**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**
* **Precedenční 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 (*stdin*) 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**

**Syntaktická analýza**

Syntaktická analýza je založená na LL gramatice, která je simulována pomocí rekurzivního sestupu. Pomocí funkce *getToken* žádá další token od lexikální analýzy. Pro zajištění, aby se příkaz return nevyskytoval mimo tělo funkce, používáme proměnnou *inFuncDef*. Není snadné rozhodnout, zda se jedná o výraz, přiřazení výrazu, nebo přiřazení volání funkce, v případě tokenu id. Používáme funkci *preloadToken*. Tato funkce nám sdělí, který token by byl další. Pokud bychom použili pouze funkci *getToken*, ztratili bychom token, který potřebuje precedenční analýza.

Pro zpracování výrazů byla využitá precedenční syntaktická analýza zdola nahoru. Byla vytvořena funkce *expression*, v které je cyklus, který se opakuje, dokud výraz není zpracován na konečný výsledek, který je následně vrácen parseru. Funkce využívá pomocnou datovou strukturu zásobník *TStackToken*, na který si ukládá položky struktury *TStackTokenItem*, ve které jsou informace o operandech a operátorech jako například datový typ a typ tokenu. Funkce volá pomocné 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).

**Sémantická analýza**

Informace o proměnných a jejich typech jsou ukládány do tabulky symbolů. Stejně tak informace o funkcích, zda byly definovány, počet parametrů. Tyto informace jsou následně využívány ke definic a redefinic. Taktéž kontrolujeme, zda byly definovány funkce před jejich volání z hlavního těla programu včetně funkcí, na kterých jsou závislé. Dále kontrolujeme práci s globální a zároveň lokální proměnnou stejného jména ve funkci. K tomu slouží proměnná *reviewed* v tabulce symbolů spolu s funkcí *unreviewVariables*.

1. **Generování cílového kódu**

Při tvorbě generátoru cílového kódu bylo potřeba několik pomocných struktur. Hlavní strukturou při generovaní byl seznam instrukcí, do kterého bylo možné vkládat na libovolné místo. Této možnosti jsme využili zejména při vkládaní definic funkcí před hlavní tělo programu, ale taktéž u větvení příkazu, nebo u cyklu while. Dále bylo nutno vytvořit dvě pomocné datové struktury typu stack, z kterých jedna byla použita na ukládaní indexů aktuálně zpracovávaných if-else častí, aby bylo následně možné provést skoky na jednotlivé if-else instrukce, kdežto druhá struktura sloužila na ukládaní pozic v seznamu instrukcí, pomocí kterých generátor vkládal instrukce na správna místa, tedy před, do nebo za tělo while cyklu, nebo také do různých části konstrukce if-else.

Komunikace generátorů s ostatními modulmi probíhala jako volání některé funkce z generátoru, na což generátor přidal určité instrukce do seznamu, nebo vrátil chybu při nezdařené alokaci. Potřebné operandy instrukcí se do generátoru dostávali jako odkazy z tabulky symbolů, z kterých si generátor vytáhl podstatné informace a té vložil do struktury “Instr“, která byla zároveň položkou v seznamu instrukcí.

Pro moduly pod generátorem byli určené zejména funkce na generováni konstrukce if-else a while, jednotlivé funkce pro generovaní do určité části (generate\_if, generate\_else, …) a nebo také funkce pro přímé generování libovolné instrukce. Nesměli také chybět funkce pro generovaní definice funkce, ukončení, voláni funkce a také funkce pro práci s návratovou hodnotou funkce.

Úlohou generátoru bylo také nagenerovat vestavěné funkce. Toto generovaní probíhalo hned na začátku, kdy byla zavolána funkce „generator\_start“. Po jejím zavoláni se vygenerovali vestavěné funkce spolu s několika pomocnými proměnnými, které sloužili pro případné konverze a kontrolu typů za běhu programu.

Kontrola typů probíhala před generováním jednotlivých instrukcí a to tak, aby již nagenerována instrukce obsahovala implicitně konverzované operandy, nebo aby se interpret ukončil s návratovým kódem již před vykonání samotné instrukce.

**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**

1. Program -> FuncDec Program
2. Program -> Stat Program
3. Program -> eof
4. FuncDec -> def id ( ParamList ) : eol indent Stat StatList dedent
5. ParamList -> ε
6. ParamList -> Param ParamList2
7. ParamList2 -> , Param ParamList2
8. ParamList2 -> ε
9. Param -> id
10. Param -> int
11. Param -> double
12. Param -> string
13. StatList -> Stat StatList
14. StatList -> ε
15. Stat -> FuncCall eol
16. Stat -> Expression eol
17. Stat -> id eq Assignment
18. Stat -> pass eol
19. Stat -> if ( Expression ) : eol indent StatList dedent else indent StatList dedent
20. Stat -> while ( Expression ) : eol indent StatList dedent
21. FuncCall -> id ( ParamList )
22. Assignment -> FuncCall eol
23. Assignment -> Expression eol
24. Param -> None

**LL – tabulka**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | ) | ( | id | dedent | while | if | pass | string | double | int | none | , | def | eof |
| <program> |  |  | 2 |  | 2 | 2 | 2 | 2 | 2 | 2 | 2 |  | 1 | 3 |
| <funcDef> |  |  |  |  |  |  |  |  |  |  |  |  | 4 |  |
| <paramList> | 5 |  | 6 |  |  |  |  | 6 | 6 | 6 | 6 |  |  |  |
| <paramList2> | 8 |  |  |  |  |  |  |  |  |  |  | 7 |  |  |
| <param> |  |  | 9 |  |  |  |  | 12 | 11 | 10 | 24 |  |  |  |
| <statList> |  |  | 13 | 14 | 13 | 13 | 13 | 13 | 13 | 13 | 13 |  |  |  |
| <stat> |  | 16 | 15,16,17 |  | 20 | 19 | 18 |  |  |  |  |  |  |  |
| <funcCall> |  |  | 21 |  |  |  |  |  |  |  |  |  |  |  |
| <assignment> |  | 23 | 22,23 |  |  |  |  | 23 | 23 | 23 | 23 |  |  |  |

**Precedenční tabulka**

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

r- relační operátory == != <= < >= >

i- (id, int, double, string)