Dokumentace projektu

Formální jazyky a překladače

tým 068, varianta II

Členové týmu:	xlogin:	Bodové rozdělení:
Tomáš Oplatek	xoplat01	100
Daniel Procházka	xproch0d	0
Filip Kružík	xkruzi04	0
Adam Kulla	xkulla01	0

Obsah

Bodové rozdělení

Bodové rozdělení projektu je přesně takové jak je uvedeno na úvodní stránce. Je to bodové rozdělení, které odpovídá tomu kolik kdo udělal práce. Kolik kdo udělal práce bylo určeno dle statistik z gitu. Není to ale rozdělené na procento přesně (to by bylo 92%, 6%, 2%, 1%), protože dávat někomu 6 nebo 1% bodů nedává moc smysl. Zbytek týmu je s tímto srozuměn.

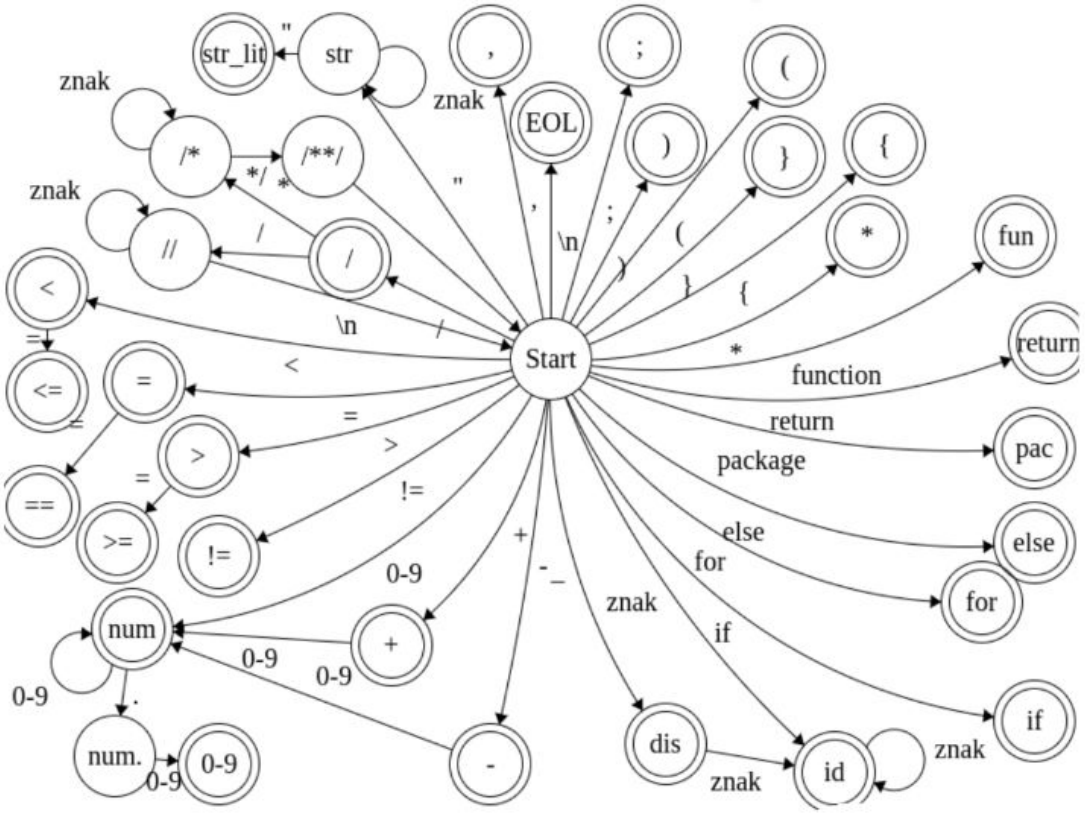
Původní plán byl, aby jsme na projektu pracovali všichni nějak přibližně stejně na všech částech (všichni se naučí vše a kdyby jeden z nás nic nedělal, neznamená to nutně že nefunguje celý projekt), ale v praxi tento postup znamenal, že až na pár drobností je autorem celého kódu jeden, v některých částech dva lidi.

Samotný repozitář pro případnou kontrolu těchto informací bude dostupný zde po termínu odevzdání projektu. V následujícím obrázku je procentuální výpis z programu git-quick-stats, avšak bez email adres, které byly odebrány z důvodu soukromí.

```
Daniel Procházka
 insertions:    205     (6%)
 deletions:     46      (12%)
 files:         12      (9%)
 commits:       5       (19%)
 lines changed: 251     (7%)
 first commit:  Tue Dec 1 19:23:23 2020 +0100
 last commit:   Mon Dec 7 13:21:57 2020 +0100

Adam Kulla
 insertions:    23      (1%)
 deletions:     0       (0%)
 files:         1       (1%)
 commits:       1       (4%)
 lines changed: 23      (1%)
 first commit:  Mon Nov 9 22:15:57 2020 +0100
 last commit:   Mon Nov 9 22:15:57 2020 +0100

Tomáš Oplatek <
 insertions:    3129    (92%)
 deletions:     332     (86%)
 files:         114     (86%)
 commits:       16      (59%)
 lines changed: 3461    (91%)
 first commit:  Mon Nov 9 18:51:48 2020 +0100
 last commit:   Wed Dec 9 20:11:12 2020 +0100

ItIsI-Orient
 insertions:    1       (0%)
 deletions:     1       (0%)
 files:         1       (1%)
 commits:       1       (4%)
 lines changed: 2       (0%)
 first commit:  Tue Dec 1 19:54:26 2020 +0100
 last commit:   Tue Dec 1 19:54:26 2020 +0100

Orient
 insertions:    59      (2%)
 deletions:     8       (2%)
 files:         4       (3%)
 commits:       4       (15%)
 lines changed: 67      (2%)
 first commit:  Mon Nov 9 19:20:42 2020 +0100
 last commit:   Tue Nov 10 14:33:49 2020 +0100
```

1 Lexikální analýza

1.1 Konečený automat



1.2 Implementace

Lexikální analýza je implementována konečným automatem, který čte znak po znaku dokud nenarazí buď na cílový stav, nebo na konec čteného vstupu. Když automat dojde ke konečnému stavu, vrátí strukturu obsahující informace o tomto tokenu (typ, hodnotu atd.) Všechna tato volání jsou ze sémantické analýzy, která si vždy žádá o nový token. Tokeny mohou být i vráceny “zpět”. V praxi to znamená že se token uloží na zásobník tokenů a při další žádosti o token je místo nového tokenu navrácen tento. Tato vlastnost je užitečná v případě kdy syntaktická analýza přečte token navíc a jiná část syntaktické analýzy tento token očekává.
Příklad tohoto může být například kontrola pro klíčové slovo if, kde nastane situace, že kontrola přečte i otevírací složenou závorku, kterou tam ale očekává funkce na kontrolu bloku.

1.3 Soubory

- lex_analyzer.c, lex_analyzer.h - samotná analýza.
- lex_token.c, lex_token.h - definice struktury, která obsahuje informace o samotném tokenu, včetně všech potřebných typů

1.4 Spolupráce

Tohle byla část, kde se ještě členové relativně zapojili. Vytvořené funkce ostatními jsou čtení typů double, int, string včetně odebrání escape sekvencí a odebrání komentářů z projektu

2. Syntaktická analýza

2.1 Návrh

LL tabulka:

Precedenční tabulka:

Precedence	Operátory
1	*,/
2	+,-

Operátory nižším číslem mají vyšší prioritu.

2.2 Implementace

Syntaktická analýza je provedena rekurzivním sestupem. Volání syntaktické analýzy proběhne zavoláním funkce get_syntax_tree, která postupně čte všechny tokeny a rekurzivně vytváří syntaktický strom. Implementovaný strom je n-ární. Jeho kořen je začátek vstupního programu a jeho větve pak tvoří uživatelské definice všech funkcí, které pak již obsahují těla funkcí samotných. Implementace je provedena přes funkci, která kontroluje vždy blok kódu. V této funkci je relativně velký switch, který si volá funkce na kontrolu konkrétních struktur programu, které opět volají zpátky funkci na kontrolu bloku. Největším problémem zde byla kontrola a vytvoření stromu z výrazů. Na to byl zvolen “shunting yard” algoritmus, který pracuje se dvěma zásobníky a precedenční tabulkou. V jednom jsou operátory a v druhém operandy. Operátory a operandy jsou postupně poskládané dohromady až do momentu kdy na zásobníku s operátory je jediná položka, která obsahuje hotový podstrom. Výrazy jsou zpracovány bez operátoru pro porovnání. Ty jsou zpracovány zvlášť a v případě kontroly porovnání (např. v if-u) probíhá zpracování ve tvaru výraz operace výraz.

Syntaktická analýza také hlídá některé části sémantické analýzy. Hlavní věc kterou si hlídá je validita typů při výraz a případné přetypování (z typu float na int a naopak). Druhou věc

kterou hlídá je, jestli jsou ve výrazu použity proměnné které již byly definovány v tabulce symbolů. Tyto kontroly jsou tvořeny tak, že scope proměnných je hlídán na úrovni funkcí, ale už ne na úrovní například ifu, nebo foru. Jinými slovy k proměnným z jiných funkcí není přístup ale pokud jsme již za ifem kde byla definovaná proměnná tak k ní přístup pořád je.

2.3 Soubory

- syntax_analyzer.c syntax_analyzer.h - analýza samotná
- expression_stack.c expression_stack.h - stack potřebný ke kontrole výrazů
- symtable.c symtable.h - tabulka symbolů (detailněji popsaná v sekci 3)
- syntax_tree.c syntax_tree.h - definice struktury syntaktického stromu a obslužných funkcí ke stromu

2.4 Spolupráce

Až na kontrolu porovnání a kontrolu konstrukce for byla celá tato kontrola vytvořena jedním členem týmu, ale od člena který dělal kontrolu foru a porovnání se aspoň projevila snaha dělat víc.

3. Tabulka symbolů

Tabulka symbolů je implementovaná jako tabulka s rozptýlenými položkami (varianta zadání II).
Velikost tabulky je dána na 97, aby nedocházelo k zbytečné alokaci paměti, ale zároveň aby nebyla malá. 97 jsem usoudil že je vhodná velikost vzhledem k tomuto projektu - ani moc velká ani moc malá. Druhý důvod proč bylo číslo 97 zvoleno je protože je to prvočíslo což je třeba vzhledem k funkci využívající operaci modulo, který určuje index prvku uloženého do tabulky.
Při vložení nového prvku do tabulky na index, kde již nějaký prvek je je tento nový prvek vložen před existující.
Jednotlivá položka v tabulce může obsahovat další tabulku symbolů. To je zde z důvodu že na “první” úrovni jsou uloženy pouze funkce, které pak obsahují své vnitřní proměnné.

4. Sémantická analýza

4.1 Implementace

Kontrola probíhá tak, že se jako první do tabulky symbolů přidají předdefinované funkce. Následně se strom, který je výsledkem sémantické analýzy převede do pořadí preorder a poté je pro všechny prvky kde je potřeba kontrola (volání funkcí, return atd.) zkontrolováno, jesti všechny zadané typy opravdu jsou správně, při volání funkce je provedena kontrola jestli opravdu sedí počet argumentů, při funkci s návratovými hodnotami je zkontrolováno že

je return na všech místech (např kdyby funkce končila konstrukcí if a v else by return chyběl) a jestli sedí typy vrácených hodnot. Pro dělení je pouze zkontrolováno jestli probíhá dělení nulou v případě statické hodnoty. V případě že by zde byl výraz složen pouze z konstantních hodnot, tak výraz není vyhodnocen.

4.2 Soubory

- semantic_analyzer.c semantic_analyzer.h - všechny části potřebné k sémantické kontrole. Zde nebyly třeba žádné další věci

5. Generování kódu

5.1 Implementace

Generování kódu je část kterou jsem již nestihl dokončit. Probíhá podobně jaké sémantická kontrola. Ze stromu je vytvořen preorder a postupně se generuje kód podle tohoto pořadí.

Začátek probíhá tak že se definuje hlavička po které proběhne call na label s názvem main, který když doběhne do konce proběhne návrat zpět na začátek a ukončení programu. Mezi hotové části patří načítání hodnoty, avšak již neprobíhá kontrola jestli bylo načítání úspěšné, výpis hodnot, vytváření proměnných, přiřazení do proměnných a zpracování výrazů. Pokud zbyde chvíle volného času (aktuální čas je 21:48 a datum je 9.12.) pokusím se přidat také podmínky a cyklus for. Kdyby tam tyto části nebyly tak jsem je prostě a jednoduše již nestihl implementovat.
Generování výrazů zde probíhá přes stack kde jsou operátory dosazeny ve správném pořadí a následně je na nich operace provedena. Eliminuji tak potřebu vytvářet nějaké zbytečné dočasné proměnné které mohou kolidovat s uživatelem vytvořenými.
Z generování kódu tedy očekávám velmi málo procent, protože opravdu asi neprojde větší množství testů.

5.2 Soubory

- Pouze code_generator.c a code_generator.h kde je celá nekompletní analýza

6. Práce s pamětí

Práce s paměti v projektu je udělaná přes VELMI hloupý garbage collector. Ve zkratce pokaždé když je paměť alokována nebo realokována je upraven nebo přidán ukazatel do seznamu ukazatelů a při ukončení projektu jsou všechny uvolněny.
V případě větších reálných kompilátorů a projektů je tato možnost naprosto nereálná a stupidní, protože využití paměti může být opravdu vysoké, ale zde jsem tuto možnost zvolil protože to je relativně jednoduchý způsob jak se ujistit, že nedochází k úniku paměti.

Zároveň to znamená že nemusím případnou chybu kompilace programu propagovat například v syntkatické analýze o mnoho mnoho volání rekurze, ale můžu program rovnou na místě ukončit volání ukončovací funkce, což značně ulehčí práci také z důvodu že v IDE můžu vložit breakpoint na právě ukončovací funkci a následně v call stacku vidím přesně kde nastala chyba.