**Vysoké učení technické v Brně**

Fakulta informačních technologii

IFJ a IAL

2019/2020

**PROJEKT**

**Tým 018, varianta II**

**Členové týmu:**

Matej Hockicko (xhocki00)

Tomáš Julina (xjulin08) - vedoucí

Tomáš Kantor (xkanto14)

Lukáš Kuchta (xkucht09)

**Rozdělení bodů:**

xhocki00: 25%

xjulin08: 25%

xkanto14: 25%

xkucht09: 25%

**TODO!!! (smazat potom)**

* **LL tabulka do příloh**
* **Popis návrhu: syntax, sémantika, generování**
* **Doplnit použité „speciální“ datové struktury (lehce popsat)**

1. **Rozdělení práce:**

Práci na projektu jsme si rozdělili rovnoměrně (každý 25%), kvůli rozsahu se určil jeden člen týmu na každý modul s tím, že v případě problému jsme jej řešili společně. Konkrétní práce členů na jednotlivých částech jsou uvedený níže:

***Matej Hockicko*** – generování cílového kódu (+ pomocné struktury), běhové kontroly, testování, dokumentace

***Tomáš Julina*** – vedení týmu, lexikální analýza, testování, dokumentace

***Tomáš Kantor*** – syntaktická analýza (+ pomocné struktury), sémantická analýza (+ pomocné struktury), testování, dokumentace

***Lukáš Kuchta*** – sémantická analýza (výrazy), testování, dokumentace

1. **Komunikace:**

Pro komunikaci při vývoji jsme využívali výhradně týmových chat v aplikaci Messenger či časté osobní schůzky.

1. **Verzovací systém**

Při vývoji jsme pro verzování a vzájemnou spolupráci používali verzovací systém *Git* se vzdáleným repozitářem na *Github.com.*

Tento postup nám umožnil pracovat na více modulech současně a v případě chyby v programu jsme se mohli snadno vrátit na předchozí verzi.

**Popis návrhu:**

Řešení projektu jsme si rozdělili do dílčích modulů, které mezi sebou komunikují postupem, který je popsán níže. Projekt jsme začali řešit na začátku semestru, prvním krokem byla lexikální analýza, poté jsme již zbylé moduly na základě předávaných tokenů tvořili simultánně.

1. **Lexikální analýza**

Lexikální analýza (scanner) byl vytvořen na základě návrhu konečného automatu (viz. přílohy), konečný automat je v kódu implementován ve funkci *getToken* s využitím příkazu *switch* a jednotlivým stavům automatu odpovídají *case*.

Scanner dále komunikuje se syntaktickou analýzou prostřednictvím pomocné struktury *TokenPTR*, který představuje předávaný token, tato struktura obsahuje veškeré informace o konkrétním tokenu – jeho hodnotu, typ, alokovaný prostor, atd... Scanner postupně načítá jednotlivé znaky ze vstupu dokud neprojde do koncového stavu, v tomto případě posílá hotový token, v opačném případě dochází k chybě s kódem 1.

Pro zpracování *INDENT* a *DEDENT* (odsazení) byla vytvořena pomocná struktura *iStack* (indent stack), do kterého se při odsazování ukládá aktuální úroveň odsazení a počet mezer v odsazení, zároveň byla použita statická proměnná, která slouží k poznačení si, zda je aktuálně zpracovávaný token prvním na řádku, či nikoliv, tyto informace jsou následně využity pro odsazování a vracení se z odsazení.

Pro ESCAPE SEKVENCE je použita statická proměnná, díky který lze zkontrolovat, zda byl předchozí znak „\“ či nikoliv.

1. **Syntaktická a sémantická analýza**

TODO

1. **Syntaktická a sémantická analýza výrazů**

Pro zpracování výrazů byla využitá precedenční syntaktická analýza zdola horu. Byla vytvořena funkce expression, v který je cyklus, který se opakuje, až není výraz zpracován na konečný výsledek, který je následně vracen parsru. Funkce využívá pomocnou datovou strukturu zásobník TStackToken, na který si ukládá položky struktury TStackTokenItem, kde jsou informace o operandech a operátorech jako například datovej typ a typ tokenu. Funkce volá pomocní funkce shift (slouží na ‘‘pushování‘‘ položek na zásobník) a reduce (slouží na vyhodnocování podvýrazů). Funkce reduce využívá funkce test\_rule (vrací pravidlo pro vyhodnocení podvýrazu) a semantic (vykonává sémantické kontroly operandů a operátorů a taktéž volá funkce na generování cílového kódu).

1. **Generování cílového kódu**

TODO

**Speciální algoritmy a datové struktury:**

1. **Dynamický řetězec**

Pro ukládání obsahu (řetězce) tokenu do jeho struktury bylo zapotřebí implementovat dynamický řetězec, jelikož délka obsahu není předem známá.

Při vytvoření nového nového tokenu se alokuje prostor pro dynamický řetězec o základní velikosti 8, následně se při zápisu každého znaku ve funkci *updateDynamicString* kontroluje, zda při zápisu nebude velikost alokovaného prostoru přesažena, v případě, že ano, alokuje (pomocí *realloc*) se další „chunk“ (velikost alokovaného prostoru zvětšíme o 8).

1. **Dynamický indent stack**

Pro zpracování zanoření se (indent) a následného vynoření (dedent) bylo nutné ukládat si hodnotu aktuálního zanoření a zároveň i hodnoty předchozích zanoření. Proto nebylo nic jednoduššího, než vytvořit pomocnou (dynamickou) datovou strukturu *iStack,* tato obsahuje informaci o aktuální úrovni zanoření, počtu mezer a také odkaz na další položku zásobníku. Zásobník také obsahuje pomocné funkce pro práci s ním initStack (inicializace zásobníku), pushStack (uložení nové hodnoty do zásobníku), popStack (odstranění aktuální hodnoty z vrcholu zásobníku), destroyStack (zrušení celého zásobníku a uvolnění paměti).

1. **Hashovací tabulka**

TODO

1. **Nějaký další zásobník, tabulka? TODO**

TODO

**Přílohy:**

**Diagram konečného automatu specifikující lexikální analyzátor**

****

**LL – gramatika**

Program -> FuncDec Program

Program -> StatList Program

Program -> eof

FuncDec -> def id ( ParamList ) : eol indent StatList dedent

ParamList -> ε

ParamList -> Param ParamList2

ParamList2 -> , Param ParamList2

Param -> Expression

StatList -> Stat StatList

Stat -> id = FuncCall eol

FuncCall -> id ( ParamList )

Stat -> FuncCall eol

Stat -> id = Expression eol

Stat -> Expression eol

Stat -> pass eol

Expression -> id

Expression -> number

Expression -> (

Stat -> if ( Expression ) : eol indent StatList dedent else indent StatList dedent

Stat -> while ( Expression ) : eol indent StatList dedent

**LL – tabulka**

**Precedenční tabulka**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | +- | \*/ | // | r | ( | ) | i | $ |
| +- | > | < | < | > | < | > | < | > |
| \*/ | > | > | > | > | < | > | < | > |
| // | > | < | > | > | < | > | < | > |
| r | < | < | < |  | < | > | < | > |
| ( | < | < | < | < | < | = | < |  |
| ) | > | > | > | > |  | > |  | > |
| i | > | > | > | > |  | > |  | > |
| $ | < | < | < | < | < |  | < |  |

r- relační operátory == != <= < >= >

i- (id, int, double, string)