Dokumentace k projektu IFJ24

Implementace překladače imperativního jazyka IFJ24

Tým: xukropj00
Varianta: vv-BVS

Glos Robert	Pazúr Marek	Tomaško Patrik	**Ukropec Jan**
xglosro00	xpazurm00	xtomasp00	xukropj00
25%	25%	25%	25%

4. prosince 2024

Implementovaná rozšíření
Žádná

Obsah

1 Práce v týmu

1.1 Rozdělení práce v týmu

Marek Pazúr

- Návrh a implementace lexikálního analyzátoru
- Precedenční syntaktická analýza
- Návrh a implementace sémantické analýzy kódu
- Struktura projektu, podpůrné struktury a funkce

Patrik Tomaško

- Částečná účast na implementaci lexikálního analyzátoru
- Dynamické pole
- Syntaktická analýza (kromě výrazů) a sémantické akce během ní

Robert Glos

- Testování jednotlivých částí a překladače jako celku

Jan Ukropec (vedoucí)

- Návrh a implementace vyhledávací tabulky
- Struktura abstraktního syntaktického stromu
- Generátor cílového kódu

1.2 Způsob práce v týmu

Práce na projektu byla zahájena pár dní po zveřejnění zadání, kdy proběhla diskuse o projektu jako celku, poté následoval předběžný návrh struktury programu a přidělování práce členům týmu na základě dohody, včetně stanovení termínů pro dokončení přidělených částí.

Komponenty překladače byly vypracovávány buďto samostatně, nebo v tandemu, ve kterém se členové doplňovali.
V případě nejasností a problémů proběhly online schůzky za účelem vyjasnění, popřípadě vyřešení problémů.

1.3 Vývojový cyklus a komunikace

Týmové schůzky se uskutečňovaly každý týden ve čtvrtek, během nichž byla diskutována problematika dílčích částí projektu a jejich možné implementace.

Verzovací systém Git byl vyhodnocen všemi členy týmu jako ten nejvhodnější. Umožnil paralelní vývoj jednotlivých komponent překladače současně pomocí větvení a následné integraci kódu na vzdálený repozitář platformy GitHub.

Komunikace mezi členy probíhala téměř na denní bázi, primární platformou pro komunikaci byl zvolen Discord.
Byl kladen přísný důraz na řádné otestování komponent při provádění zásadních úprav.

2 Lexikální analýza

Implementace lexikálního analyzátoru se nachází ve zdrojovém souboru `lexer.c`, jeho výstupem je datová struktura token_t, definovaná v hlavičkovém souboru `token.h`.

Struktura token_t obsahuje následující položky

- Identifikátor příslušného typu tokenu - položka výčtu token_id, definovaným ve stejnojmenném hlavičkovém souboru `token.h`
- Dynamické pole obsahující, v případě potřeby (literál, identifikátor, klíčové slovo), načtený lexém jako řetězec znaků - implementováno ve zdrojovém souboru `dynamic_array.c`

Klíčovou funkcí je `get_token`, jejímž voláním se spustí proces lexikální analýzy a po jeho úspěšném dokončení funkce vrátí hodnotou příslušný token, který byl načtený ze vstupního souboru.

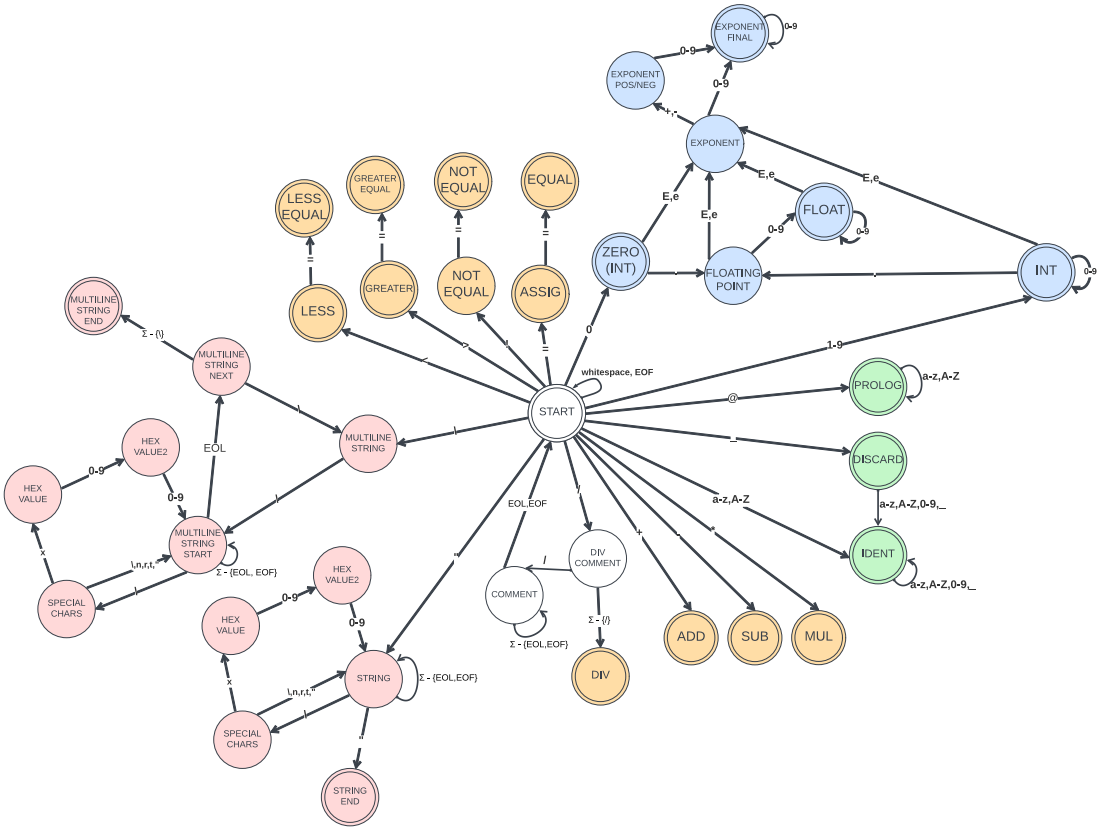
Lexikální analyzátor byl navržen jako deterministický konečný stavový automat na základě předem připraveného diagramu[2.1].

Samotná implementace se nachází uvnitř těla již zmíněné funkce `get_token` - struktura automatu je realizována pomocí konstrukce *switch-case* uvnitř cyklu *while*, kde jsou ze vstupního souboru načítány jednotlivé znaky, dokud není vstupní řetězec přijat, nebo dokud nenastane chyba, přičemž se podle přechodových pravidel přechází do příslušných stavů.

Je také vhodné podotknout, že v případě stavů značících *literály, identifikátor, prolog a klíčové slovo* dochází k ukládání znaků ze vstupní pásky na konec dynamického pole v příchozím pořadí, poté při posledním přečteném znaku dojde k porovnání tohoto lexému v případě potřeby (*identifikátor, prolog*) s klíčovými slovy kvůli rozlišení správného typu výsledného tokenu.

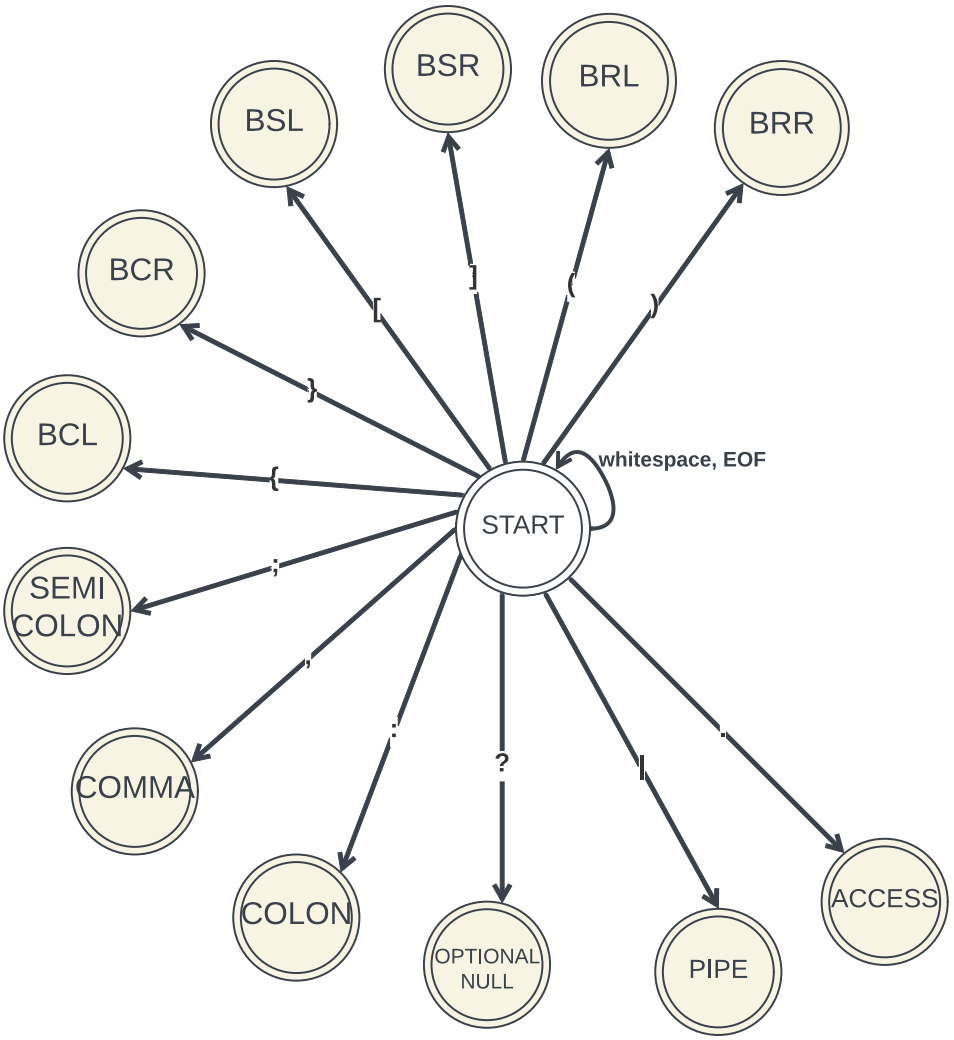
V případě, že automat úspěšně přijme řetězec získaný ze vstupu, funkce vrátí hodnotou token daného typu, jinak vrátí token typu *token_error* a nastaví příslušný chybový stav.

2.1 Konečný stavový automat lexikální analýzy



Legenda
Modrá - větve zabývající se literály celočíselných a desetinných čísel
Červená - větve zabývající se literály standardních a víceřádkových řetězů
Zelená - větve zabývající se indentifikátory a klíčovými slovy
Oranžová - větve zabývající se aritmetickými a logickými operátory
EOF - konec souboru
EOL - konec řádku
Σ - všechny znaky použité abecedy
Σ-{a,b,c} - vše mimo znaky ve složené závorce
a,c,d,... - kterýkoliv z uvedených znaků
a-z - kterýkoliv znak v uvedeném rozmezí

2.2 Konečný stavový automat - pokračování



Pozn. tohle je automat tvořený pouze z jednoduchých stavů, aby předchozí diagram nebyl vizuálně přeplněný.

Legenda
BRL - Bracket Round Left "("
BRR - Bracket Round Right ")"
BCL - Bracket Curly Left "{"
BCR - Bracket Curly Right "}"
BSL - Bracket Square Left "["
BSR - Bracket Square Right "]"

3 Synktatická a sémantická analýza

Syntaktická analýza je rozdělená mezi dva hlavní páry souborů, prvním párem jsou `syna.c` spolu s `syna.h`, který obsahuje všechny funkce, které zprostředkovávají analýzu shora dolů, druhým párem souborů je `precedence.c` a `precedence.h`, které obsahují funkce, jenž zprostředkovávají precedenční syntaktickou analýzu.

Syntaktická analýza probíhá jedním během přes zdrojový kód, během kterého se zároveň vytváří abstraktní syntaktický strom *(AST)* a zároveň probíhají určité sémantické kontroly. Také se vytváří tabulky symbolů, které jsou následně naplňovány informacemi o funkcích/proměnných.

3.1 Syntaktická analýza shora dolů

Syntaktická analýza shora dolů je řešena LL gramatikou (3.2), kdy se využívá rekurzivní sestup do jednotlivých funkcí na základě LL tabulky (3.3).

Syntaktická analýza se spouští voláním funkce `init_parser`, která zároveň inicializuje samotnou datovou strukturu parser. Pro chod syntaktické analýzy obsahuje parser stav typu `Pfsm_state_syna` a strukturu current_token typu `token_t`, která obsahuje právě zpracovaný token. V syntaktické analýze se vyskytují funkce uvedené v LL gramatice (3.2).
Během každého kroku syntaktické analýzy se volá funkce `get_token` ze souboru `lexer.c`, která parseru dodá další token v pořadí. Token se uchovává v již zmíněné struktuře `current_token`.

3.1.1 Řešení sémantické analýzy za průchodu zdrojovým kódem

Sémantické chyby řešené v rámci souboru `syna.c` zahrnují zejména problémy spojené s plněním tabulky symbolů a situacemi, kdy by kontrola prostřednictvím abstraktního syntaktického stromu (AST) byla zbytečně komplikovaná. Mezi tyto kontroly patří ověřování existence proměnných či konstant použitých v rámci výrazů, kontrola redefinice proměnných a konstant a u konstant navíc probíhá dodatečná kontrola, která zajišťuje, že se v kódu nemůže vyskytnout přiřazení konstantě.

3.1.2 Tvorba abstraktního syntaktického stromu

Během syntaktické analýzy je zároveň tvořen abstraktní syntaktický strom, který reprezentuje strukturu zdrojového kódu v podobě binárního stromu. Tento strom je dále využíván v semantické analýze a při generaci kódu. Generování probíhá tím, že při rekurzivním sestupu je volané funkci vždy předán uzel, na který má napojit nový podstrom, který funkce vytvoří.
Strom je tvořen uzly různých typů. Kořenem stromu vždy bývá kořen typu PROGRAM. Napravo od kořenového uzlu se vždy nachází uzel s prologem. Nalevo od kořene stromu se nachází řada na sebe navazujících uzlů (vždy přes levého potomka) typu FN, které reprezentují jednotlivé funkce zdrojového kódu.
Pravým potomkem uzlu FN je vždy řada uzlů typu COMMAND (navazujících na sebe právým potomkem), každý z těchto uzlů reprezentuje jeden příkaz v těle funkce, levým potomkem uzlu COMMAND je vždy konkrétní instance příkazů, který se dále člení dle příslušného typu.
Těla typu *if, else, while* a *bezhlavičková těla* jsou reprezentována podobným způsobem jako funkce, nutno podotknout, že všechny výše zmíněné příkazy s vlastním tělem obsahují uvnitř struktury svého uzlu položku typu `scope_t`. Jedná se o datovou strukturu, jenž obsahuje odkaz na lokální tabulku symbolů - rozsah platnosti proměnných a konstant uvnitř daného těla. Další položkou je rekurzivní odkaz na strukturu stejného typu, která reprezentuje rozsah platnosti nadřazeného těla - **zřetězená hierarchie rozsahu platnosti**.

3.2 LL-gramatika

1. ⟨root_code⟩ → ⟨import_func⟩ ⟨root_code⟩
2. ⟨root_code⟩ → ⟨function_header⟩ ⟨root_code⟩
3. ⟨root_code⟩ → EOF
4. ⟨import_func⟩ → const identifier = @import (literal_string) ;
5. ⟨function_header⟩ → pub fn identifier (⟨function_params⟩) ⟨func_return⟩ { ⟨body⟩ }
6. ⟨var_type⟩ → i32
7. ⟨var_type⟩ → f64
8. ⟨var_type⟩ → []u8
9. ⟨func_return⟩ → void
10. ⟨func_return⟩ → ⟨nullable_sign⟩ ⟨var_type⟩
11. ⟨nullable_sign⟩ → ϵ
12. ⟨nullable_sign⟩ →?
13. ⟨function_params⟩ → ϵ
14. ⟨function_params⟩ → identifier : ⟨nullable_sign⟩ ⟨var_type⟩ ⟨function_params_n⟩
15. ⟨function_params_n⟩ →, ⟨function_params⟩
16. ⟨function_params_n⟩ → ϵ
17. ⟨body⟩ → ϵ
18. ⟨body⟩ → while ⟨if_while_header⟩ ⟨body⟩
19. ⟨body⟩ → if ⟨if_while_header⟩ ⟨possible_else⟩ ⟨body⟩
20. ⟨body⟩ → _ = ⟨expression⟩; ⟨body⟩
21. ⟨body⟩ → return ⟨expression⟩; ⟨body⟩
22. ⟨body⟩ → { ⟨body⟩ } ⟨body⟩
23. ⟨body⟩ → var ⟨var_const_declaration⟩; ⟨body⟩
24. ⟨body⟩ → const ⟨var_const_declaration⟩; ⟨body⟩
25. ⟨body⟩ → identifier ⟨identifier_followup⟩; ⟨body⟩
26. ⟨identifier_followup⟩ →= ⟨expression⟩
27. ⟨identifier_followup⟩ → (⟨function_call_params⟩)
28. ⟨identifier_followup⟩ → . ⟨function_call⟩
29. ⟨possible_else⟩ → else { ⟨body⟩ }
30. ⟨possible_else⟩ → ϵ
31. ⟨if_while_header⟩ → (⟨expression⟩) ⟨null_replacement⟩ { ⟨body⟩ }
32. ⟨null_replacement⟩ → | identifier |
33. ⟨null_replacement⟩ → ϵ
34. ⟨var_const_declaration⟩ → identifier ⟨possible_var_type⟩ = ⟨expression⟩
35. ⟨possible_var_type⟩ →: ⟨nullable_sign⟩ ⟨var_type⟩
36. ⟨possible_var_type⟩ → ϵ
37. ⟨function_call⟩ → identifier(⟨function_call_params⟩)
38. ⟨function_call_params⟩ → ϵ
39. ⟨function_call_params⟩ → ⟨input_param⟩ ⟨function_call_params_n⟩
40. ⟨function_call_params_n⟩ →, ⟨function_call_params⟩
41. ⟨function_call_params_n⟩ → ϵ
42. ⟨input_param⟩ → identifier
43. ⟨input_param⟩ → literal_i32
44. ⟨input_param⟩ → literal_f64
45. ⟨input_param⟩ → literal_string
46. ⟨input_param⟩ → null

3.3 LL-tabulka

	identifier	EOF	=	const	var	const	pub	void	i32	f64	[]u8	?	(	)	,	.	—	:	while	if	else	}	_	return	{	literal_i32	literal_f64	literal_string	null
<root_code>		3		1			2																						
<import_func>				4																									
<function_header>							5																						
<var_type>									6	7	8																		
<nullable_sign>									11	11	11	12																	
<func_return>								9	10	10	10	10																	
<function_params>	14													13															
<function_params_n>														16	15														
<body>	25				23	24													18	19		17	20	21	22				
<identifier_followup>			26										27			28													
<possible_else>	30				30	30													30	30	29	30	30	30	30				
<if_while_header>													31																
<null_replacement>																	32								33				
<var_const_declaration>	34																												
<possible_var_type>			36															35											
<function_call>	37																												
<function_call_params>	39													38												39	39	39	39
<function_call_params_n>														41	40														
<input_param>	42																									43	44	45	46

3.4 Precedenční syntaktická analýza

Pokud je za daným tokenem očekáván výraz (pravidlo v LL-gramatice), předá aktuální metoda rekurzivního sestupu řízení precedenční syntaktické analýze. Cílem je ověřit syntaktickou korektnost výrazu a sestavení abstraktního syntaktického stromu, který je zároveň **výstupem** precedenční analýzy.

Nejprve se zavolá funkce `expression` implementovaná v souboru `syna.c`, kde dojde k vyhodnocení, zda se jedná o výraz, nebo o volání funkce. Rozhodnutí se provádí podle následujících dvou načtených tokenů. Aby byly tokeny uchovány pro pozdější použití při precedenční analýze, ukládají se do dynamické fronty tokenů - `t_buf`, implementované ve zdrojovém souboru `token.c`.

Pokud se jedná o výraz, následuje volání hlavní funkce precedenční analýzy - `precedent`, implementované ve zdrojovém souboru (včetně všech pomocných funkcí) `precedent.c`, s frontou tokenů a očekávaným ukončovacím znakem jako parametry.

Precedenční syntaktická analýza probíhá následujícím způsobem - pokud je fronta tokenů neprázdná, čte z ní tokeny funkcí `fetch_token`. Jakmile je prázdná, získává tokeny ze vstupu funkcí `get_token`, které převede pomocí převodové funkce `token_to_symbol` na datovou strukturu `symbol` - definovanou v hlavičkovém souboru `symbol.h`, jejímž cílem je zvýšení abstrakce a zjednoduššení precedenční analýzy. Zároveň je k dispozici zásobník těchto symbolů implementovaný v zdrojovém souboru `precedent.c`.
Dokud se nejvrchnější terminál (reprezentovaný symbolem) zásobníku a zároveň příchozí terminál nerovnají ukončovacímu znaku - $ (';' nebo ')'), tak se jako parametry, pomocí mapovací funkce `pt_map` namapují na příslušný index do precedenční tabulky, kde již leží daná priorita.
Na základě této priority se vyvolá daná procedura

- < Nižší priorita, způsobí volání procedury `shift`, která před nejvrchnější terminál na zásobníku symbolů vloží symbol posunu (shift) R
- > Vyšší priorita, způsobí volání procedury `reduce`, která na základě počtu symbolů ke zredukování nejprve vyhodnotí správnost podvýrazu, poté v případě úspěchu symboly zredukuje podle gramatických pravidel na neterminál E, odstraní symbol posunu a na závěr vytvoří uzel pro AST, tímto je aktuální podvýraz zpracován
- = Ekvivalentní priorita, způsobí volání procedury `equal`, která na vrchol zásobníku pouze vloží příchozí symbol
- e - Chyba, nastává v případě nepovolenné kombinace terminálů

Jakmile je nejvrchnější i příchozí terminál roven symbolu $ - obsah zásobníku ve tvaru *E*, syntaktická analýza proběhla úspěšně, jinak končí chybou.

3.4.1 Precedenční tabulka

	+	*	r	(	)	i	$
+ -	>	<	>	<	>	<	>
* /	>	>	>	<	>	<	>
r	<	<		<	>	<	>
(	<	<	<	<	=	<	
)	>	>	>		>		>
i	>	>	>		>		>
$	<	<	<	<		<	

Tabulka 1: Precedenční tabulka výrazů (řádek - vrchol zásobníku, sloupec - příchozí symbol)

Legenda
r - relační operátory >, <, >=, <=, ==, !=
i - literály, identifikátory konstant či proměnných

4 Tabulka symbolů

4.1 Implementace

Tabulka symbolů je implementovaná jako výškově vyvážený binární vyhledávací strom s řetězcovými klíči v souboru `symtable.c` s rozhraním v souboru `symtable.h`. Každý uzel stromu uchovává ukazatel na levý a pravý podstrom a data. Data popisují buď proměnnou, konstantu nebo funkci.

4.2 Data

U funkce:

- Typy parametrů
- Návratová hodnota
- Jestli vrací null
- Lokální tabulka symbolů

U proměnné:

- Datový typ
- Jestli může nabývat null
- Jestli byla použita
- Jestli byla změněna
- Jestli má hodnotu známou při překladu

U konstanty:

- Datový typ
- Jestli byla použita
- Jestli má hodnotu známou při překladu

4.3 Rozhraní

Binární strom je obalen ve struktuře `symtable`, se kterou uživatel pracuje. Prázdnou tabulku lze vytvořit pomocí funkce `symtable_init`. Před koncem programu je nutno ji uvolnit pomocí funkce `symtable_free`. Data se vkládají pomocí funkce `symtable_insert`, při vložení s klíčem, který se již v tabulce nachází, se pouze přepíší data. Dále je k dispozici funkce pro mazání `symtable_delete`, pro vyhledání dat `symtable_search` a pro získání dat `symtable_get_data`.

4.4 Vkládání do tabulek symbolů

Tabulky symbolů se inicializují a naplňují během syntaktické analýzy. V programu vždy existuje právě jedna globální tabulka symbolů, která obsahuje záznamy o funkcích. Dále každé tělo funkce, if, else, while a bezhlavičkové tělo má svou lokální tabulku symbolů, která obsahuje záznamy o lokálně deklarovaných proměnných a konstantách.

5 Sémantická Analýza Abstraktního Syntaktického Stromu

Zbylé sémantické kontroly probíhají na již sestaveném AST stromu, kvůli nutnosti sběru a doplnění všech potřebných informací do tabulek symbolů, k provedení zbývajících kontrol, které se nemohly uskutečnit při syntaktické analýze. Kontroly jsou implementovány ve zdrojovém souboru `semantic.c` s rozhraním v hlavičkovém souboru `semantic.h`.

5.1 Průchod AST v rámci sémantické analýzy

AST je procházen kombinací iterativního a rekurzivního průchodu - provádí se sémantické kontroly jednotlivých funkcí a příkazů uvnitř jejich těl pomocí funkcí `FunctionSemantics`, `CommandSemantics`. V případě, že má příkaz vlastní tělo, se rekurzivně volá funkce `CommandSemantics` a po dokončení kontrol všech příkazů v podbloku se vrátí řízení volajícímu.

5.2 Sémantické kontroly a akce

Sémantické kontroly a akce probíhají u

- Definic funkcí - kontrola chybějícího nebo přebývajícího výrazu v příkazu návratu z funkce.
- Volání funkcí - kontrola správného počtu a typu parametrů, správného typu návratové hodnoty a kontrola, zda může být zahozena. Ověření, že volaná funkce je definována.
- Deklarací a přiřazení - kontrola kompatibility typu proměnné a výrazu či návratové hodnoty funkce. Ověření možnosti odvodit typ proměnné/konstanty z inicializačního/přiřazovaného výrazu.
- Výrazů - typová kontrola a kompatabilita operandů. V případě potřeby dojde k implicitnímu přetypování operandů. Pokud přetypování nelze provést, dojde k chybě.
- Proměnných a konstant - Kontrola využití proměnných a konstant v jejich rozsahu platnosti. U proměnných se kontroluje možnost změny po inicializaci.
- Hlaviček podmíněných příkazů - kontrola očekávaného typu výrazu. V případě, že je příkaz podmíněný neprázdnou hodnotou, proběhne aktualizace záznamu o typu konstanty v její tabulce symbolů.
- Návratový příkaz - Ověření shody typu výrazu (nemusí být uveden) s návratovým typem funkce.

Kontroly týkající se volání funkcí, proměnných či konstant využívají své příslušné tabulky symbolů. U funkcí se jedná o globální proměnnou `globalSymTable` odkazující na globální tabulku symbolů obsahující informace o všech definovaných funkcích. U proměnných se využívá datová struktura `scope_t`[3.1.2] pro přístup k hierarchii rozsahu platnosti.

Součástí analýzy je ověření existence definice hlavní funkce *main*. Hlavička její definice nesmí obsahovat žádné parametry ani návratovou hodnotu.

Výstupem sémantické analýzy při úspěšném dokončení všech sémantických kontrol a akcí je **aktualizovaný** abstraktní syntaktický strom.

6 Generace kódu

Generátor kódu a jeho rozhraní jsou uloženy v souborech `codegen.c` a `codegen.h`. Generování kódu probíhá až poté, co je dokončena syntaktická a sémantická analýza. Rozhraní obsahuje pouze jednu funkci – codegen, která přijímá jako parametr abstraktní syntaktický strom (dále jen AST), na základě kterého kód generuje a vypisuje ho na standardní výstup.

6.1 Rozhraní

Funkce codegen na začátku vypíše příkaz pro inicializaci interpretu, vytvoří pomocné proměnné na globálním rámci, vypíše příkaz pro zavolání funkce main, následně příkaz pro ukončení interpretace s chybovou hodnotou 0 – úspěch. Poté se generují pomocí AST uživatelem definované funkce včetně funkce main. Na závěr se vygenerují vestavěné funkce jazyka IFJ24.

6.2 Pojmenování proměnných a návěští

Pro snadné rozlišení a zamezení konfliktům se názvy generují následovně. Názvy proměnných se generují s předponou `VAR_`, návěští funkcí se generují s předponou `FUN_` (u vestavěných funkcí jazyka IFJ24 s předponou `FUN_ifj-`), ostatní návěští se generují ve formátu `Lid`, kde `id` je nezáporné celé číslo – pořadí, ve kterém bylo návěští vygenerováno.

6.3 Zamezení opakovaným deklaracím proměnných

Aby se zabránilo opakovaným deklaracím na lokálním rámci např. při deklaraci proměnné se stejným názvem ve dvou blocích kódu, uchovávají se při generaci funkce ve stromové struktuře. Pokud se v této struktuře již nachází, nebudou znovu generovány.
Aby se zabránilo tomuto problému i u while cyklů, vygenerují se všechny deklarace proměnných před generací cyklu.

7 Závěr

Cílem projektu bylo dokončit kompletní překladač včětně rozšíření se stanoveným termínem prvního pokusného odevzdání. Toto se bohužel nevyplnilo v důsledku objevujících se problémů a nedostatku času.
Nastávaly různé odchylky od počáteční představy o celkovém fungování a struktuře projektu, což vedlo k drobným až zásadním změnám struktury překladače a jeho komponent.
Projekt byl velice přínosný z hlediska nabytých zkušeností a vědomostí, jelikož se jednalo o zatím nejkomplexnější projekt, se kterým se každý člen týmu doposud setkal.

Reference

Materiály z přednášek a demonstračních cvičení.