VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Formálne jazyky a prekladače – Projekt

Prekladač pre jazyk ZIG

Tým xluptas00 varianta TRP-izp

Bodové rozdelenie: 25/25/25/25%
Implementované rozšírenia: žiadne

27. novembra 2024

Vedúci: Samuel Lupták (xluptas00)
Petr Němec (xnemecp00)
Lukáš Houzar (xhouzal00)
Mário Klopan (xklopam00)

1 Návrh

Prekladač sa skladá z 3 hlavných častí a 4 pomocných dátových štruktúr.
Hlavné časti prekladaču (a ich podčasti):

- Lexikálny analyzátor [1]
- Dvojprechodový syntaktický a sématický analyzátor [2]
  - Analýza kódu pomocou rekurzívneho zostupu
  - Analýza výrazov pomocou precedenčnej analýzi
- Generátor výsledného kódu

Pomocné štruktúry použité v prekladači:

- Tabulka s rozptýlenými položkami s implicitným zreťazením položiek [3]
- Abstraktný syntaktický strom [4]
- Zásobník
- Fronta

Využitie jednotlivých štruktúr je nasledovné:

Hašovacia tabulka: Bola použítá pre implementáciu tabulky symbolov. Podmienka pre implicitné zreťazenie nám robila mierny problém, pretože teoreticky nekonečný počet identifikátorov sa nemestí do konečne velkej tabulky

ASS: Slúži na komunikáciu medzi parserom a generátorom kódu

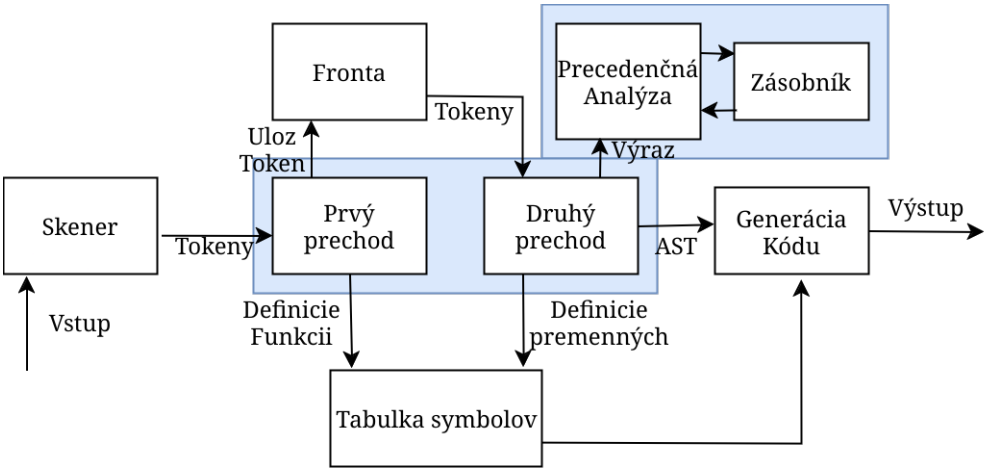
Zásobník: Je využitý precedenčnou analýzou, ktorá ho používa na spracovanie výrazov

Fronta: Má význam pri dvojprechodovej analýze ako uložisko tokenov. Pre dvojprechodovú analýzu sme sa rozhodli po zistení, že definicia funkcie nemusí lexikálne predchádzať jej volaniu.

[1] Ďalej len *skener*
[2] Ďalej len *parser*
[3] Ďalej len *hašovacia tabulka*
[4] Ďalej len *ASS*

2 Popis komunikácie

2.1 Diagram



Obr. 1: Diagram komunikácie

2.2 Stručný popis

Bolo by vhodné začať tým, že parser (modrá časť diagramu) inicializuje všetky operácie v prekladači. Preklad začína vložením programu v jazyku zig na vstup. Skener (Na zavolanie) postupne fragmentuje vstup na jednotlivé lexémi a posiela ich vo forme tokenov do parseru. Ako bolo už spomenuté, tak parser je dvojprechodový. Tokeny idú najprv cez prvý prechod, ktorý kontroluje syntax a sematiku iba pre hlavičky funkcií, ktoré následne ukladá do tabulky symbolov, aby informácie o nich boli dostupné v druhom prechode. Prvý prechod ukladá všetky prečítané tokeny do fronty. Z fronty si tokeny po jednom berie druhý prechod, ktorý kontroluje syntax a sématiku pre ostatok kódu. V prípade že v kóde sa nachádza výraz, zavolá sa precedenčná analýza ktorá tento výraz s pomocov zásobníku spracuje. Druhý prechod zároveň pridáva definované premenné do tabulky symbolov (Samotnú tabulku symbolov však aj využíva, napr. pre kontrolu redefinície). Počas druhého prechodu sa zároveň vytvára ASS ktorý po úspešnom dokončení analýzi slúži ako výstup a zároveň vstup do generátoru kódu. Generátor kódu s pomocou tabulky symbolou generuje cieľový kód.

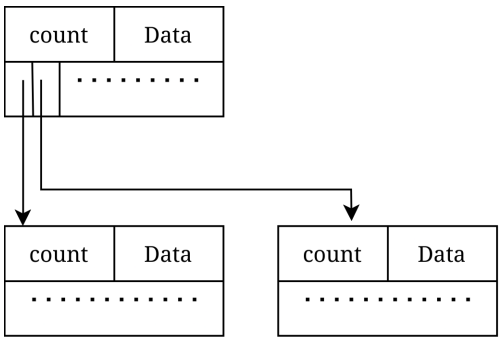
3 Implementácia

Skener: *lexer.*, token.h* (xhouzal00, xnemecp00)
Parser: *syntax.*, queue_fill.*, precedence.** (xluptas00, xklopam00)
Generátor kódu: *code_gen.** (xhouzal00, xnemecp00)
Hašovacia tabulka: *symtable.** (xluptas00)
ASS: *tree.** (xklopam00)
Zásobník: *stack.** (xklopam00)
Fronta: *queue.** (xluptas00)
Ostatné časti: *error.** (xluptas00)

Implementácia dátových štruktúr sa nachádza v jednotlitvých súboroch pomenovaných podla danej štruktúry.

Implementácia častí samotného prekladača spočíva v súboroch skeneru, parseru, generátoru a súboru *error.**, ktorý implementuje základné pracovanie s chybami. Prvý prechod je implementovaný v súboroch *queue_fill.** a druhý prechod je implementovaný v súboroch *syntax.**. *syntax.c* zároveň obsahuje funkciu Main. Prílohy A, B, C, D ukazujú využitú teóriu, ktorá slúžila ako podklad pre jednotlivé časti prekladaču.

3.1 Strom



Obr. 2: Štruktúra stromu

Pre kompletnosť sme sa rozhodli vizualizovať ASS, ktorý sme navrhli pre tento prekladač. Ako je vidno na diagrame, tak každý uzol obsahuje 3 hlavné časti a to sú: *count, data, children*. Kde *count* určuje počet detí ktoré daný uzol má. Hlavná časť, *data*, v sebe uchováva data potrebné na správne generovanie kódu (viac viď. tree.*). Posledná časť *children* je pole ukazovaťelov na deti. V texte ďalej ukážeme ako sa "kódujú"jednotlivé časti kódu do tohto stromu.

Typy uzlov sú špecifikované v *tree.h:15*. Strom má presne 1 koreň typu **ROOT_NODE** a má presne tolko detí, kolko je funkcií vo vstupnom programe. Tieto uzly sú typu **TOP_FUNCTION_NODE** a sú v nich uložené základné informácie o funkciách potrebné pre generáciu kódu. Z jednotlivých uzlov funkcií vychádzajú uzly špecifikujúce jednotlivé časti kódu. Tieto časti kódu sa delia na: *priradenie, definiciu premennej, návrat, vetvenie, cyklus, volanie funkcie*.

Priradenie začína uzlom **ASSIGN_NODE**, jeho prvé dieťa určuje do akej premennej sa priraďuje a jeho druhé dieťa určuje čo sa priraďuje (funkcia alebo výraz).

Definicia premennej má ten istý tvar ako *priradenie* až na to že vrchný uzol má typ **DEFINITION_NODE**

Návrat začina uzlom typu **RETURN_NODE** . Jeho jedniným dieťaťom je strom výrazu, v prípade funkcie nevracajúcej hodnotu nemá deti žiadne.

Vetvenie a *cyklus* začínajú uzlom typu **WHILE_NODE** alebo **IF_NODE**. Na prvom mieste sa na-

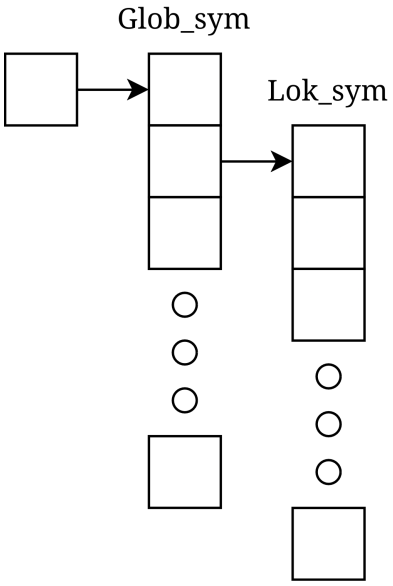
chádza podmineka. Následuje postupnosť častí kódu. Pokiaľ sa jedná o podmineku, začiatok druhej vetvy označuje dieťa typu **ELSE_NODE** následovaná postupnosťou častí kódu.

Volanie funkcie začína uzlom **FUNCTION_NODE** a jeho deti sú postupnosť argumentov. Podobne fungujú aj vstavané funkcie s miernou zmenou typu hlavného uzlu

Zauimavým doplnkom stromu je inklúzia spätného odkazu na otca v každdom uzle. Toto umožňuje celkom elegantné zarovanie a vynorovanie v rekurzívnom zostupe. Pokiaľ je napríklad volaná funkcia, program sa zanorý do uzlu **FUNCTION_NODE** a zaplní ho uzlami argumentov. po jeho spracovaní sa pomocov spätného odkazu program vynorí z tohto uzlu a je pripravený spracovať ďalšie riadky kódu.

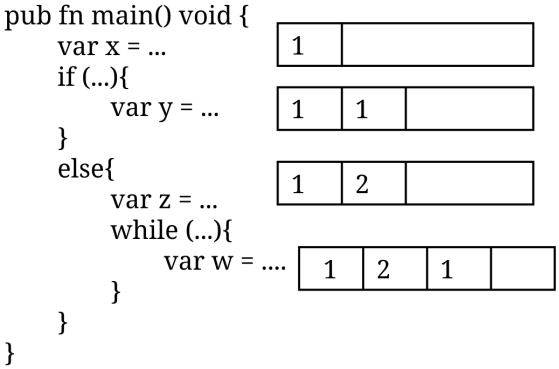
Generácia kódu následne generuje kód jednoduchým prechádzaním tohoto stromu.

3.2 Tabulka symbolov



Obr. 3: Tabulka symbolov

Tabulku symbolov sme implementovali podla zadania ako hašovaciu tabulku. V prekladači existuje ukazateľ na *globálnu tabulku* v ktorej sú záznami definovaných funkcií. Každá funkcia má v sebe odkaz na *lokálnu tabulku* symbolov. Lokálna tabulka symbolov, obsahuje záznami premenných v daných funkciách. Parametre funkcie sú na začiatku tiež interpretované ako lokálné premenné. Zauímavosťou je určite riešenie rozsahu platnosti lokálnych premenných. Namiesto riešenia zásobníku tabuliek symbolov, sme vymysleli jednoduchý systém, kde každá premenná má svoj vlastný zásobník čísiel. Tento zásobník určuje rozsah premennej následovne: Premenné definované v tele hlavnej funkcie majú velkosť zásobníku 1 a na zásobníku je číslica 1, Pokiaľ definujeme premennú v neakom podbloku, tak na zásobník vložíme hodnotu ktorá je jedinečná pre daný podblok (ilustrujeme obrázkom). Tabulka je kvôli požadovanému implicitnému zreťazeniu obmedzená na 1000 položiek, čo si uvedomujeme že nie je najvhodnejšie riešiene, avšak veríme že pre projekt bolo dostačujúce.



Obr. 4: Riešenie rozsahu platnosti

3.3 Implementácia ostatných častí

Ostatné časti (Skener, Parser, Generácia kódu, Zásobník, fronta) sú implementované štandardne metódami prednášanými na FIT VUT, preto ich špecifikáciu vynecháme.

4 Práca v tíme

Na projekte sme pracovali počas celého semestra. Na správu verzií sme využívali platformu GitHub, ktorá nám umožnila prehľadnú organizáciu zdrojového kódu, sledovanie zmien a jednoduché riešenie prípadných konfliktov v kóde. Vďaka tomu mal každý člen tímu vždy prístup k aktuálnej verzii projektu. Na komunikáciu sme používali Discord, ktorý nám poskytol priestor na rýchlu výmenu informácií, plánovanie úloh a riešenie problémov v reálnom čase. Táto platforma bola užitočná najmä pri každodennom zdieľaní poznatkov alebo konzultáciách ohľadom implementácie jednotlivých častí projektu. Okrem online komunikácie sme sa raz týždenne stretávali osobne, aby sme spoločne zhodnotili doterajší pokrok, stanovili priority na nadchádzajúce obdobie a riešili zložitejšie časti projektu, ktoré si vyžadovali detailnú diskusiu. Tento pravidelný rytmus spolupráce zabezpečil plynulý vývoj projektu a pomohol nám efektívne rozdeliť prácu medzi jednotlivých členov tímu. Navyše sme sa rozdelili do tímov po dvoch podľa toho, ako sme ubytovaní na internátoch, čo nám umožnilo ešte rýchlejšiu komunikáciu pri problémoch špecifických pre jednotlivé tímy.

5 PRÍLOHA A - Konečný automat

6 PRÍLOHA B - LL1 gramatika

7 PRÍLOHA C - LL1 tabulka

8 PRÍLOHA D - Precedenčná tabulka