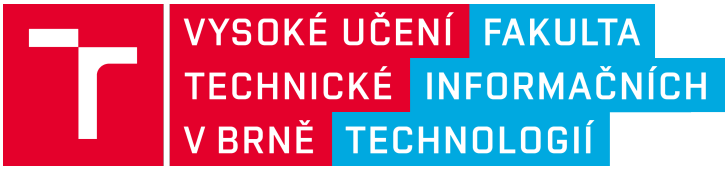

IFJ Projekt

Babuljak, Milan - xbabulm00 - 25% [VEDÚCI]
Mesík, Juraj - xmesikj00 - 25%
Rous, Jan - xrousja00 - 25%
Chalupka, Marek - xchalu18 - 25%

5. decembra 2024

Obsah

Úvod

Rozdelenie práce

- Milan Babuljak - lexikálna analýza (tokeniser.c, tokeniser.h), generátor kódu (codegen.c, codegen.h)
- Juraj Mesík - syntaktická analýza pre výrazy (psa.c, psa.h, prec_table.c, prec_table.h expr_stack.c expr_stack.h)
- Jan Rous - sémantická analýza (symtable.c, symtable.h)
- Marek Chalupka - syntaktická analýza okrem výrazov (parser.c, parser.h)

Rozdelenie úloh je len hrubý odhad kto sa na čom podieľal najviac. Počas práce sme si často vzájomné pomáhali s jednotlivými časťami. Na dokumentácii a testoch sme pracovali spoločne.

Funkcionalita

Na nasledujúcom jednoduchom programe v jazyku v IFJ24, by sme vám radi demonštrovali funkcionalitu našeho prekladača. Tento rozbor by mal pomôcť k hlbšiemu pochopeniu prekladača.

```
const ifj = @import("ifj24.zig");

pub fn main() void {
    ifj.print("Hello, world!\n");
}
```

Syntaktická analýza začne volaním funkcie start() priamo z main.c. Tá inicializuje štruktúru typu tParser ktorá obsahuje všetky potrebné dáta pre prechádzanie tokenmi, a to súčasný token, nasledujúci token, ukazatel na vstupný súbor, tabuľku symbolov pre aktuálny rámec, zásobník tabuliek symbolov. Premenná typu tParser je inicializovaná ako ukazovateľ a je parametrom každej funkcie v parser.c súbore. Vďaka tomuto prístupu sme schopný spoľahlivo prechádzať vstupný súbor a aktualizovať hodnoty aktualneho tokenu.

Týmto spôsobom zaistíme, že pri nesprávnom poradí tokenov, program vráti syntaktickú chybu. Tokeny prechádzame vďaka funkcii update(), ktorá nastaví súčasný token na nasledujúci. Zoznam tokenov sme používali pri debuggovani a získali sme ho jednoducho vypisovaním aktuálneho tokenu vo funkcii update() za pomoci tokenTypeToString() funkcie, ktorá vracia reťazec podľa načítanej celočíselnej hodnoty tokenu.

Tokenizer

`tokeniser.c, tokeniser.h`

Prvý krok nášho riešenia je tokenizer (scanner). Jeho úloha je "rozsekať"kód na menšie časti, ktoré parser následovne skontroluje.

Náš jednoduchý program obsahuje niekoľko základných elementov:

- **Kľúčové slová**: `const`, `pub`, `fn`, `void`.
- **Identifikátory**: `ifj`, `main`, `print`.
- **Reťazcový literál (String Literal)**: `"Hello, world!\n"`.
- **Separátory a operátory**: `=`, `{}`, `()`, `;`.
- **Špeciálne konštrukcie**: `Prolog`.

Tieto tokeny sa potom vracaju postupne pomocou následujúcej štruktúry struct token:

- Typ tokenu (`token_type`) – napr. kľúčové slovo, identifikátor, operátor, atď.
- Dĺžku parametra (`param_length`) – dĺžka textového obsahu parametra tokenu.
- Obsah tokenu (`param[]`) – dynamicky alokovaný reťazec reprezentujúci parameter tokenu (napr. pri string literal sa jedná o samotný string).

Spracovanie jednotlivých kategórií

Funkcia `getToken`, ktorá je vyvolavaná neskôr v parseri postupne číta znaky zo súboru, a rozdeluje ich pomocou nasledujúcich pravidiel:

- **Identifikátory a kľučové slová**: Pokial program narazí na alfanumerické sekvencie začínajúce písmenom alebo znakom `_`, cyklí dokým nenarazí na medzeru. Následovne sa overí, či sa nejedná o kľučové slovo. Program obsahuje preddefinovaný zoznam kľúčových slov, ktoré sú kontrolované funkciou `isKeyword`. Pokiaľ sa jedná o kľučové slovo, vytvorí sa špecifický token pre dané kľučové slovo (napr. TOKEN_CONST). V opačnom prípade sa vytvorí token TOKEN_IDENTIFIER s parametrom.
- **Čísla (INT a FLOAT)**: Pokial program narazí na číslo, číta ďalej, dokým nenarazí na nečíselný charakter. Následovne skontroluje, či sa jedná o bodku, alebo o písmeno E. Ak je následujúcí charakter medzera alebo ;, vytvorí sa TOKEN_INT_LITERAL. Pokial sa narazí na charakter e alebo E, vytvorí sa TOKEN_INT_FLOAT.
- **Operátory a separátory**: Napr. `=`, `+`, `{`, `}` - pokial sa narazí na špecialny charakter, skontroluje sa pomocou funkcie `nextChar` další znak (pre prípady <=). Následovne sa vytvorí token pre daný operátor (napr. TOKEN_LESS_EQUAL)
- **Komentáre**: `//` - Pokiaľ sa prečíta znak /, pomocou funkcie `nextChar` sa prečíta následujúci znak, a pokiaľ sa jedná znova o znak /, program ignoruje zvyšok riadku
- **Reťazce**: Uvedené v úvodzovkách `"..."` - program číta a zapisuje reťazec do dočasného buffera, ktorý sa vracia ako parameter. Špecialny prípad nastáva pri tzv. escape sekvenciach (napr. n.) V tomto prípade sa opať použije funkcia `nextChar`, a správne odignoruje charakter.

Funkcia `nextChar` číta jednotlivé znaky zo súboru, pričom umožňuje vrátenie znaku späť do vstupu pomocou funkcie `pushBack`. Toto zabezpečuje, že pokiaľ máme nevyhovujúci vstup, vieme navrátit charakter spať do vstupného buffera pre dialšie spracovanie.

Na konci procesu program dynamicky alokuje nový token, nastaví jeho typ a obsah a následne ho vráti.

Ukážka analýzy kódu

Pre vyššie uvedený vstupný kód by tokenizér vytvoril nasledujúcu sekvenciu tokenov:

Typ tokenu	**Obsah**
TOKEN_KEYWORD_CONST	`const`
TOKEN_IDENTIFIER	`ifj`
TOKEN_ASSIGN	`=`
TOKEN_IMPORT	`@import`
TOKEN_LEFT_PAREN	`(`
TOKEN_STRING_LITERAL	`" ifj24.zig "`
TOKEN_RIGHT_PAREN	`)`
TOKEN_SEMICOLON	`;`
TOKEN_KEYWORD_PUB	`pub`
TOKEN_KEYWORD_FN	`fn`
TOKEN_IDENTIFIER	`main`
TOKEN_LEFT_PAREN	`(`
TOKEN_RIGHT_PAREN	`)`
TOKEN_KEYWORD_VOID	`void`
TOKEN_LEFT_BRACE	`{`
TOKEN_IDENTIFIER	`ifj.print`
TOKEN_LEFT_PAREN	`(`
TOKEN_STRING_LITERAL	`"Hello, world!\n"`
TOKEN_RIGHT_PAREN	`)`
TOKEN_SEMICOL	`;`
TOKEN_RIGHT_BRACE	`}`

Parser

Syntaktická analýza

`parser.c, parser.h`

- všetky funkcie pre rekurzívny zostup, úvodná funkcia start(), deklarovanie dátového typu tParser

Výzvou pri syntaktickej analýze bolo pre nás najmä zahrnutie prvkov pre sémantickú analýzu vo funkciách pre rekurzívny zostup. Tento spôsob nie je veľmi priehľadný - lepšie riešenie by bolo využitie abstraktného syntaktického stromu, ktorého prechod by zaistil spoľahlivejšiu sémantickú analýzu a následne genrovanie 3AK. Žiaľ kvôli časovej tiesni sme sa rozhodli pre časovo menej náročnejšiu alternatívu za cenu iba čiastočnej funkcionality.

parser.c , preto volá funckie obsiahnuté v codegen.c súbore, ktoré sú priamo zodpovedné za vypisovanie správnych inštrukcií na výstup, ktoré sa môžu predať interpretu.

LL gramatika

```
PROG -> PROLOG FUNCDEFS
PROLOG -> const ifj = @import ( "ifj24.zig" ) ;
FUNCDEFS -> FUNCDEF FUNCDEFS
FUNCDEFS -> eps
FUNCDEF -> pub fn id ( PARAM ) RTYPE { STLIST }
PARAM -> id : TYPE NEXTPARAM
PARAM -> eps
NEXTPARAM -> , PARAM
NEXTPARAM -> eps
TYPE -> NULLABLE SIZE
SIZE -> i32
SIZE -> f64
SIZE -> [ ] u8
NULLABLE -> ?
NULLABLE -> eps
RTYPE -> TYPE
RTYPE -> void
STLIST -> STATEMENT ; STLIST
STLIST -> eps
STATEMENT -> VARDECL
STATEMENT -> IDST
STATEMENT -> IFCOND
STATEMENT -> WHILELOOP
STATEMENT -> RETURNST
VARDECL -> VARTYPE id TYPING ASSIGN
TYPING -> : TYPE
TYPING -> eps
VARTYPE -> const
VARTYPE -> var
IDST -> BUILTIN id IDSTTYPE
IDSTTYPE -> ASSIGN
IDSTTYPE -> FUNCCALL
ASSIGN -> = VALUE
```

```
VALUE -> BUILTIN id FUNCCALL
VALUE -> expr
BUILTIN -> ifj .
BUILTIN -> eps
FUNCCALL -> ( CALLPARAM )
CALLPARAM -> TERM NEXTCALLPARAM
CALLPARAM -> eps
NEXTCALLPARAM -> , CALLPARAM
NEXTCALLPARAM -> eps
TERM -> intliteral
TERM -> floatliteral
TERM -> strliteral
TERM -> null
TERM -> id
IFCOND -> if ( expr ) NONNULL_RESULT { STLIST } else { STLIST }
WHILELOOP -> while ( expr ) NONNULL_RESULT { STLIST }
NONNULL_RESULT -> | id |
NONNULL_RESULT -> eps
RETURNST -> return RETURNVAL
RETURNVAL -> expr
RETURNVAL -> eps
```

LL tabuľka

	const	ifj	=	@import	(	“ifj24.zig“	)	;	pub	fn	id	{	}	:	,	i32	f64	[	]	u8	?	void	var	expr	.	intliteral	floatliteral	strliteral	null	if	else	while	|	return	$
PROG	1																																	
PROLOG	2																																	
FUNCDEFS									3																									4
FUNCDEF									5																									
PARAM							7				6																							
RTYPE																	16	16	16		16	17												
STLIST	18	18									18		19										18							18		18		18	
TYPE																	10	10	10		10													
NEXTPARAM							9								8																			
NULLABLE																	15	15	15			14												
SIZE																	11	12	13															
STATEMENT	20	21									21												20							22		23		24	
VARDECL	25																						25												
IDST		30									30																								
IFCOND																														48					
WHILELOOP																																49			
RETURNST																																		52	
VARTYPE	28																						29												
TYPING			27											26																					
ASSIGN			33																																
BUILTIN		36									37																								
IDSTTYPE			31		32																														
FUNCCALL					38																														
VALUE		34									34													35												
CALLPARAM							40				39															39	39	39	39						
TERM											47															43	44	45	46						
NEXTCALLPARAM							42								41																				
NONNULL_RESULT												51																					50		
RETURNVAL								54																53											

Precedenčná analýza

`psa.c, psa.h`

- hlavná funkcia precedenčnej analýzy parser_expr() a pomocné funkcie loadToken(), get_input() a get_top()

`expr_stack.c, expr_stack.h`

- implementácia ADT zásobník pre tokeny vo výrazoch a funkcii nad ním - exPop(), exPush(), exInit(), exTop(), exDispose(), exEmpty()

`prec_table.c, prec_table.h`

- inicializačné funkcie pre precedenčnú tabuľku a definície pomocných dátových typov

Precedenčná analýza je zodpovedná za spracovávanie logických a aritmetických výrazov. Pri tvorení algoritmu som sa inšpiroval hlavne z prezentácií IFJ o prec. analýze a IAL, kde som našiel potrebné informácie k implementácii ADT zásobníku.

Prvý problém, na ktorý som narazil sa týkal získania potrebnej hodnoty prec. tabuľky v závislosti od symbolu na vstupe a na vrchu zásobníka. Vyriešil som ho enumeráciou jednotlivých logických a aritmetických operátorov ako aj tokenov typu literál a spodok zásobníka. Ku všetkým týmto tokenom pristupujem pri presnom poradí, ktoré je definované v súbore prec_table.h ako pole charakterov, cez ktoré indexujem pri inicializácií tabuľky. Následne som bol schopný pomocou cyklu switch implementovať všetky hodnoty prec. tabulky SHIFT, REDUCTION, EQUAL, ERROR , ktoré sú tiež enumerovane. Funkcia loadToken() získa enum. hodnotu tokenu na vstupe a na vrchole zásobníka, čo znamená, že mám všetko potrebné na získanie správnej hodnoty prec. tabulky. Po získaní tejto hodnoty (v súbore psa.c vo funkcii parse_expr() pod premennou "rule"), program vykoná jednu zo 4 vyššie uvedených akcií, tak ako je uvedené v prezentácii.

Ďalší väčší problém pre mňa bolo rozlíšiť, kedy sa jedna o výraz v zátvorkách napr. po tokene "if" alebo "while" a kedy je výraz pre pridelenie hodnoty do premennej. Môj algoritmus totiž spracováva tokeny dovtedy, kým nenarazí na token, ktorý nie je medzi tokenami, ktoré sú pri výrazoch povolené (tieto typy tokenov sú definované v súbore psa.c ako pole "allowed_tokens"), ale pri podmienke "if" alebo "while" je povinná jedna otváracia a zatváracia guľatá zátvorka, čo neplatí pri priraďovaní do premennej. Algoritmus teda musí v jednom prípade skončiť pri povolenom tokene pravej guľatej zátvorky, ale inak až pri nepovolenom tokene. Vyriešil som do vďaka štruktúre " tParser ", ktorá okrem aktuálneho tokenu obsahuje aj ukazatel na nasledujúci token a lokálnej premennej "end_if", ktorá sa nastaví na hodnotu 1 ak nasledujúci token je buď "{" alebo "|" , teda sa jedná o "if" alebo "while" výraz. V takom prípade sa hlavný cyklus neplánovane preruší a danú zátvorku už nespracuje podľa tabuľky, lebo "end_if" rovné 1 znamená, že táto zátvorka je ukončujúca pre nejakú podmienku. V opačnom prípade sa program správa k zátvorke akoby je súčasťou výrazu. Počet zátvoriek je sledovaný vďaka lokálnej premennej "brackets", ktorá sa inkremetuje pri otvorenej guľatej zátvorke a dekrementuje pri zatvorenej takže ak na konci analýzy je hodnota "brackets" iná od 0, program vráti synt. chybu.

Precedenčná tabuľka

	+	-	*	/	(	)	<	>	<=	>=	==	!=	$	i
+	>	>	<	<	<	>	>	>	>	>	>	>	>	<
-	>	>	<	<	<	>	>	>	>	>	>	>	>	<
*	>	>	>	>	<	>	>	>	>	>	>	>	>	<
/	>	>	>	>	<	>	>	>	>	>	>	>	>	<
(	<	<	<	<	<	=	<	<	<	<	<	<	err	<
)	>	>	>	>	err	>	>	>	>	>	>	>	>	err
<	<	<	<	<	<	>	>	>	>	>	>	>	>	<
>	<	<	<	<	<	>	>	>	>	>	>	>	>	<
<=	<	<	<	<	<	>	>	>	>	>	>	>	>	<
>=	<	<	<	<	<	>	>	>	>	>	>	>	>	<
==	<	<	<	<	<	>	>	>	>	>	>	>	>	<
!=	<	<	<	<	<	>	>	>	>	>	>	>	>	<
$	<	<	<	<	<	err	<	<	<	<	<	<	err	<
i	>	>	>	>	err	>	>	>	>	>	>	>	>	err

Sémantická analýza

Tabuľka symbolov

Tabuľka symbolov sa používa na mapovanie identifikátorov (napr. premenných alebo funkcií) a ich atribútov (názov, hodnota, dátový typ atď.). Umožňuje identifikovať duplicitné definície premenných, konštánt alebo funkcií. Jeho implementácia sa nachádza v súbore symtable.c a jeho definícia v súbore symtable.h. Implementácia zásobníka pre tabuľky sa tiež nachádza v týchto súboroch. Ten mal byť pôvodne vo vlastnom súbore, ale kvôli cyklickým závislostiam som musel ustúpiť a umiestniť zásobník a tabuľku do jedného súboru.

Tabuľka funguje na nasledujúcom princípe. Každá funkcia má svoju vlastnú tabuľku, ktorá je uložená v zásobníku. Globálna tabuľka slúži ako jej dno. Definície premenných a konštánt sú vždy uložené v tabuľke, ktorá je na vrchu zásobníka. Definície funkcií sú však uložené v spodnej časti globálnej tabuľky, pretože funkcie je možné volať skôr, ako sú definované.

S touto časťou bol problém. Keď sa funkcia v programe volá niekde v inej funkcii, ako overíte, že bola alebo dokonca bude definovaná. Tento problém som vyriešil tak, že som ich pri prvom stretnutí (definovaní alebo volaní) pridal do globálnej tabuľky symbolov. Tabuľka prvkov dostala aj nový parameter „defined“. Ten určuje, či je položka definovaná. Pre premenné a konštanty je vždy pravdivý. Pre funkcie bude pravdivý len vtedy, keď sa nájde správna definícia.

Prvý variant tabuľky symbolov bol založený na predavanie tokenov, podľa ktorých sa zisťovalo, či sa definuje premenná alebo funkcia, aký má dátový typ, akú má hodnotu atď. Pri riešení problémov spojených s týmto prístupom ma napadlo urobiť tabuľku oveľa „hlúpejšou“. Koniec koncov, uvedené operácie musí aj tak spracovať parser, ktorý by mohol podľa potreby volať jednotlivé funkcie tabuľky. Namiesto tokenov teda tabuľka dostane sériu už spracovaných hodnôt, ktoré len vloží na správne miesto.