z nasledujúcich typov: identifikátor, kľúčové slovo, reťazec, viacriadkový reťazec, celé číslo, desatinné číslo, indent (odsadenie), dedent, EOL, EOF, ale aj aritmetický operátor, zátvorka, dvojbodka, porovnávacie operátory alebo iný znak, ktorý je v jazyku IFJ19 podporovaný. Okrem typu, každý token obsahuje údaj o odsadení a reťazcovú časť, ktorá je naplnená len v prípade, že daný token má typ: identifikátor, kľúčové slovo, celé alebo desatinné číslo, jednoriadkový alebo viacriadkový reťazec. Reťazcová časť tokenu je tvorená štruktúrou dynamic string, ktorá umožňuje postupné zväčšovanie poľa, v ktorom je reťazec uložený, a tak nie sme obmedzení napr. veľkosťou identifikátora.
Získavanie tokenov umožňuje funkcia get_token.

Pred implementáciou lexikálnej analýzy sme vytvorili detereministický konečný automat (viď nižšie), na základe ktorého sme lexikálnu analýzu implementovali.
Automat je v implementácii reprezentovaný ako nekonečný while cyklus, v ktorom vchádzame do stavov podľa vstupu a ak je tento stav konečný, vystupujeme z cyklu while a posielame token. Ak do jedného zo stavov vchádza nepovolený znak, tak funckia vracia 1 – ktorá signalizuje lexikálnu chybu.

Pre generovanie a spoznanie indentov a dedentov sme použili odporúčaný postup – pomocný zásobník, ktorý si pamätá aktuálne odsadenie a v prípade zmeny buď pridá nové odsadenie na zásobník – indent alebo odoberie – dedent.

3. Sémantická analýza

Při sémantické analýze využíváme strukturu pData, která slouží pro hlavní „komunikaci“ mezi moduly a funkcemi. Ukládáme si do ní všechny potřebná data a pomocné proměnné používané například při vyhodnocování určitého výrazu, či kontroly datovoho typu.
K ukládání proměnných a jmen funkcí používáme tabulky s náhodně rozptýlenými položkami. Pro ukládání lokálních proměnných *symtable local*, pro funkce a globální proměnné *symtable global* . S názvem proměnné se ukládá i její datový typ *type_item type* nebo jestli již byla definována *bool def*.

4. Syntaktická analýza

Hlavní povinností syntaktické analýzy je kontrolovat správnou posloupnost tokenů. Celá syntaktická analýza se řídí pomocí LL(k)-gramatiky a metodou rekurzivního sestupu určuje pravidla vygenerované pomocí LL(1) – tabulky. LL(1)-gramatika nám tvořila základní kostru celého modulu parser.c, tak, že každé pravidlo začínající stejným neterminálem získané z LL(1)-tabulky má vlastní funkci.
Data potřebná k dalšímu postupu jsou předané ve struktuře pData, která je předána jako parametr v každé funkci. Pomocí lexikální analýzy dostává syntaktická analýza tokeny přes makro TOKEN(), které bere ze standardního vstupu. Tyto tokeny postupně projdou sítem pravidel LL(1) gramatiky, a zjistí se jestli daná posloupnost tokenů odpovídá zadaným pravidlům.
Při špatné analýze celý program končí a nedostává se ani do fáze generování kódu.
Problémy, které se naskytly během studia zadání jsme řešili přidáváním proměnných do zmiňované struktury pData, nebo přidáním dalších pravidel jako je například funkce 'body_after_indent', která zajišťuje příchod definice funkce kdekoliv v těle kódu (definice může přijít až po volání dané funkce). Další velký problém nastal při příkazu 'return', který muže přijít pouze v těle funkce. Ten jsme vyřešili přidáním proměnné *bool in_func*, která nám během celé analýzy říká, zda se vyskytujeme či nevyskytujme v těle funkce.

Výrazy se zpracovávají pomocí precedenční syntaktické analýzy. Tato parsovací metoda postupuje zdola nahoru a je založena na precedenční tabulce. Jak již název napovídá, tabulka určuje precedenci (prioritu) operátorů. Tabulka převádí infixovou notaci na postfixovou, za použití zásobníků na tokeny. Aby tabulka nebyla příliš obsáhlá a veliká, zredukovali jsme operátory se stejnou prioritou do jednoho sloupce/řádku. Výraz {+, -} obsahuje operátory plus a mínus, výraz {*, /, //} operátory násobení, dělení a celočíselný zbytek po dělení. Výraz {Rel} zastupuje všechny relační operátory tedy větší/menší (< ,>), větší nebo rovno/menší nebo rovno (<=, >=), porovnání dvou operátorů (==) a negaci porovnání (!=). Výraz {i} zastupuje string, a hodnoty typu double a integer. Výraz {$} zastupuje dno zásobníku případně čím může výraz v definici funkce končit což je { , } nebo {) }. Dále tabulka levou či pravou závorku. Pokud výraz obsahuje nějaký znak, který do výrazu nepaří, tabulka automaticky vrací chybu.

Pokud tabulka vyhodnotí symbol jako { < }, vložíme zarážku (token STOP) za nejvrchnější terminál a načítám další token. Symbol { > } zredukuje nejvrchnější terminály podle určitého pravidla (redukce na E_TOK) a odstraní nejvrchnější zarážku ze zásobníku. Pokud neexistuje pravidlo, podle kterého by šli pravidla na zásobníku zredukovat, opět hlásíme chybu. Symbol { = } odstraní nejvrchnější terminál ze zásobníku a načteme další znak. Symbol { _ } hlásí úspěšné načtení a zredukování výrazu, zatímco symbol { - } označuje chybu, potkají-li se znaky které ve výrazu po sobě být nesmí a hlásíme chybu ve výrazu.

5. Generace vnitřního kódu

Generovanie kódu sa skladá z viacerých funkcií - ktoré sú implementované v súbore codegen.c a deklarované v codegen.h - povolávaných zo syntakticko-sémantického analyzátoru.
Všetky funkcie využívajú tzv. dynamický reťazec, do ktorého zapisujú svoje hodnoty a ten sa po úspešnom preklade vytlačí na štandartný výstup.
Pre definície funkcií a tela main sme použili dva rozličné dynamické reťazce (out_ a main_).
Začiatok generovania kódu spočíva vo vygenerovaní hlavičky medzikódu ".IFJcode19", ktorá je pre správny beh medzikódu rozhodujúca.
Pokračuje generovanie globálnych premenných, ktoré sú pre vyhodnotenie výrazov - aritmetických, relačných či logických - nepostrádateľné.
Následuje generácia skoku na hlavné telo kódu &main, vytlačenie obsahu "out_" v ktorom sú definované všetky vstavané a uživateľské funkcie a až potom sa vytlačí main_ - telo &main.

Generovanie návestí mimo vstavaných funkcií spočíva v zložení názvu inštrukcie/funkcie, globálnej premennej label_id a prípadne dodatkov podobným prepínačom (napr. -exit).
Každý label sa povinne začína špeciálnym znakom &.
Generovanie premenných sme rozdelili na pred-definované a uživateľom definované, aby nenastala kolízia v názvoch a to priradením špeciálneho znaku v názve (* vstavané, _ uživateľské).

Generovanie funkcií začína návestím danej funkcie, posuvom rámca a vytvorením nového rámca.
Nasleduje preklad funkcie z IFJ19 do IFJcode19 a ukončuje sa návratom rámca a návratovou inštrukciou - prípadná návratová hodnota je posunutá na vrchol dátového zásobníka.
Predávanie hodnôt prebieha cez dátový zásobník zľava-do-prava pre výrazy (napr. a+b) a inverzne pre funkcie a vstavané funkcie - parametre cez rámce nepredávame.

Výrazy fungujú na princípe presunu parametrov do pomocných globálnych premenných definovaných v hlavičke kódu.
Po presune sa skontroluje správnosť dátových typov, implicitná konverzia operandov (jedine typ integer na float) pre vybrané inštrukcie.
Výsledky inštrukcií sa ukladajú do globálnej premennej, ktorej hodnota sa prideľuje podľa výrazu.
Kontrola dátových typov prebieha aj pri zásobníkových verziách inštrukcií, pričom sa vezmú z vrcholu zásobníku, skontrolujú a vrátia naspäť v opačnom poradí, aby sa dodržalo poradie.

6. Speciální algoritmy a datové struktury

6.1 Tabulka s rozptýlenými položkami:

Pri implementácii projektu sme použili tabuľky s rozptýlenými položkami – podľa zadania. Takéto tabuľky boli využívané ako úložisko premenných a funkcií, ktoré boli v programe definované. Ak program deklaroval premennú, bola následne pridaná do tabuľky (podobne funkcia). Vďaka tomuto sme vedeli spätne overovať, či je premenná globálna alebo lokálna a či už bola definovaná, a tak sme dokázali hlásiť chyby typu SEM_UNDEF (číslo 3).
Veľkosť tabuľky bola vybraná so zámerom dosiahnuť, čo najväčšiu efektivitu, a tak sme zvolili prvočíslo 12289. Hashovacia funkcia bola vytvorená s rovnakým zámerom a inšpirovaná predchádzajúcim projektom z predmetu IJC.
Každá položka v tabuľke je štruktúrou obsahujúcou identifikátor, dáta a ukazateľ na ďalší prvok v tabuľke. Dáta sú taktiež štruktúrou, v ktorej je uložený typ itemu alebo funkcie, informácia, či bola daná premenná alebo funkcia definovaná, identifikátor a informácia o tom či je daná položka funkciou – poprípade koľko očakáva parametrov.

6.2 Dynamický řetězec:

Pri ukladaní rôznych vstupných dát – napr. identifikátorov, čísla, stringu, sme vopred nevedeli ako dlhý bude ,a tak sme nedokázali odhadnúť do akého veľkého poľa ho uložiť. Tento problém sme riešili pomocou tzv. dynamického reťazca, ktorý nám umožnil do „nekonečna“ pridávať nové a nové znaky do stringu a to vďaka realokácii, ktorá v ňom automaticky prebiehala na základe dĺžky pridávaných znakov. Dynamický reťazec si ukladá aktuálnu veľkosť, na ktorú je alokovaný a ak by mal presiahnuť dostupný priestor, tak sa automaticky reallocuje na o 8b väčší string. Táto štruktúra je implementovaná v module dynamicstring.c a dynamicstring.h

6.3 Zásobník na tokeny:

Pri precedenčnej syntaktickej analýze sme využili dátovú štruktúru zásobník, ktorý bol implementovaný tak, aby sme dokázali na neho vkladať jednotlivé tokeny. Na to aby sme mohli s týmto zásobníkom pracovať bolo nutné vytvoriť niekoľko funkcií: Push, Top, Pop, Full, Empty, Init. Ďalej pomocné funkcie copy_token a insert_stop. Funkcia copy_token sa stará o korektné prekopírovanie dát z tokenu na token a funkcia insert_stop sa využíva

Zásobník funguje ako pole ukazateľov – každý ukazateľ v poli ukazuje na daný token a ak je nový token pridaný – vrchol zásobníka posúvame.

7. Práce v týmu

Ondrej Kondek – 32% (Lexikální analýza, Generování kódu, testování, dokumentace)

Daniel Kamenický – 25% (Syntaktická analýza, Semantická analýza, testování, dokumentace, prezentace)

Marek Fiala – 25% (Syntaktická analýza, Semantická analýza, testování, dokumentace, prezentace)

Tomáš Lisický – 18% (Generování kódu, dokumentace)

Naša práca v tíme spočívala zdieľaním zdrojových kódov medzi sebou cez repozitár GitHub, pričom sme medzi sebou komunikovali buď cez internet (tj. Discord, Facebook), alebo osobne v priestoroch fakulty.
Práca s Gitom nám umožnila pracovať na viacerých súboroch naraz, taktiež umožnila aj prácu viacerých ľudí na jednom súbore.

Spočiatku sme stanovili spravili náčrt projektu, resp. smer, ktorým bude napredovať a rozdelili sme si programové časti - komu nalieha určitá časť kódu.
Po implementovaní hrubej kostry celého programu, sme pomaly nachádzali prekážky, ktoré bolo treba odstrániť pred samotným spojazdnením programu.
V mnohých prípadoch sme museli testovať naslepo či simulovať očakávané vstupy.
V konečnej fázy, kedy sa nám konečne podarilo program spojazdniť, sme dolaďovali nedostatky a chybné vstupy.
Bohužiaľ na naše prekvapenie sme stále nachádzali chyby a postupne sme boli nútení ich odstraňovať, čo viedlo k časovej tiesni.

8. LL - gramatika

1. <body> -> def function_id (<def_params>) : EOL INDENT <body_after_indent> DEDENT <body>
2. <body> -> id = <after_id> EOL <body>
3. <body> -> expression <body>
4. <body> -> if condition : EOL INDENT <body_after_indent> DEDENT else : EOL INDENT <body_after_indent> DEDENT <body>
5. <body> -> while condition : EOL INDENT <body_after_indent> DEDENT <body>
6. <body> -> pass EOL <body>
7. <body> -> EOL <body>
8. <body> -> EOF

9. <body_after_indent> -> id = <after_id> EOL <body_after_indent>
10. <body_after_indent> -> expression <body_after_indent>
11. <body_after_indent> -> EOL <body_after_indent>
12. <body_after_indent> -> if condition : EOL INDENT <body_after_indent> DEDENT else : EOL INDENT <body_after_indent> DEDENT <body_after_indent>
13. <body_after_indent> -> while condition : EOL INDENT <body_after_indent> DEDENT <body_after_indent>
14. <body_after_indent> -> pass EOL <body_after_indent>
15. <body_after_indent> -> return <return_value> <body_after_indent>
16. <body_after_indent> -> ε

17. <after_id> -> ε
18. <after_id> -> expression
19. <after_id> -> id (<params>)

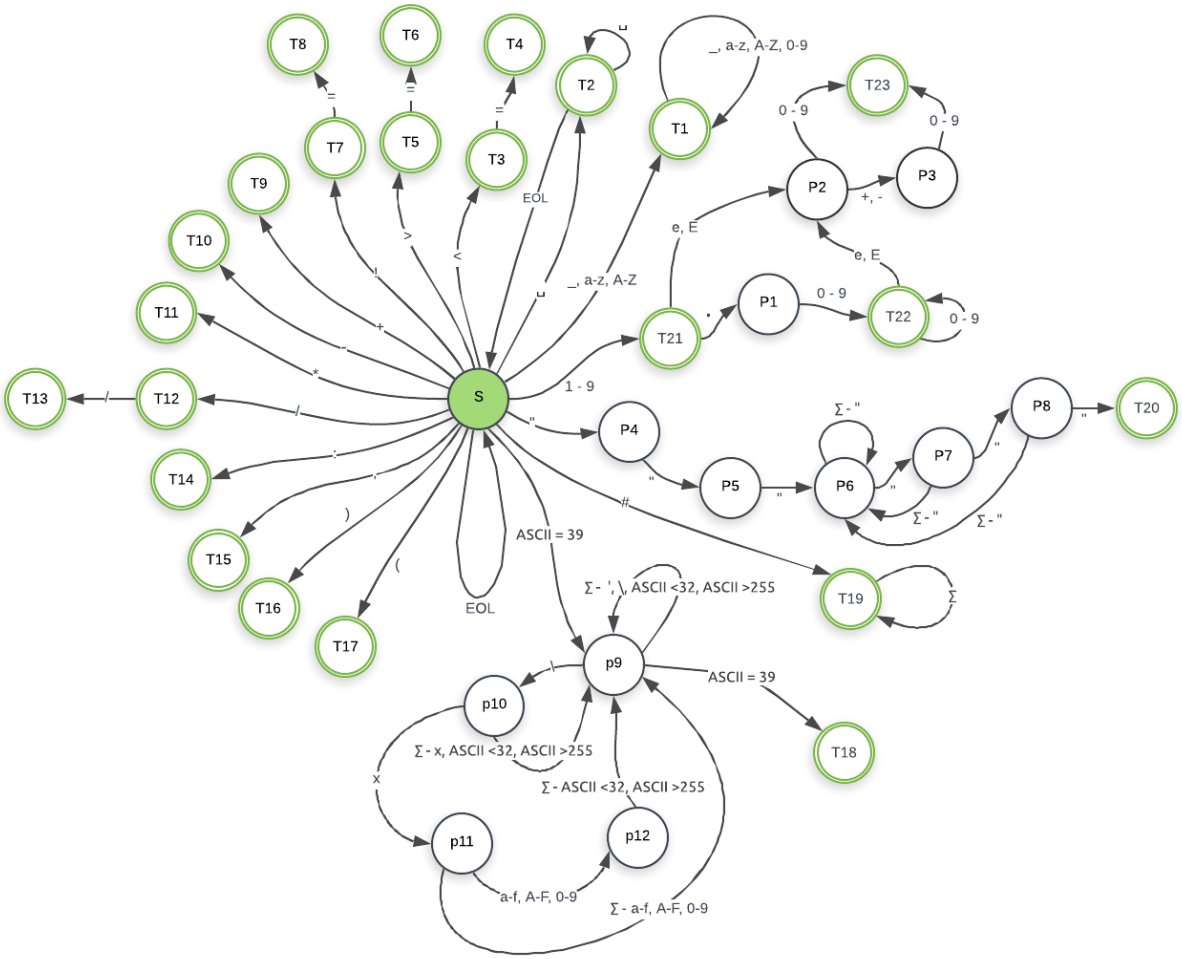
20. <params> -> ε
21. <params> -> id <params_n>
22. <params> -> expression <params_n>

23. <params_n> -> ε
24. <params_n> -> , id <params_n>

25. <def_params> -> ε
26. <def_params> -> id <def_params_n>

27. <def_params_n> -> ε
28. <def_params_n> -> , id <def_params_n>

29. <return_value> -> expression
30. <return_value> -> id
31. <return_value> -> ε



S – Start
T1 – Token ID/KW
T2 – Token DEDENT/INDENT
T3 – Token LESS
T4 – Token LE_OR_EQ
T5 – Token GREATER
T6 – Token GR_OR_EQ
T7 – Token NEGATION
T8 – Token NOT_EQUAL
T9 – Token PLUS
T10 – Token MINUS
T11 – Token MULTIPLY
T12 – Token DIVIDE
T13 – Token DIV
T14 – Token COLON
T15 – Token COMMA
T16 – Token RBRACKET
T17 – Token LBRACKET
T18 – Token STRING
T19 – Token COMMENT
T20 – Token STRING_BLOCK
T21 – Token INT
T22 – Token FLOAT
T23 – Token INT/FLOAT
P1 – Previous state of FLOAT
P2 – Previous state of EXP_NUMBER
P3 – Previous state EXP_NUMBER_MARK
P4 – Previous state STRING_BLOCK_quote
P5 – Previous state STRING_BLOCK_quote_quote
P6 – Previous state STRING_BLOCK_quote_quote_quote
P7 – Previous state STRING_BLOCK_quote_quote_final
P8 – Previous state STRING_BLOCK_quote_final
P9 – Previous state of STRING
P10 – State BACKSLASH
P11 – State HEX_NUMBER
P12 – State HEX_NUMBER_IN

9. LL-tabulka

	def	EOL	id	expression	if	while	pass	EOF	return	COMMA	$
body	1	7	2	3	4	5	6	8			
def_params			26								25
body_after_indent		11	9	10	12	13	14		15		16
after_id			19	18							17
def_params_n										28	27
return_value			30	29							31
params			21	22							20
params_n										24	23

10. Precedenční tabulka

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

11. Závěr

Projekt IFJ hodnotíme ako veľmi dobrú skúsenosť a väčšinu času nás aj bavilo jeho riešenie. Prvýkrát sme si vyskúšali projekt tak veľkého rozsahu a zároveň prvýkrát sme na projekte pracovali v tíme. Tím bol častokrát aj prekážkou v efektívnom riešení, no učili sme sa tolerovať ostatných a pomáhať si.
V priebehu riešenia projektu sme narazili na niekoľko väčších prekážok, ktorých riešenie nás značne zdržalo, avšak dokázali sme sa s nimi v rámci možností úspešne popasovať. Nakoniec sme získali veľa nových vedomostí, ktoré sa nielen týkajú fungovania prekladačov, ale aj práce v tíme, či sme si lepšie vyskúšali prácu s verzovacím systémom.