Projektová dokumentácia

Implementácia prekladača imperatívneho jazyka IFJ25

Tým xklusaa00, varianta vv-BVS

Alexander Klusaček (xklusaa00) 25%
Michal Stanislav Málik (xmalikm00) 25%
Matej Mikuš (xmikusm00) 25%
Jan Černoch (xcernoj00) 25%

2. decembra 2025

Obsah

1. Úvod

Cieľom projektu bolo vytvoriť aplikáciu v jazyku C, ktorá dokáže spracovať zdrojový kód napísaný v jazyku IFJ25, čo je zjednodušená podmnožina jazyka Wren. Táto aplikácia preloží zdrojový kód do medzikódu v jazyku IFJcode25. Program funguje ako konzolová aplikácia, ktorá načíta zdrojový kód zo štandardného vstupu, vygeneruje medzikód na štandardný výstup a v prípade nájdenia chyby vráti príslušný chybový kód.

2. Implementácia

Implementácia sa skladá z niekoľkých úzko prepojených čiastkových krokov, ktoré sú opísané nižšie. Kto a akým spôsobom sa podieľal na jednotlivých častiach, je popísané v príslušnej kapitole.

2.1. Lexikálna analýza

Pri tvorbe prekladača sme začali spracovaním lexikálnej analýzy. Najprv sme si vytvorili štruktúru Token, ktorá obsahuje typ a hodnotu. Potom sme začali vytvárať hlavnú funkciu get_token() ako deterministický konečný automat podľa vopred vytvoreného diagramu. Konečný automat v jazyku C sme implementovali ako while cyklus, v ktorom máme switch, kde každý prípad case zodpovedá jednému stavu automatu. Ak je načítaný neznámy znak, ktorý nevyhovuje žiadnemu stavu, program sa ukončí a vracia chybu 1. Najväčší problém nám spôsobilo spracovanie substringu Ifj, tokenu bodky a tokenu identifikátoru dohromady ako jeden token identifikátor. Zároveň sme museli dbať na to, aby vo vnútri tohto spojenia neboli žiadne medzery.

2.2. Syntaktická analýza

Syntaktická analýza predstavuje centrálnu fázu prekladu, ktorá premosťuje lexikálne spracovanie a následnú sémantickú kontrolu. Vstupom do tohto modulu je postupnosť tokenov generovaná lexikálnym analyzátorom, pričom hlavným cieľom je overenie, či táto postupnosť zodpovedá formálnej gramatike jazyka IFJ25 . Okrem samotnej validácie syntaxe je kľúčovou úlohou parsera vybudovanie vnútornej reprezentácie programu – abstraktného syntaktického stromu (AST). Táto hierarchická stromová štruktúra slúži ako jediný zdroj dát pre generátor cieľového kódu a umožňuje efektívnu prácu s kontextom programu.

Architektúra syntaktického analyzátora je hybridná a kombinuje dva rozdielne prístupy. Hlavná riadiaca logika je implementovaná metódou rekurzívneho zostupu (Recursive Descent), čo je analýza typu Top-Down . Každému neterminálu v LL(1) gramatike zodpovedá konkrétna funkcia v kóde, ktorá na základe predikcie, rekurzívne volá ďalšie funkcie alebo konzumuje terminálne symboly.

Pre syntaktickú analýzu výrazu, kde je nutné rešpektovať prioritu a asociativitu operátorov, by bol rekurzívny zostup neefektívny. Preto je táto časť delegovaná na samostatný podmodul expression parser, ktorý využíva precedenčnú syntaktickú analýzu (metóda Bottom-Up) . Tento modul pracuje so zásobníkom (expr_stack), na ktorý ukladá prichádzajúce tokeny (operandy a operátory). Na základe pravidiel definovaných v precedenčnej tabuľke sa vykonáva redukcia postupnosti symbolov na vrchole zásobníka. Terminály sa postupne redukujú na uzly stromu (Nodes), čím vznikajú podstromy reprezentujúce čiastkové výpočty.

Komunikácia medzi hlavným parserom a expression parserom prebieha dynamicky. Keď rekurzívny parser narazí na začiatok výrazu (napríklad pri priradení alebo v podmienke), odovzdá riadenie precedenčnému analyzátoru. Ten spracuje celý výraz, vybuduje preň kompletný podstrom a vráti koreň tohto podstromu volajúcej funkcii. Hlavný parser následne tento podstrom integruje do celkového AST programu. Po úspešnom ukončení analýzy parser vráti návratový kód potvrdzujúci správnosť syntaxe a odovzdá ukazovateľ na koreňový uzol

vytvoreného AST (Program) pre ďalšie spracovanie.

2.3. Sémantická analýza

Pri implementácii sémantickej analýzy sme sa zhodli na tom, že by bolo ideálne, ak by sémantická analýza nemusela kontrolovať zbytočné veci, ale rovno nejakú štruktúru, k čomu sme vytvoril AST strom s určitými pravidlami, ktorý vytvára syntaktická analýza a dáva doň všetko potrebné ako pre sémantickú časť tak, takisto pre generátor kódu, s tým, že o niektoré vlastnosti sa obohatí v rámci behu sémantickej analýzy.

V rámci našej verzie sme využili AVL strom (symtable.c) ktorý slúžil k uchovaniu daných symbolov programu. Tu sme narazili na problém a to že každý blok môže mať rôzne symboly a preto sme zaviedli rozsahy (Scopes), ktoré vždy ukazovali na nadradený rozsah a každý mal vlastnú tabuľku symbolov a takto celkom jednoducho sme zabezpečili rozdiel symbolov rámci ich platnosti.

Celá sémantická analýza spočíva v dvoch priechodoch. Prvý priechod (semantic_definition()) zbiera všetky definície funkcií, getterov a setterov a registruje ich do globálnej tabuľky symbolov. Pri tom kontrolujeme duplicitu definícií a vytvárame preťažovacie kľúče vo formáte názov$počet_parametrov. Druhý priechod (semantic_visit()) vykonáva hlavnú sémantickú analýzu – kontrolu typov, validáciu volaní funkcií, inicializáciu premenných a správnosť návratových hodnôt. Tu sme využili rekurzívne volanie kontrol pre jednotlivé uzly v rámci nášho AST stromu, to znamená vždy bolo známe čo je naľavo a čo je napravo od stromu a podľa toho sa buď nastavil daný uzol alebo sa zavolala rekurzívna kontrola pre poduzly a v prípade že sa objavila nejaká chyba, tak sa poslal príslušný chybový kód, v prípade že všetko prebehlo bez problémov, tak sme mali správne overený a nastavený strom, ktorý mohol použiť generátor kódu.

2.4. Generovanie kódu

Generovanie kódu je posledná fáza prekladača. Vytvára výsledný program v jazyku IFJcode25 priamym prechodom cez AST strom. Generátor pracuje len s jednou dátovou štruktúrou, AST, a zapisuje inštrukcie do objektu FILE, ktorý reprezentuje výstup na STDOUT. Premenné sa generujú podľa rozsahu. Globálne premenné sa definujú sa v globálnom rámci. Ostatné premeny nesú doplnené číslo rozsahu, ktoré sa vypočíta podľa úrovne vnorenia v AST. Každá premenná sa pri definovaní inicializuje hodnotou nil, aby nebolo možné pristupovať k neinicializovaným dátam.

Generovanie postupuje cez funkciu next_step, ktorá rozlišuje typ uzla a podľa neho volá príslušný generátor. Priradenia vyhodnocujú výraz a hodnotu ukladajú do cieľovej premennej. Bloky zoskupujú viacero príkazov bez ďalších vedľajších efektov. Riadiace štruktúry používajú unikátne číslované návestia. Podmienka v if sa vyhodnotí v dočasnom rámci, kde sa určia jej typ a pravdivostná hodnota. Cyklus while vytvára návestie začiatku a konca a pri každom opakovaní znovu vyhodnocuje podmienku.

Výrazy sa generujú v postfixovej forme. Literály sa ukladajú na zásobník, premenné sa načítavajú z rámca a binárne operácie kontrolujú typy operandov. Na typovú kontrolu sa často používa dočasný rámec, kde sa určia typy oboch operandov pomocou inštrukcie TYPE. Operátor is má osobitnú implementáciu a porovnáva typový reťazec.

Definície funkcií, getterov a setterov používajú samostatné návestia. Pred vstupom do tela funkcie sa vytvorí nový rámec, premenné sa definujú a parametre sa načítajú zo zásobníka. Ak funkcia nevráti hodnotu, generuje návrat nil

Hlavná časť programu má návestie $$main. Po jej vykonaní generátor pridá ukončovacie inštrukcie a program sa ukončí návratovým kódom 0.

3. Špeciálne algoritmy a dátové štruktúry

3.1. Výškovo vyvážený binárny vyhľadávací strom

Pre uchovávanie dát tabuľky symbolov sme použili výškovo vyvážený binárny strom. Štruktúra uzla (SNode) obsahuje výšku, kľúč, ľavý a pravý uzol a dáta. Dáta sú tu prezentované ako union, kde rozlišujeme, či dáta patria funkcii, premennej, getteru alebo setteru. Ďalej dáta uchovávajú informácie o tom, či bola funkcia alebo premenná definovaná, použitá alebo aký má typ. Takisto sme si implementovali funkcie pre vytvorenie stromu, vytvorenie a vloženie dát do stromu, následné balancovanie a rotácie stromu na rôznych uzloch, aby bol strom výškovo vyvážený a ľahko sa v ňom hľadalo. Dodatočne sme ešte vytvorili funkciu my_strdup, pretože funkcia pre duplikáciu reťazca strdup nie je v C štandarde.

3.2. Dynamický reťazec

Táto štruktúra nám umožní pracovať s reťazcom, bez toho aby sme dopredu poznali jeho dĺžku. Používame ju predovšetkým pri ukladaní reťazca alebo identifikátora do hodnoty tokenu v lexikálnej analýze. Štruktúra v sebe uchováva reťazec, jeho dĺžku a maximálnu dĺžku, ktorá sa pri naplnení dynamicky zväčšuje. Ďalej sme implementovali niekoľko operácií s touto štruktúrou ako je alokácia, pridanie znaku, porovnanie alebo kópia.

3.3. Zásobník (*Stack*)

Zásobník je dátová štruktúra, ktorá funguje na princípe LIFO (Last In, First Out). V našom projekte sa zásobník využíva predovšetkým v syntaktickej analýze zdola-nahor, kde zásobník ukladá symboly a riadi aplikáciu gramatických pravidiel.

4. Práca v tíme

4.1. Spôsob práce v tíme

Práca na projekte bola rozdelená medzi členov tímu podľa jednotlivých fáz vývoja, pričom konkrétne úlohy boli pridelené po každom spoločnom zasadnutí tímu. Cieľom tohto prístupu bolo zabezpečiť efektívnu spoluprácu a zároveň rozdeliť prácu rovnomerne medzi všetkých členov.
Rozdelenie prebiehalo predovšetkým do menších skupín (obvykle dvoch až troch skupín), pričom každá skupina sa zameriavala na konkrétnu oblasť projektu. Tento spôsob práce umožnil členom tímu venovať sa špecializovaným úlohám a zabezpečiť hladší postup prác.

4.2. Verzovací systém

Počas projektu sme používali verzovací systém Git, ktorý nám umožnil efektívne pracovať paralelne pomocou viacerých vetiev, efektívne zdieľať kód a sledovať históriu zmien. Pre vzdialené ukladanie repozitára sme využili platformu GitHub.

4.3. Komunikácia

Tím na komunikáciu a koordináciu práce využíval primárne platformu Discord, ktorá poskytovala flexibilné prostredie pre všetky potrebné formy komunikácie. Na tejto platforme prebiehali všetky hovory a textové konverzácie, pokiaľ nebolo možné stretnúť sa osobne. Discord slúžil nielen ako komunikačný kanál, ale aj ako priestor pre zdieľanie dokumentov, poznámok, odkazov a dôležitých informácií týkajúcich sa projektu. Pokiaľ sa však objavil problém, bolo možné zvolať schôdzu aj mimo plán.

4.4. Rozdelenie práce medzi členmi tímu

Jan Černoch (xcernoj00)

Honza sa zameral na tvorbu lexikálnej analýzy a implementáciu tabuľky symbolov, pričom tiež prispel k vývoju funkcií v rámci práce s dynamickými reťazcami. Okrem technických úloh mal na starosti aj prípravu a spracovanie dokumentácie, ktorá opisuje implementačné procesy a výsledky projektu.

Matej Mikuš (xmikusm00)

Matej bol zodpovedný za kompletnú implementáciu syntaktickej analýzy. Vytvoril hlavný parser metódou rekurzívneho zostupu a pre spracovanie výrazov implementoval samostatný modul expr_parser založený na precedenčnej analýze. Súčasťou jeho práce bol aj návrh a realizácia dátových štruktúr – zásobníka pre analýzu výrazov (expr_stack) a definície uzlov pre abstraktný syntaktický strom (expr_ast).

Alexander Klusaček (xklusaa00)

Alex bol zodpovedný za kompletnú implementáciu generácie výsledného kódu a taktiež bol vedúcim týmu. Vytvoril celý generátor kódu, a taktiež spolupracoval s Mišom na návrhu abstraktného syntaktického stromu (ast) ktorý obsahoval celý spracovaný kód. Mimo programu taktiež organizoval stretnutia a podobne záležitosti.

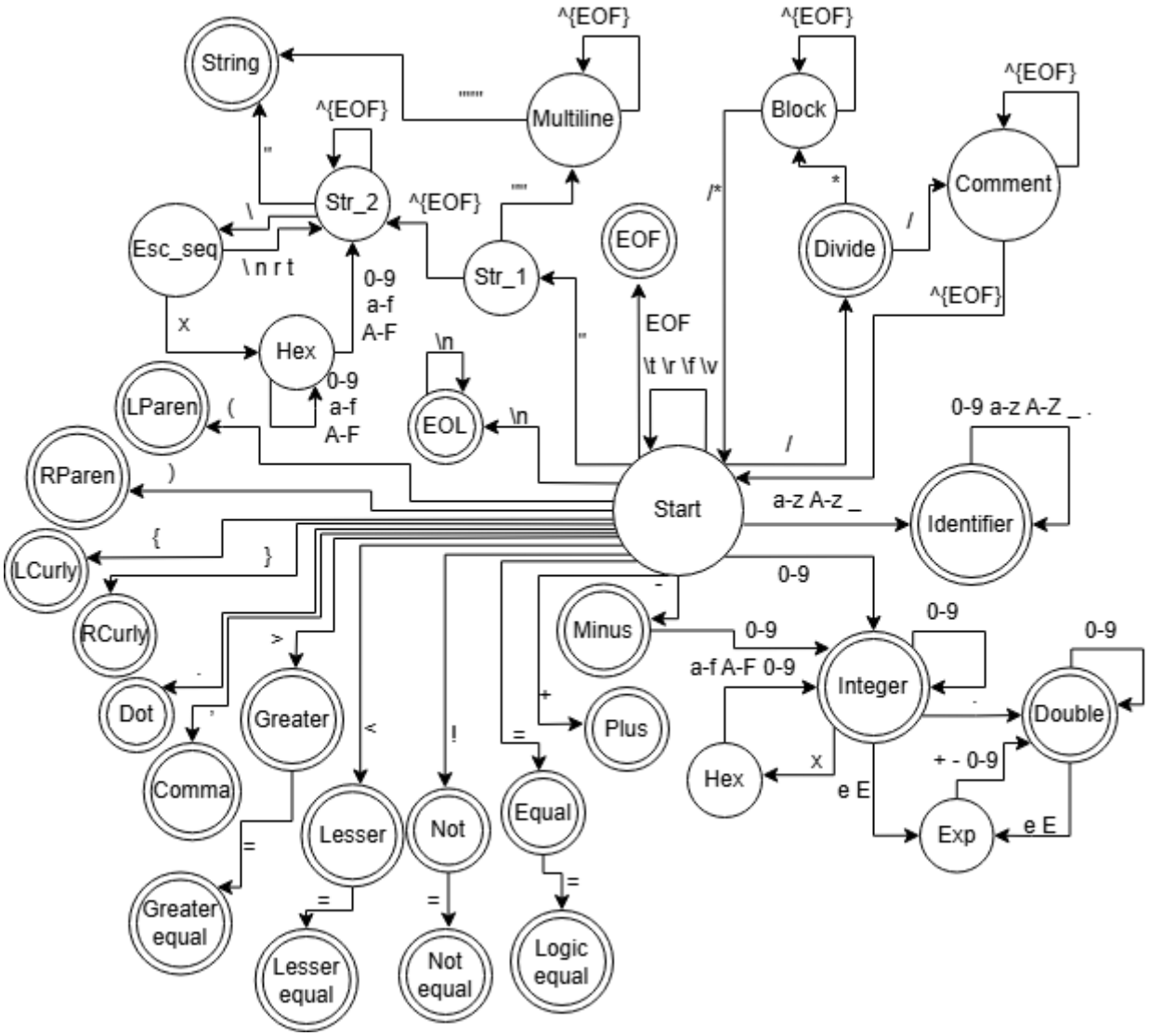
Michal Stanislav Málik (xmalikm00)

Michal mal za úlohu navrhnúť vhodnú štruktúru, ktorá by bola vytváraná syntaktickou analýzou pre reprezentáciu celého programu, ktorou sa stal AST strom s presne definovanou postupnosťou uzlou a následnou kontrolou daných uzlov a priradenie správnych informácii o jednotlivých symboloch pre generátor kódu.

5. Záver

Projekt, ktorého cieľom bolo vytvoriť prekladač pre jazyk IFJ25, bol pre náš tím výzvou, najmä vzhľadom na časové nároky spojené s polsemestrálnymi skúškami. Aj keď sme si vedomí niektorých problémov, veríme, že projekt splnil svoje hlavné ciele.
Tento projekt nám poskytol cenné skúsenosti, ako v technických zručnostiach, tak v tímovej spolupráci. Napriek malým nedostatkom sme vytvorili stabilný a funkčný základ.

6. Diagram konečného automatu pre lexikálnu analýzu



7. LL – gramatika

1. PROGRAM -> PROLOG eol CLASS eof
2. PROLOG -> import "ifj25" for Ifj
3. CLASS -> class Program { eol DEF_FUN_LIST }
4. DEF_FUN_LIST -> DEF_FUN DEF_FUN_LIST
5. DEF_FUN_LIST -> ε
6. DEF_FUN -> static id DEF_FUN_TAIL
7. DEF_FUN_TAIL -> (PARAMETER_ LIST) BLOCK eol
8. DEF_FUN_TAIL -> SETTER
9. DEF_FUN_TAIL -> GETTER
10. GETTER -> BLOCK eol
11. SETTER -> = (id) BLOCK eol
12. PARAMETER_LIST -> id PARAMETER_TAIL
13. PARAMETER_LIST -> ε
14. PARAMETER_TAIL -> , id PARAMETER_TAIL
15. PARAMETER_TAIL -> ε
16. BLOCK -> { eol STML_LIST }
17. STML_LIST -> STML_LINE STML_LIST
18. STML_LIST -> ε
19. STML_LINE -> STML eol
20. STML -> VAR
21. STML -> IF
22. STML -> WHILE
23. STML -> RETURN
24. STML -> id STML_ID
25. STML -> BLOCK
26. STML_ID -> (ARGUMENT _LIST)
27. STML_ID -> = EXPRESSION
28. VAR -> var id
29. IF -> if (EXPRESSION) BLOCK else BLOCK
30. WHILE -> while (EXPRESSION) BLOCK
31. ARGUMENT_LIST -> TERM ARGUMENT_TAIL
32. ARGUMENT_LIST -> ε

33. ARGUMENT_TAIL -> , TERM ARGUMENT_TAIL
34. ARGUMENT_TAIL -> ε
35. TERM -> literal_num
36. TERM -> literal_string
37. TERM -> literal_null
38. TERM -> id
39. RETURN -> return EXPRESSION
40. EXPRESSION -> #call EXPR_PARSER

**Pravidlo 40: Toto pravidlo je špeciálne prechodové pravidlo. V momente, keď LL(1) analyzátor narazí na neterminál EXPRESSION, pozastaví svoju činnosť a odovzdá riadenie modulu pre spracovanie výrazov.

8. LL – tabuľka

LL(1) Table	import	eol	class	static	id	,	{	}	(	)	var	if	else	while	return	literal_num	literal_string	literal_null	=
PROGRAM	1																		
PROLOG	2																		
CLASS			3																
DEF_FUN				6															
DEF_FUN_TAIL							9		7										8
DEF_FUN_LIST				4				5											
PARAMETER_LIST					12					13									
BLOCK							16												
PARAMETER_TAIL						14				15									
STML					24		25				20	21		22	23				
STML_ID									26										27
STML_LIST					17		17	18			17	17		17	17				
STML_LINE					19		19				19	19		19	19				
VAR											28								
IF												29							
WHILE														30					
RETURN															39				
ARGUMENT_LIST					31					32						31	31	31	
ARGUMENT_TAIL						33				34									
TERM					38											35	36	37	
EXPRESSION					40				40							40	40	40	
GETTER							7												
SETTER									8										

- Riadky tabuľky predstavujú neterminály , stav v ktorom sa parser nachádza.
- Stĺpce tabuľky predstavujú terminály (vstupné tokeny), ktoré prichádzajú z lexikálneho analyzátora.
- Obsahom buniek:
    - Čísla(1-40) - Indikuje číslo pravidla z gramatiky.
    - Eps (epsilon)
    - Prázdna bunka – predstavuje syntaktickú chybu

9. Precedenčná tabuľka

Precedence Table	TERM	+	-	*	/	(	)	<	>	<=	>=	is	==	!=	$
TERM	err	>	>	>	>	err	>	>	>	>	>	>	>	>	>
+	<	>	>	<	<	<	>	>	>	>	>	>	>	>	>
-	<	>	>	<	<	<	>	>	>	>	>	>	>	>	>
*****	<	>	>	>	>	<	>	>	>	>	>	>	>	>	>
/	<	>	>	>	>	<	>	>	>	>	>	>	>	>	>
(	<	<	<	<	<	<	=	<	<	<	<	<	<	<	err
)	err	>	>	>	>	err	>	>	>	>	>	>	>	>	>
<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	>
>	<	<	<	<	<	<	>	>	>	>	>	>	>	>	>
<=	<	<	<	<	<	<	>	>	>	>	>	>	>	>	>
>=	<	<	<	<	<	<	>	>	>	>	>	>	>	>	>
is	<	<	<	<	<	<	>	>	>	>	>	>	>	>	>
==	<	<	<	<	<	<	>	>	>	>	>	>	>	>	>
!=	<	<	<	<	<	<	>	>	>	>	>	>	>	>	>
$	<	<	<	<	<	<	T	<	<	<	<	<	<	<	err

- < (Shift) – Znamená, že operátor na vstupe má vyššiu prioritu. Token zo vstupu sa posunie na zásobník.
- > (Reduce) - Znamená, že operátor na vstupe má nižšiu prioritu. Na zásobníku sa nájde (Handle) < a ohraničenú časť zredukujeme podľa pravidiel.
- = (Match) – Znamená rovnosť priorít, používa sa pri párovaní zátvoriek.
- err – Znamená neplatná kombinácia znakov. Vedie ku syntaktickej chybe.
- T(Terminate) - pravidlo na ukončenie expr_parser

10. Zdroje

1) Prednášky, materiály a záznamy z predmetu IFJ
2) https://wren.io/syntax.html