Dokumentácia k projektu IFJ a IAL

**Implementácia prekladača imperatívneho jazyka IFJ19**

Tím 127, varianta I

**Jiří Žák (xzakji02) 25%**

Ivan Halomi (xhalom00) 25%

Adam Ševčík (xsevci64) 25%

Martin Hiner (xhiner00) 25%

1. Úvod

Cieľom tohto projektu bolo vytvoriť prekladač, ktorý načíta zdrojový kód v IFJ19, čo je zjednodušená podmnožina jazyka Python 3, a prekladá ho do IFJcode19 kód. Program je konzolová aplikácia, ktorá vracia výsledný kód alebo prípadne zodpovedajúci kód chyby. Program načíta zo štandardného vstupu a vráti sa na štandardný výstup.

1. Návrh a implementácia

Jednotlivé modely boli paralelne vyvíjané každým členom tímu, podľa rozdelenia.

* 1. Lexikální analyzátor

Implementácia lexikálneho analyzátora sa nachádza v súbore scanner.c a využíva pomocnú knihovňu string.c pre jednoduchšiu prácu s reťazcami.

Hlavná funkcia lexikálneho analyzátora je get\_token(), ktorá je implementácia konečného automatu podľa grafu vytvoreného na základe zadania. Funkcia vracia ukazateľ na objekt typu tToken, ktorý reprezentuje jednu lexému zo zadaného kódu. Sú v ňom uložené informácie ako riadok, na ktorom sa token nachádza, obsah a typ tokenu, prípadne podtyp, podla požiadaviek ostatných modulov. Jednotlivé stavy automatu sa nachádzajú v type enum tState, ktorý sa ale pre jednoduchosť zároveň využíva ako označenie typov a podtypov tokenov. Funkcia je implementovaná ako nekonečný switch, ktorý znak po znaku prechádza štandardný vstup a podla zadaných pravidiel ho delí na jednotlivé tokeny.

Jazyk IFJ19 využíva odsadenie riadkov na zoskupenie príkazov do sekvencie, preto bolo pre kontrolu nutné využiť pomocný zásobník typu LIFO. Zmena odsadenia sa prejaví odoslaním tokenov sIndent alebo s sDedent, ktoré nahradzujú zložené zátvorky známe z jazyka C.

V súbore sa ešte nachádzajú pomocné funkcie init\_token() na inicializáciu tokenu, unget\_token() na vrátenie tokenu na štandardný vstup a assign\_type() na zistenie, či sa je v tokene uložené kľúčové slovo alebo identifikátor. Taktiež sa tu nachádzajú pomocné funkcie pre prácu so zásobníkom.

* 1. Syntaktický analyzátor

Funkcia čaká na toke a zisťí či je dobrý a keĎ prišiel keď nemal je zle

* 1. Sémantický analyzátor

Kontroluje či čiselko prišlo kedy malo

* 1. Generovanie kódu

// z parseru príde operand/y, podla počtu operandov sa volá funkcia, ktorá sa uloží do dvojsmerne viazaného zotnamu, nakoniec inst\_print, ktorá po jednom vypisuje inštrukcie a vymazáva ic zo zoznamu //

Generovanie kódu je implementované v súbore instruction\_list.c.

* 1. Makefile

Požiadavky projektu zahŕňali súbor Makefile, ktorý prekladá projekt príkazom make. V súbore sú nastavené pravidlá, podla ktorých sa má projekt prekladať. Projekt se prekladá prekladačom gcc s nastavenými flagmi. Zo súborov s príponou .c sa vytvoria objektové súbory s príponou .o a z tých sa potom vytvorí jeden spustiteľný súbor s názvom main.

Tiež sme použili Makefile na testovanie a odstránenie dočasných súborov.

1. Použité špeciálne techniky a algoritmy

Zaviedli sme niekoľko špeciálnych dátových štruktúr pre projekt.

* 1. Tabuľka s použitím binárneho vyhľadávacieho stromu

Vytvorili sme tabuľku s použitím binárneho vyhľadávacieho stromu, ktorý slúži ako tabuľka symbolov, čo súvisí s predmetom IAL, kde sme sa o tom učili. Implementovali sme potrebné funkcie na prácu s touto tabuľkou ako inicializáciu, pridanie novej položky, odstránenie konkrétnej položky, vyhľadávanie a odstránenie celej tabuľky z pamäte.

* 1. ADT String

Pre jednoduchšiu prácu s reťazcami sme implementovali abstraktnú dátovú štruktúru string v zdrojovom súbore string.c. V štruktúre je uložená dĺžka reťazca, alokovaná dĺžka reťazca a ukazateľ na reťazec. Funkcie sprostredkovávajú inicializáciu a uvoľnenie pamäti, pridanie jedného znaku, pridanie reťazca a porovnanie dvoch reťazcov.

* 1. LIFO zásobník

//TODO

* 1. Dvojsmerne viazaný zoznam

S cieľom usporiadaného ukladania inštrukcií, pre neskoršie spracovanie, bola využitá abstraktná dátová štruktúra Dvojsmerne viazaný zoznam, ktorá bola bližšie preberaná v predmete IAL. Tento typ zoznamu nám poskytoval obojsmerný prechod zoznamom a možnosť pristupovania k prvému a poslednému prvku. Štruktúra bola implementovaná priamo v súbore instruction-list.c spoločne s potrebnými funkciami.

1. Tímová práca

Najprv sme rozdelili projekt podľa častí a potom sme postupne na ňom pracovali. Každý člen tímu pracoval samostatne na svojej časti, s prípadnou pomocou druhých členov.

1. Spôsob vývoja
   1. Git

Ako verzovací systém sme použili git a ako vzdialený repozitár službu GitHub. Každý začínal vo svojej vlastnej vetve, kde vyvíjal svoju časť zadania. Neskôr sme všetky časti spojili do hlavnej vetvy, kde sme pokračovali vo vývoji už aspoň čiastočne funkčných modulov.

* 1. Komunikácia

Komunikácia bola väčšinou vykonávaná osobne alebo prostredníctvom služby Messenger. V neskorších etapách prevládala osobná komunikácia za prítomnosti celého tímu.

* 1. Rozdelenie práce

Rozdelili sme prácu rovnomerne s ohľadom na zložitosť. Takže každý člen tímu dostal hodnotenie 25%.

Jiří Žák - syntaktický a sémantický analyzátor, testovanie, dokumentácia

Martin Hiner - lexikálny analyzátor, testovanie, dokumentácia

Ivan Halomi - syntaktický a sémantický analyzátor, testovanie

Adam Ševčík - generovanie kódu, testovanie

1. Záver

Projekt bol pre nás určite zatiaľ najnáročnejší za našu dobu pôsobenia na tejto fakulte, a to nielen z hľadiska implementácie ale tiež komunikácie v tíme a dodržania zadania. Počas vývoja sme sa stretli s mnohými prekážkami, ktoré sme museli riešiť ladením, testovaním alebo konzultáciou s iným tímom. Dokončením tohto projektu sme si nielen utvrdili naše znalosti z predmetov IFJ a IAL ale tiež získali skúsenosti s prácou v tíme.

Obrázok, na ktorom je mapa, text

Automaticky generovaný popis