Projektová dokumentácia k implementácii prekladača imperatívneho jazyka IFJ20

Tím 014, varianta II

Kurz *Formální jazyky a překladače*

FIT VUT Brno

9. decembra 2020

Rozšírenia:

BOOLTHEN

FUNEXP

MULTIVAL

Autori:

(**vedúci**) *Tomáš Hladký*, xhladk15 25%

Jakub Bartko, xbartk07 25%

Adam Kostolányi, xkosto04 25%

Jozef Makiš, xmakis00 25%

Obsah

Úvod

Cieľom tejto dokumentácie je popísať postup riešenia projektu pre kurzy IFJ a IAL. Obsahuje zhrnutie rozdelenia práce medzi jednotlivých členov tímu a ich spolupráce, a popis návrhu a implementácie jednotlivých súčastí a ich vzájomných rozhraní.

Výstupom projektu implementácia prekladača imperatívneho jazyka IFJ20 je prekladač jazyka IFJ20 (predstavuje podmnožinu jazyka GO), ktorý prekladá zo zdrojového súboru (na štandardnom vstupe) do cieľového jazyka IFJcode20 (na štandardnom výstupe).

Samotný prekladač je rozdelený na niekoľko funkčných častí, ktoré ďalej využívajú menšie moduly rozširujúce a sprehľadňujúce ich činnosť. Týmito funkčnými časťami sú:

1. **scanner**
2. **parser**
3. **code generator**

Tímová spolupráca

Založenie tímu

Na založení tímu sme sa medzi sebou dohodli ešte pred začiatkom semestra. Poznáme sa osobne už od začiatku štúdia, odkedy sme nadobudli skúsenosti s prácou v skupine v rámci kurzov IVS, či ILG a IDM. Založenie tímu, ani dohoda o konkrétnych členoch, teda pre nás nepredstavovali prekážku.

Prvé tímové osobné stretnutie prebehlo v prvom týždni semestra, kedy sme sa dostali jedine ku diskusií o tom, čo nás bude čakať, nakoľko zadanie projektu ešte nebolo k dispozícií. Prostredníctvom aplikácie Discord sme si založili spoločnú skupinu, ďalej využívanú ako hlavný komunikačný kanál, a GitHub repozitár. Ďalej sme si zvolili vedúceho a postupne si začali vyberať úlohy, za ktoré by sme chceli byť zodpovedný. Následné sme si stanovili ďalší termín hovoru v čase, v ktorom sme očakávali, že už budeme mať aspoň základné znalosti o jednotlivých častiach projektu a ich náplni, a o konkrétnych deadlinoch.

Komunikácia

Keďže sme, po zrušení prezenčnej výuky, už ďalej nemali možnosť pravidelne sa stretávať, zostali sme odkázaní výhradne na neosobnú komunikáciu prostredníctvom chatu a pravidelných hovorov, ktoré boli doplnené o častú komunikáciu najmä ohľadom rozhraní jednotlivých častí, prípadne riešenia nejasností.

Riešená bola implementácia rozhraní medzi jednotlivými časťami, výpomoc s implementáciou a riešením problémov, závady, testovanie, dokumentácia, a mnoho ďalšieho. Intenzita spolupráce taktiež narástla pri integrácií rôznych súčastí a v obdobiach pokusných odovzdaní.

Vzniknuté problémy sme riešili v spoločnom chate prípadne hovorom, ktorého obsah sa vždy zhrnul do spoločného chatu, prípadne začlenil do dokumentácie. Na obsiahlejšie problémy bývalo využívané *Issues* rozhranie GitHubu.

Dohodnuté konvencie

Verzovanie nášho projektu prebiehalo prostredníctvom súkromného repozitára na GitHube. Konvencie ohľadom formátovania a štýlu kódu, spôsob práce s vetvami, pomenovávania a popisu commitov, atď. boli dohodnuté v rámci prvého stretnutia.

Rozdelenie úloh

Jakub Bartko

- **všeobecný parser**
- moduly error, linked-list
- Makefile
- LL gramatika pre všeobecný parser
- vnútorný kód

Tomáš Hladký

- **code generator**
- moduly tabuľka symbolov, codegen stack, escape formát
- vnútorný kód

Adam Kostolányi

- **parser výrazov**
- LL gramatika pre parser výrazov
- modul expression parser stack

Jozef Makiš

- **scanner**
- testy
- DFA pre lexikálnu analýzu
- tokeny

Vývojový cyklus

Časť poznámok o vývojovom cykle tohoto projektu (z hľadiska spolupráce a komunikácie jeho autorov) je uvedená v kapitole tímová spolupráca.

Vývojový cyklus každej z troch hlavných častí nášeho prekladača je možné rozdeliť na tieto kroky:

1. zhromažďovanie a osvojovanie si teoretických podkladov
2. vypracovanie podkladov pre implementáciu (DFA, LL gramatiky, formát inštrukcií a prvkov tabuľky symbolov)
3. implementácia modulov nevyhnutných na implementáciu hlavnej časti samotnej (moduly str, linked-list, symtable, atď.)
4. implementácia kostry hlavnej časti
5. dohodnutie podoby rozhrania s nadväzujúcou časťou
6. doplnenie o funkcionalitu potrebnú na generovanie cieľového výstupu danej časti

7. konsolidácia a integrácia s nasledujúcou hlavnou časťou
8. doplnenie o registrované rozšírenia
9. systémové testovanie

Od kroku 4 taktiež prebiehalo podrobné unit testovanie. Približne od kroku 5 zároveň začínal vývojový cyklus nadväzujúcej hlavnej časti. Parser výrazov býval priebežne integrovaný do parsera všeobecného.

Výnimkou bola implementácia všeobecného parsera a generátora výstupného kódu, ktorých vývojové cykly začali dohodnutím a čiastočnou implementáciu ich rozhrania (zoznam inštrukcií a tabuľka symbolov).

Návrh a implementácia

Lexikálna analýza

Po oboznámení sa s deterministickým koncovým automatom na prednáške sme začali s návrhom DKA pre našu lexikálnu analýzu. Začali sme tým, že sme si určili jednotlivé lexémy, ktoré budeme rozpoznávať pomocou DKA. Každý koncový stav nášho DKA predstavuje jeden typ lexému. Z lexikálnych jednotiek sme si vytvorili tokeny, ktoré okrem typu lexému obsahujú aj jeho atribút v prípade identifikátoru, číselných literalov a reťazového literalu.

Hlavnú časť našej lexikálnej analýzy obsahuje funkcia `get_next_token`. Funkcia prijíma vo svojom parametri štruktúru token, ktorá pozostáva z jeho typu a prípadne aj atribútu. Hlavná časť funkcie je implementovaná formou switch-u, ktorý je vo vnútri nekonečného cyklu. Tento nekonečný cyklus beží do doby, kým nenačíta celú lexému, znak konca súboru alebo lexikálnu chybu. Switch sa riadi jednotlivými stavmi, ktoré môžu nastať počas behu programu. Jednotlivé stavy switchu predstavujú pomocné stavy, prípadne aj koncové stavy v našom DKA. Lexikálna analýza využíva pre ukladanie atribútov dynamické pole, do ktorého sa ukladajú jednotlivé znaky načítané zo štandardného vstupu. Po úspešnom načítaní je výraz prekopírovaný do atribútu tokena. Scanner obsahuje funkciu `get_err_line`**,** pomocou ktorej sa predáva aktuálny riadok v prípade detekovanej chyby prekladačom. Pomocná funkcia `source_file_setup` pomáha nastaviť štandardný vstup pre scanner. V prípade že sme dospeli ku koncovému stavu DKA, alebo ku lexikalnej chybe voláme vo väčšine funkciu `char_clear`. Táto funkcia vráti jeden načítaný znak zo štandardného vstupu, pretože scanner je implementovaný žravým spôsobom. Tento načítaný znak zo štandardného vstupu je na viac, pretože bol načítaný už po detekovaní validneho tokena. K tomu je potrebné uvoľniť inicializovaný modul s dynamickým reťazcom. Definície typov tokenov a stavov nájdete v súbore `scanner.h`. Implementáciu nájdete v súbore `scanner.c`. Jednotlivé tokeny ďalej spracúva parser.

Syntaktická analýza

Všeobecná syntaktická analýza (okrem výrazov) je založená na LL(1) gramatike, a implementovaná metódou rekurzívneho zostupu v súbore `parser.c`. Pri implementácií sme vychádzali z prednášanej látky kurzov IFJ a IAL, a knihy *Compilers: Principles, Techniques, and Tools (informácie z nej využité sa však, ako sa nakoniec ukázalo, zhodujú s prednášanou látkou).*

Jednotlivé neterminály sú implementované ako funkcie, ktoré sa rekurzívne volajú. Terminály sú implementované ako tokeny: s povinným typom, ktorý jednoznačne určuje typ

lexémy (s výnimkou rozlíšenia typov identifikátorov na funkcie a premenné), a dátovej štruktúry `Literal_attr` typu union, využívanej na dáta typu: názov premennej alebo hodnota konštanty. Výstup parsera, ktorý je ďalej spracovávaný *generátorom výstupného kódu,* má formu *zoznamu trojadresných inštrukcií* a *záznamov v tabuľke symbolov*.

Prostredníctvom týchto funkcií je zabezpečené rozlíšenie jednotlivých kontextov programu, ktoré sú ďalej ovládané pomocou pomocných premenných a funkcií. Túto funkcionalitu je možné rozdeliť na niekoľko častí.

Kontrola tokenov

Kontrolu zhody medzi očakávaným a prijatým (zo *scannera*) tokenom je vykonávaná primárne funkciou `match`. Jej základ predstavuje porovnanie očakávaného tokenu, ktorý dostane ako argument, a prijatého tokenu uloženého v globálnej premennej `next` typu `tToken`. Okrem toho rieši akcie ohľadom identifikátorov premenných a funkcií potrebné na sémantickú analýzu. Funkcia `match` je ďalej riadená globálnymi premennými `last_func` a `last_elem`, ktoré slúžia na zaznačenie poslednej funkcie a premennej.
Príklad využitia: func foo (a int) (int) {...} - V prípade definície funkcie vzniká potreba priradenia parametrov a návratových typov definovanej funkcií, a dátových typov parametrom; no tie ešte v bode prijatia identifikátora *foo* nie sú dostupné.

Okrem nich využíva globálne flagy `def`, `eps` a `get_next` typu bool. *Def* slúži na prepínanie medzi definíciou a volaním premennej alebo funkcie; `eps` na možnosť návratu z testu tokenu (využívané na účely výberu nasledujúcej vetvy parsovania - "*otestuj, či si dostal IF. ak nie: (eps) vráť sa, (!eps) error*"); a `get_next` na prepínanie získavania ďalšieho tokenu prostredníctvom `get_next_token`, zahrnutého vo funkcií `match`.

Voľba očakávaného tokenu je implementovaná switchmi.

Kontrola kontextu

Tu patrí niekoľko skupín funkcií, medzi ktoré patria:

- ovládanie kontextu tiel funkcií, podmienok a cyklov
- ovládanie zoznamov premenných
- kontrola typov premenných
- kontrola premenných s nedefinovanými typmi
- kontrola volaní a definícií funkcií
- kontrola volaní príkazu `return`

Popisy skupín, ktoré spadajú primárne pod sémantickú analýzu, nájdete v časti sémantická analýza.

Ovládanie kontextu premenných (ďalej len *scope*), ktorý sa mení medzi telami funkcií, podmienok a cyklov, je implementované formou zásobníka označení scopu typu [*char **]. Ovláda sa prostredníctvom funkcií s predponou `scope_`. Premenné s rovnakým názvom, no v rôznom scope, sú v tabuľke symbolov odlíšené prefixom svojho scopu (napr. premenná *a* vo funkcií *main* bude uložená ako *main-a*, alebo funkcia *foo* zavolaná vo funkcií *main* prvýkrát ako *main-0foo*).

Na hľadanie premenných podľa bezkontextového názvu slúži funkcia `id_find`, ktorá jednotlivé scopy postupne prehľadáva, počnúc scopom zadaným ako argument.

Zoznam premenných slúži na zozbieranie názvov premenných v prípade príkazu definície alebo priradenia, a je implementovaný formou *jednosmerne-viazaného zoznamu* názvov premenných typu [*char **]. Po vyhodnotení operácie definície (:=) alebo priradenia (=) je tento zoznam prevedený na cieľový zoznam *sym_var_item-ov,* čo je nasledované

priradzovaním zoznamu výrazov (spadá pod *parser výrazov*) a spracovania ich typov (spadá pod *sémantickú analýzu*). Prevádzanie je súčasťou funkcie `make_dest`**,** ktorá zohľadňuje rozdiely vo funkcionalite spracovávania zoznamu premenných pre operácie definície a priradenia (najmä na účely implementácie rozšírenia MULTIVAL).

Sémantická analýza

Všeobecná sémantická analýza je implementovaná ako nadstavba základu, ktorý pre ňu položila implementácia syntaktickej analýzy (ktorá ju predchádza z hľadiska funkcionality prekladača, aj štádia projektu, v ktorom bola implementovaná). V základe ju možno rozdeliť na generovanie inštrukcií, prácu s tabuľkou symbolov a kontroly. Jej implementácia je rozdelená medzi moduly linked_list a symtable*,* sémantické kontroly a pomocné funkcie.

Inštrukcie sú generované v bodoch procesu parsovania, v ktorých si to vyžadujú konvencie pre komunikáciu *parsera* a *generátora výstupného kódu.* Detaily ohľadom typov a prvkov jednotlivých trojadresných inštrukcií, rovnako ako ich detailný popis a príklady kontextov ich generovania, nájdete v súbore `codegen.h`. Implementáciu ich spracovania nájdete v súbore `codegen.c`.

Tabuľka symbolov z hľadiska sémantickej analýzy slúži ako úložisko údajov o definíciách a volaniach funkcií, argumentoch a parametroch, premenných, dočasných premenných, a najmä o ich typoch (v zmysle int, float64 alebo string); ktoré sú potrebné na kontroly, ktoré sú v rámci sémantickej analýzy vykonávané. Medzi oblasti týchto kontrol patria:

- definície a volania premenných
- dátové typy premenných (a výrazov)
- premenné s nedefinovaným dátovým typom
- definície a volania funkcií
- volania návratov z funkcií (príkaz *return*)

Réžia **definícií a volanií premenných** je rozdelená medzi ovládanie scopu (funkcie s prefixom `scope_`), záznamy v tabuľke symbolov a globálne premenné (napr. `last_elem`**,** `id_list`). Na ich hľadanie sa využíva funkcia ***`id_find`***, ktorá berie do úvahy ich kontextuálne ID (t. j. názov a scope == kľúč prvku tabuľky symbolov). Na zaznamenanie dokončenia ich definície (napr. na prevenciu (skôr nedefinovaného) *a := a + ...*) sa využíva flag *sym_var_itemu* `is_defined`**.** Funkcionalita je kompatibilná s jazykom IFJ20 a rozšíreniami MULTIVAL a FUNEXP.

Na priradzovanie a kontrolu dátových typov premenných sa primárne využíva funkcia `check_types`, začlenená do procesov parsovania výrazov a priradzovania výrazov premenným a argumentom. Vo všeobecnom parseri je začlenená do funkcií `assign_var_def_types` a `assign_var_move_types`**,** v ktorých sa iteruje cieľovým zoznamom premenných a zdrojovým zoznamom výrazov (prípadne návratmi volania funkcie), vykonáva správa typov a generujú zoznamy inštrukcií na deklaráciu premenných a priradzovanie hodnôt.
(Pokračovanie témy dátových typov premenných nájdete v častiach *premenné s nedefinovaným dátovým typom* a *definície a volania funkcií*).

Zoznam premenných s nedefinovaným dátovým typom je generovaný funkciou `check_types` v prípade, že obidve poskytnuté premenné majú nedefinovaný dátový typ.

Táto situácia nastáva výhradne v prípade, že sa vo výraze vyskytuje volanie funkcie, ktorá ešte nebola definovaná. Funkcia `check_types` je však využívaná aj pri kontrole destinácií a argumentov volaní funkcií (a definícií daných funkcií). Tým pádom sa pri kontrole zoznamu takýchto premenných (po spomínanej kontrole funkcií) ich typy ďalej inicializujú a skontrolujú. (*Pozn. tieto zoznamy pozostávajú z* ukazovateľov na dané premenné *uložené v symtable, tým pádom majú prístup k dátovému typu, ktorý je inicializovaný v rámci inej kontroly*)

Kontrola volaní funkcií je vykonávaná až po prejdení celého parsovaného programu, nakoľko definície funkcií nemusia predchádzať ich volaniam. Na ovládanie tohto procesu sa využívajú funkcie s predponou `func_defs_`, a zahŕňa kontrolu počtov a typov parametrov a argumentov, a návratov a destinácií pre volanie funkcie (výraz alebo zoznam premenných).

Kontrola návratov z funkcií pozostáva z kontroly, či funkcia s neprázdnym zoznamom návratových typov obsahuje príkaz return - pomocou flagu `has_ret`; a kontrolu typu samotného prikazu return. Jedná sa kontrolu dátových typov obdobnú s kontrolou priradzovania do premenných alebo kontrolou volaní funkcií, obohatenú o funkcionalitu zoznamu pomenovaných návratov rozšírenia MULTIVAL.

Spracovanie výrazov pomocou precedenčnej syntaktickej analýzy

Výrazy sa spracovávajú pomocou pravidiel určených precedenčnou tabuľkou. Implementácia využíva dva zásobníky, jeden na operátory (`symbolStack`) a druhý na výrazy, premenné a konštanty (`tokStack`). Precedenčná tabuľka sa ďalej využíva na určenie pravidiel. Index stĺpca reprezentuje najvrchnejší prvok na zásobníku operátorov a index riadka tokeny na vstupe. Po zavolaní hlavnej funkcie `parse_expression` sa začne spracovávať výraz. Index do tabuľky zo vstupného tokenu a tokenu na zásobníku sa získa funkciou `get_index` a na jeho základe sa použije pravidlo z precedenčnej tabuľky.

Pravidlo „S“ je implementované vo funkcii `shift`. Táto funkcia po zavolaní skontroluje, či aktuálny token je symbol alebo prvok operácie. Ak je to symbol, token sa pridá na zásobník operátorov. V opačnom prípade skontroluje, či sa daný prvok už nachádza v tabuľke symbolov ako premenná alebo je potrebné ho pridať. Na záver sa zavolá funkcia `get_next_token`, ktorá zo vstupu načíta ďalší token.

Pravidlo „R“ je implementované vo funkcii `reduce`. Táto funkcia spracováva výraz podľa najvrchnejšieho tokenu na zásobníku operátorov a zredukuje ho na základe daného pravidla. Po zredukovaní výrazu sa zároveň pridávajú inštrukcie na neskoršie spracovanie a generovanie kódu.

Pravidlo „Eq“ je implementované vo funkcii `equal`. Táto funkcia sa volá v prípade, že na vstupe sa nachádza token typu pravá zátvorka - v tom prípade redukuje výraz medzi dvomi zátvorkami. Na záver sa zavolá funkcia `get_next_token`, ktorá načíta ďalší token na vstupe.

Pravidlo „Err“ je implementované vo funkcii `error`. Ak je zavolaná táto funkcia, predpokladá sa, že zadaný výraz obsahuje chybu. Funkcia uvoľní z pamäte použité alokované prostriedky a predá dáta všeobecnému parseru na ďalšie spracovanie. Vyhodnotenie, či sa skutočne jedná o chybový stav, prenechá všeobecnému parseru,

z dôvodu kontextu, ktorý pri samotnom parsovaní výrazov nie je potrebný. (*Pozn. v prípade foo(2+2) sa jedná o ")" navyše, ktorá by v rámci daného výrazu (počnúc prvou 2) bola chybou, no všeobecný parser ju dokáže spracovať ako súčasť volania funkcie*)

Posledné pravidlo „A", resp. pravidlo `accept`**,** slúži na spracovanie koncového stavu: uvoľnenie alokovaných prostriedkov a návrat spracovaného výrazu a posledného (neprijatého) tokenu (napr. EOL za *a := 2+2,* alebo ")" za *b := foo(2+2)*) všeobecnému parseru.

Implementáciu nájdete v súbore `expression.c`.

Generátor výsledného kódu

Generátor výsledného kódu sme začali návrhom tabuľky symbolov (`symtable.h`) a samotnými inštrukciami (`codegen.h`), ktoré sme prispôsobili podľa inštrukčnej sady IFJcode20. Generovanie sa spustí volaním funkcie `codegen`. Najskôr sa vygeneruje hlavička, ktorá je rovnaká pre každý program. V hlavičke sa tiež nachádza deklarácia na jedinú globálnu premennú `dev-null`. Jej funkcionalita je rovnaká ako unixový `/dev/null` a slúži nám na zahodenie premenných s podčiarkovníkom - aby sa mohla premenná uvoľniť z dátového zásobníka. Po vygenerovaní hlavičky sa začne generovať obsah podľa inštrukcií. Inštrukcie sú uložené v `linked_list`. Funkcia `codegen_generate_instruction` prechádza celý list a podľa inštrukcie zavolá konkrétnu funkciu (`call_fun, add_var, ...`). Každá inštrukcia má zároveň definovaný počet a obsah validných elementov. Vyhodnotí sa obsah a vygenerujú sa potrebné IFJcode20 inštrukcie. Deklarácie premenných s rovnakým názvom v rôznych blokoch sme vyriešili deklarovaním na začiatku funkcie s meniacimi sa prefixmi a postfixmi. Vstavané funkcie, ktoré vyžadujú chybovú kontrolu počas behu sú implementované priamo v generátore kódu a pokiaľ je takáto funkcia zavolaná aspoň raz, na konci programu sa pre ňu vytvorí definícia, ktorá je použitá pre všetky jej volania. Pokiaľ niektorá vstavaná funkcia nebola použitá, jej definícia sa negeneruje. Na odlíšenie rôznych návestí pre if a cyklus for sme použili inkrementují postfix, ktorý sa podľa typu priradí návestiu a hodnota postfixu sa zapíše na konkrétny `codegen_stack`. Týmto zaručíme, že sa návestia nezmiešajú (napr. podmienka v podmienke, for v ďalšom for cykly, a pod.).

Algoritmy a dátové štruktúry

Dynamický reťazec

Na načítavanie vopred neznámeho reťazca sme nemohli používať statické pole. Z tohoto dôvodu sme sa rozhodli používať dynamické pole, ktoré nám umožňuje načítavať pole o neznámej dĺžke. Dynamický reťazec prestavuje štruktúra reťazca v súbore `str.h`. Táto štruktúra obsahuje ukazovateľ na pole znakov `char *str`, dĺžku reťazca `int str_length`, ktorá nám pomáha pri alokácii pamäte, a veľkosť momentálnej alokovanej pamäte v premennej `int str_alloc_size`. Dynamický reťazec ďalej obsahuje funkcie ako napr. `str_init`, využívanej pri vytváraní nového reťazca. `str_free` sa používa pri uvoľňovaní alokovanej pamäte dynamického reťazca. Funkciu `str_clear` používame pri čistení reťazca, `str_add_char` slúži na pridanie načítaného znaku zo štandardného vstupu do nášho reťazca. Funkcia `str_copy` kopíruje už načítaný reťazec z dynamického pola do

atribútu tokenu. `str_cmp_cons` porovnáva už načítaný reťazec s konštantou, ktorá môže byť napríklad kľúčové slovo, ktoré chceme odlíšiť od identifikátorov. Tento modul bol implementovaný ako rozšírenie prevzaté zo zjednodušeného interpretu v študijných materiáloch predmetu IFJ.

Tabuľka s rozptýlenými položkami

Na implementáciu tabuľky symbolov sme si vybrali variantu č. 2, v ktorej sa implementácia realizuje pomocou tabuľky s rozptýlenými položkami. Vybrali sme si ju preto, lebo sme s ňou už boli oboznámený z voliteľného predmetu IJC. Vďaka tomu sme mohli použiť časť implementovanú ako súčasť druhého projektu z tohto predmetu ako základ našej tabuľky symbolov.

Tabuľka slúži ako primárne úložisko na premenné, dočasné premenné, konštanty a ich dátové typy; a na definície a volania funkcií. Takéto centralizované uchovávanie dát zabezpečuje jednotné ukladanie dát, prístup k dátam (napr. odkazy na elementy trojadresných inštrukcií) a čistenie v prípade ukončenia behu prekladača v ktoromkoľvek bode, ktorých počet je obohatený o veľké množstvo chybových stavov v rámci procesu syntaktickej a sémantickej analýzy.

Ovládanie tabuľky je zabezpečené funkciami s predponou *`symtable_`*. Na zapisovanie prvkov a prístup k nim sa využívajú funkcie *`symtable_insert`* a *`symtable_find`*.
Graf využívaných dátových štruktúr nájdete v prílohe. Zoznam funkcií na jej ovládanie nájdete v súbore *`symtable.h`*, a jej implementáciu v súbore *`symtable.c`*.

Escape formát

Escape formát modul slúži na konvertovanie načítaného reťazca scannerom na formát, ktorý je vhodný na výpis pre generátor výsledného kódu. Ide o ASCII znaky s kódom 000-032, 035 a 092. Tento modul obsahuje dve funkcie `escape_reformat` a `merge_str`. Funkcia `escape_reformat` prijíma už načítaný reťazec, ktorý mu predáva scanner. Merge_str je volaná z `escape_reformat` a posunie nasledujúce znaky doprava aby sa v reťazci mohla vytvoriť medzera o správnej veľkosti. Reťazec je následne prekonvertovaný do tvaru, kde *xyz* je dekadické číslo konkrétneho znaku a zapísaný do vytvorenej medzery.

Zásobník pre syntaktickú analýzu výrazov

Na spracovanie výrazov pomocou precedenčnej syntaktickej analýzy sme boli nútení implementovať abstraktnú dátovú štruktúru typu zásobník. Tento zásobník využívame na ukladanie tokenov a dát ktoré k nemu patria na ďalšie spracovanie výrazu.

Jeho základné funkcie pozostávajú z funkcií stack_init, ktorá inicializuje a pripraví zásobník na ďalšie používanie, ďalej stack_push ktorá vloží na vrchol zadaného zásobníka zadaný token, stack_pop ktorá odstráni najvrchnejší prvok zo zásobníka, stack_count ktorá vráti počet prvkov v zadanom zásobníku a nakoniec stack_free ktorý vymaže všetky prvky na zásobníku a uvoľní pamäť.

Zoznam inštrukcii

Trojadresné inštrukcie, vygenerované v parseri a spracované v generátore kódu, si tieto dve rozhrania predajú prostredníctvom zoznamu inštrukcií, implementovaného ako jednosmerne viazaný zoznam. Daný zoznam je uložený v globálnej premennej *list***.** Tento zoznam je obsluhovaný pomocou funkcií s predponou *list_***,** a jednotlivé inštrukcie funkciami s predponou *instr_*.

Implementácia vnútorného kódu ako trojadresných inštrukcií bola zvolená z dôvodu vysokej podobnosti jeho formátu s assembly formátom výstupného medzikódu; no s určitou mierou abstrakcie. Táto abstrakcia bola v určitých bodoch (inštrukcia MOVE napr. pracuje so **zoznamom** zdrojových výrazov a cieľových premenných) zavedená, aby sa zamedzilo nadbytočnému obmedzovaniu parsera v syntaktickej a sémantickej analýze.

Implementáciu zoznamu inštrukcií nájdete v súbore *ll.c* a detaily jednotlivých typov inštrukcií v súbore *codegen.h*.

Zásobník pre generátor výsledného kódu

Zásobník pre generátor výsledného kódu (`codegen_stack`) sa používa na ukladanie unikátnych hodnôt pre skoky v podmienených výrazoch a cykloch. Obsahuje dve štruktúry `jmp_label_stack_t` a `jmp_label_stack_top_t`. Štruktúra `jmp_label_stack_t` obsahuje ukazovatele na predchádzajúce a nasledujúce návestie. Štruktúra `jmp_label_stack_top_t`obsahuje ukazovateľ na vrchol zásobníka s návestiami. Zásobník obsahuje inicializačnú funkciu `jmp_label_stack_init`, ktorá inicializuje zásobníky pre podmienené výrazy a cykly. Obsahuje funkciu `jmp_label_stack_push`, ktorá vkladá návestie na zásobník aj s jeho unikátnou hodnotou vygenerovanou v generátore. Ďalej používame funkciu `jmp_label_stack_pop`, ktorá vysunie návestie zo zásobníka pokiaľ nie sme na jeho dne a vráti hodnotu návestia. Funkcia `jmp_label_stack_top` vracia hodnotu vrcholu zásobníka. Na následné vyčistenie zásobníka používame funkciu `jmp_label_stack_clear`. Na uvoľnenie konkrétneho zásobníka poslúži funkcia `jmp_label_stack_free`. Z modulu priamo využívame `jmp_label_stack_free_all`, ktorý sa volá bez parametrov a postará uvoľnenie všetkých zásobníkov.

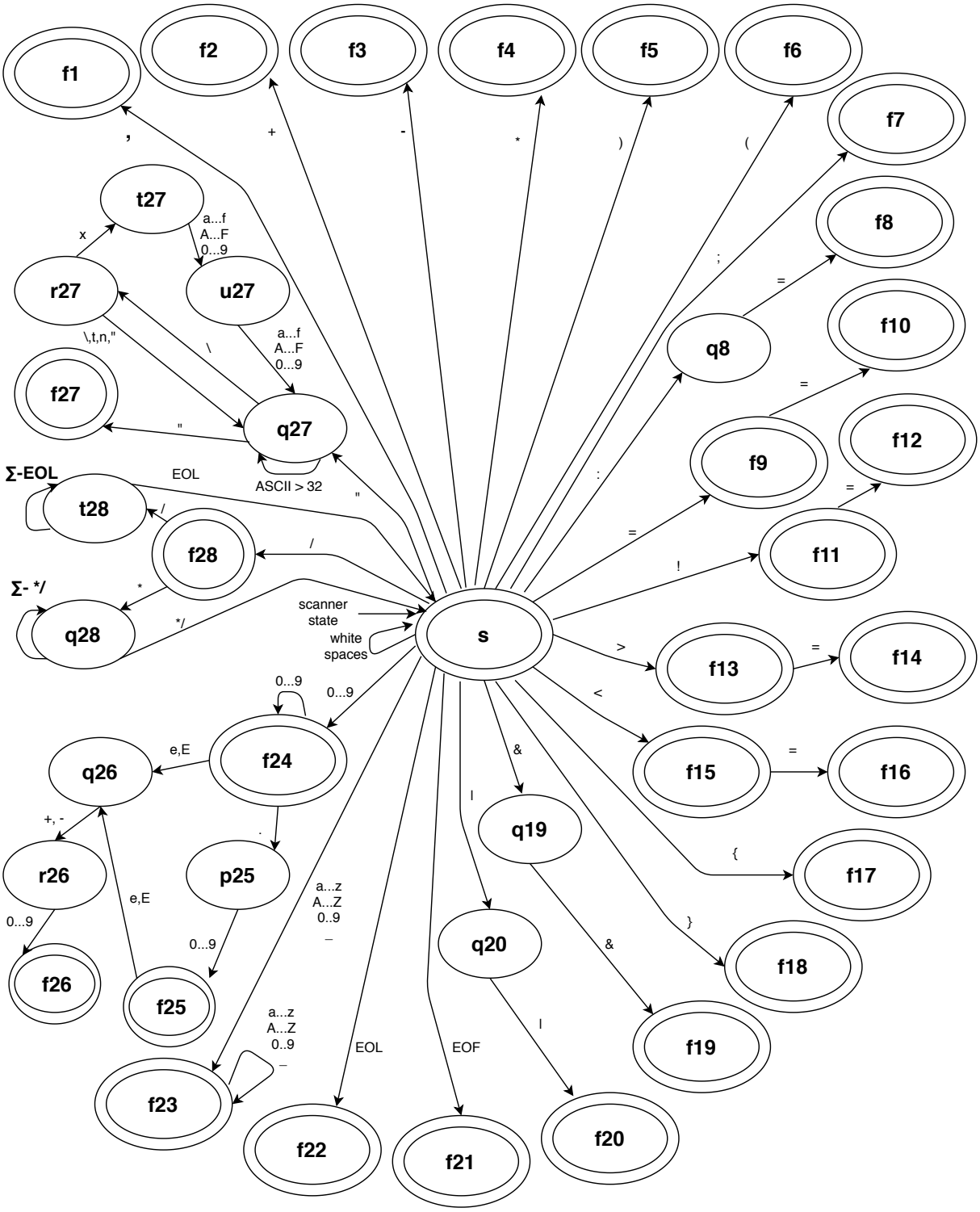
Záver

Projekt z predmetov IFJ a IAL nás motivoval nadobudnúť nové teoretické znalosti a obohatil nás o praktické skúsenosti z oblastí algoritmov, konečných automatov, formálnych jazykov a prekladačov. Vyskúšali sme si implementáciu komplexnejších dátových štruktúr a prácu s nimi na implementácií súčastí prekladača a ich rozhraní. Mali sme príležitosť oboznámiť sa s fungovaním prekladača a súčastí, algoritmov a postupov, na ktorých je postavený. Pri vývoji sme nadobudli množstvo praktických skúseností s prácou v C, využívaním GitHubu, unit aj system testovaním, udržiavaním a verzovaním rozsiahlejšieho projektu, tvorbou projektovej dokumentácie; a v neposlednej rade aj s organizáciou tímu a prácou na spoločnom projekte.

Veríme, že tieto skúsenosti sú hodné množstva času a úsilia, ktoré sme do tohto projektu investovali a dúfame, že sa preukážu ako užitočné pre naše nadväzujúce štúdium aj budúce kariéry.

Prílohy

Príloha 1: Diagram konečného automatu



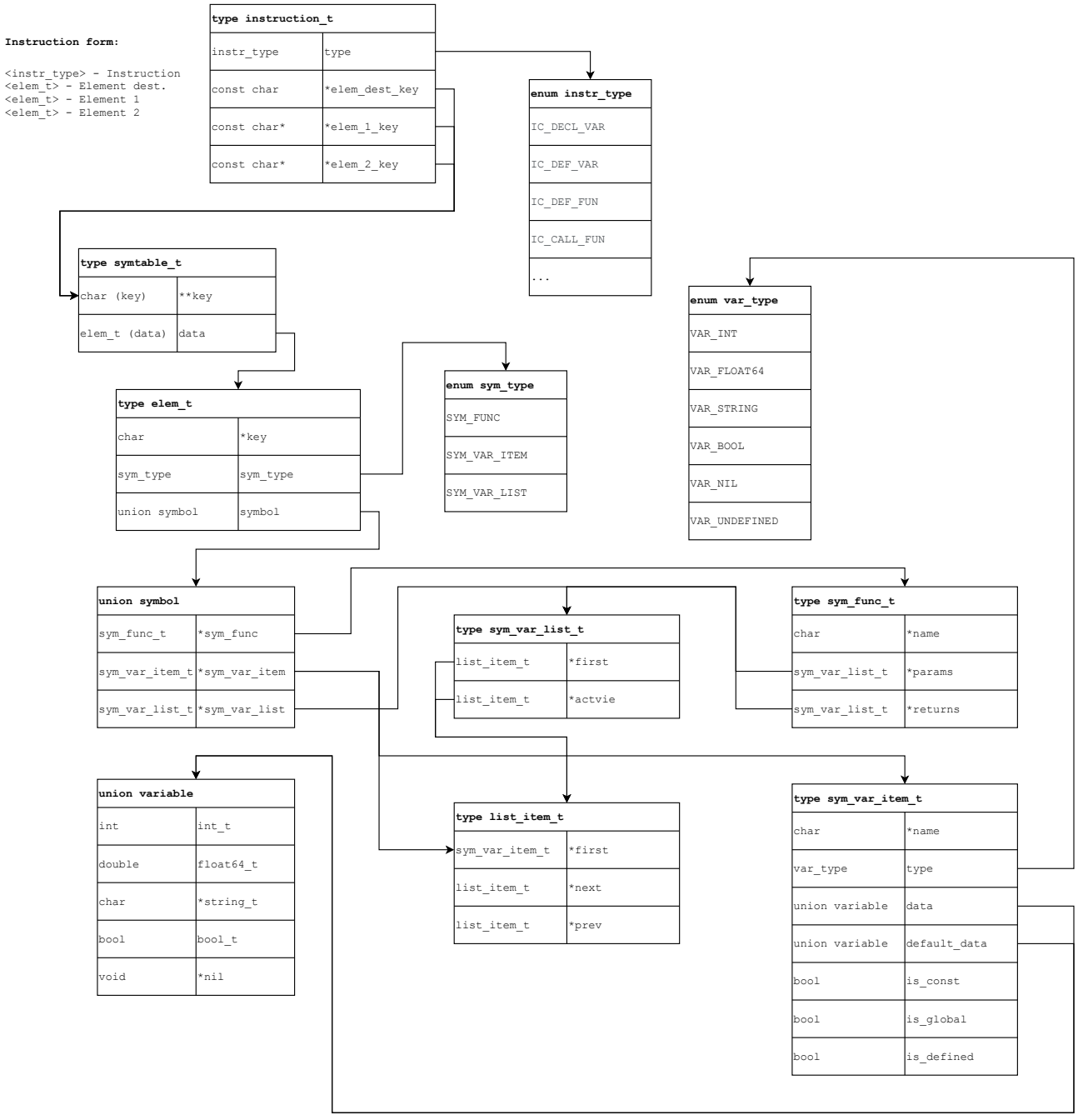
Príloha 2: LL gramatika všeobecného parsera

```
(1)  program → prolog FUNC func_list_pre MAIN '(' ')' '{' EOL body '}' EOL func_list EOF
(2)  prolog → PACKAGE MAIN EOL
(3)  func_list_pre → ε
(4)  func_list_pre → FUNC_ID '(' param_list ')' func_def_type EOL FUNC func_list_pre
(5)  func_list → ε
(6)  func_list → func_def func_list
(7)  func_def → FUNC FUNC_ID '(' param_list ')' func_def_type EOL
(8)  param_list → ε
(9)  param_list → VAR_ID type next_param
(10) next_param → ε
(11) next_param → ',' VAR_ID type next_param
(12) func_def_type → '{' EOL body '}'
(13) func_def_type → '(' func_def_ret
(14) func_def_ret → ')' '{' EOL body '}'
(15) func_def_ret → ret_list_def ')' '{' EOL body '}'
(16) ret_list_def → type next_ret_def
(17) next_ret_def → ε
(18) next_ret_def → ',' type next_ret_def
(19) body → ε
(20) body → command EOL body
(21) command → VAR_ID var_
(22) command → func_call
(23) command → if
(24) command → cycle
(25) command → return
(26) var_ → var_list var_cont
(27) var_cont → ':=' expr_list
(28) var_cont → '=' expr_list
(29) var_list → VAR_ID next_id
(30) next_id → ε
(31) next_id → ',' VAR_ID next_id
(32) if → IF COND '{' EOL body '}' if_cont
(33) if_cont → ε
(34) if_cont → ELSE else_
(35) else_ → '{' EOL body '}'
(36) else_ → if
(37) cycle → FOR for_def ';' COND ';' for_move '{' EOL body '}'
(38) for_def → ε
(39) for_def → var_list ':=' expr_list
(40) for_move → ε
(41) for_move → var_list '=' expr_list
(42) return → RETURN return_list
(43) return_list → ε
(44) return_list → func_call
(45) return_list → EXPR next_ret
(46) next_ret → ε
(47) next_ret → ',' EXPR next_ret
(48) func_call → FUNC_ID '(' func_args ')'
(49) func_args → ε
(50) func_args → func_call
(51) func_args → EXPR next_arg
(52) next_arg → ε
(53) next_arg → ',' EXPR next_arg
(54) expr_list → EXPR next_expr
(55) expr_list → func_call
(56) next_expr → ε
(57) next_expr → ',' EXPR next_expr
(58) type → INT
(59) type → FLOAT64
(60) type → STRING
```

Príloha 3: Precedenčná tabuľka na spracovanie výrazov

		INPUT										
		+, -	*, /	(	)	<, <=, >, >=, !=, ==	i	STRING	&&	||	NEG	$
STACK	+, -	>	<	<	>	>	<	<	X	X	X	>
	*, /	>	>	<	>	>	<	<	X	X	X	>
	(	<	<	<	=	<	<	<	<	<	<	X
	)	>	>	X	>	>	X	X	>	>	>	>
	<, <=, >, >=, !=, ==	<	<	<	>	X	<	<	>	>	<	>
	i	>	>	X	>	>	X	X	<	<	<	>
	STRING	>	X	X	X	>	X	X	X	X	X	>
	&&	X	X	<	>	<	<	X	>	>	<	>
	||	X	X	<	>	<	<	X	<	>	<	>
	NEG	X	X	<	>	<	<	X	>	>	<	>
	$	<	<	<	X	<	<	<	<	<	<	OK

Príloha 4: Model tabuľky symbolov



Literatúra

Prezentácie z predmetov IFJ a IAL.

A. Aho, M. Lam, R. Sethi,J. Ullman. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 2nd edition, 2006. ISBN-13 : 978-0321486813

Holub, A. I. *Compiler design in C*. In: Kernighan, B.W, editor, 2nd edition. New Jersey: Prentice Hall PTR, 1990. ISBN 0-13-155045-4