Propratna dokumentacija za projekat MicroJava compiler

Jelena Pančevski 0123/18

# Postavka projekta

Projekat predstavlja implementaciju programskog prevodioca za programski jezik Mikrojavu. Mikrojava jeste uprošćena verzija programskog jezika Java. Programski prevodilac omogućava prevođenje sintaksno I semantički ispravnog programa napisanog na Mikrojavi u bytecode koji se izvršava na virtuelnoj mašini za Mikrojava programski jezik.

## MicroJava jezik

## Svaki program napisan na MicroJavi počinje ključnom rečju program. Jezik podržava statičke promenljive I konstante, statičke metode I unutrašnje klase. Glavna metoda koja se izvšrava jeste main metoda koja ne prima ulazne argumente I povratna vrednost joj je tipa void. Programski jezik od osnovnih tipova podržava int, bool I char. MicroJava takođe podržava jednodimenzionalne nizove. Kod unutrašnjih klasa postoji nasleđivanje klasa I polimorfizam kao I redefinisanje nasleđenih metoda. Postoje I predeklarisane procedure ord, chr I len. Metoda print ispisuje vrednosti osnovnih tipova podataka (char, int, bool). Takođe je podržano uslovno grananje ( if else strukture) kao I petlje: while I foreach.

Rad programskog prevodioca podeljen je u četiri faze: leksička analiza, sintaksna analiza, semantička analiza I generisanje koda.

## Leksiča analiza

Semantička analiza se bavi prepoznavanjem jezičkih leksema i izdvaja skup tokena koji su prepoznati iz izvornog koda. Leksički analizator realizovan je pisanjem fajla **mjlexer.flex** koji se nalazi u folderu spec. Na osnovu mjlexer.flex fajla, pokretanjem .jar fajla JFlex.jar dobijamo **Yylex.java** fajl koji predstavlja generisani leksički analizator. Komanda za generisanje leksičkog analizatora:

java -jar lib/JFlex.jar –d src\rs\ac\bg\etf\pp1 spec\mjlexer.flex

## Sintaksna analiza

Sintaksna analiza se bavi analiziranjem da li prepoznati tokeni formiraju gramatički ispravne sentence na osnovu gramatike jezika. Za te potrebe napisan je takođe unutar folder-a spec fajl **mjparser.cup** koji pretstavlja LALR(1) gramatiku. Na osnovu mjparser.cup fajla, korišćenjem AST-CUP generatora generiše se sintaksni analizator tj parser. Za generisanje parsera potrebno je koristiti cup\_v10k.jar I symboltable-1-1.jar biblioteke. Nakon sintaksne analize generiše se parser **MJParser.java**.

java –jar lib/cup\_v10k.jar -destdir src/rs/ac/bg/etf/pp1 –ast src.rs.ac.bg.etf.pp1.ast -parser MJParser -dump\_states -buildtree spec/mjparser.cup

## Semantička analiza

Nakon izvršene sintaksne analize generisano je apstraktno sintaksno stablo na osnovu kojeg se formira semantički analizator napisan u fajlu src\rs\ac\bg\etf\pp1\SemanticAnalyer.java. **SemanticAnalyer** klasa predstavlja klasu izvedenu iz VisitorAdaptor klase koja omogućava obilazak čvorova apstraktnog sintaksnog stabla. Obilaženjem čvorova stabla proveravaju se semantička pravila jezika. Semantička analiza se kao I sintaksna pozivaju unutar main metode klase **Compiler**.java koja kao prvi argument komandne linije očekuje naziv ulaznog fajla programa .mj a kao drugi argument fajl u kome će biti smešten .obj fajl generisan u poslednjoj fazi projekta. U main metodi klase Compiler prvo se kreira lekser korišćenjem Yylex klase I prosledjivanjem bafera za čitanje .mj fajla. Zatim se pokreće sintaksna analiza kreiranjem parsera MJParser kome se prosleđuje kreirani lekser objekat I pokreće se metoda parsiranja koja kao rezultat vraća koren sintaksnog stabla. Nakon parsiranja, sledi semantička analiza I kreiranje objekta SemanticAnalyzer koji se koristi kao argument za prolazak kroz generisano stablo I na taj način se izvršava semantička analiza. Tokom semantičke analize popunjava se tabela simbola čiji se sadžaj potom ispisuje. Ukoliko su semantička analiza I sintaksna analiza prošle uspešno, sledi četvrta faza odnosno generisanje koda.

## Generisanje koda

Cilj poslednje faze je generisanje .obj fajla koji će se pokretati korišćenjem Mikrojava virtuelne mašine. Taj fajl sadrži spisak svih instrukcija koje je potrebno izvršiti tokom rada programa. U ovoj fazi se takođe piše klasa koja je izvedena iz VisitorAdaptor klase koja posećuje čvorove sintaksnog stabla. U te svrhe, implementirana je klasa **CodeGenerator**.java, uz korišćenje mj-runtime-1.1.jar odnosno klase Code. Nakon generisanja .obj fajla, sam fajl se potom pokreće korišćenjem disasm I Run alata.

## Novouvedene klase

U fajlu CodeGenerator.java dodate su tri pomoćne klase za obradu petlji kao i if-else truktura sa uslovima.

### JMPAddress

Klasa JMPAddress sadrži dva polja tipa int : address i code. Polje address čuva adresu umetnutog FalseJump-a, dok code predstavlja uslov za skok. Informacija o adresi nam je neophodna onda kada radimo skok na unapred, kada smo saznali adresu na koju treba skočiti i želimo da je dopišemo koristeći metodu Code.fixJump. Informacija o code-u jump-a nam je neophodna u situaciji da je u pitanju uslov koji predstavlja poslednji u nizu uslova koji je razdvojen sa uslovom OR , u tom slučaju ako je taj uslov zadovoljen potrebno je preskočiti preostale uslove i otići na then izvšavanje. Samo u tom slučaju se za jump uvodi uslov isti kao u izvornom kodu a inače se uvodi inverzni uslov. S obzirom da tokom prolaska kroz stablo prvo setujemo uslov za skok a nakon toga saznajemo da li je poslednji uslov razdvojen uslovom OR, tek kasnije mozemo odrediti da li je potreban inverzan uslov ili isti kao u code-u, kako bi prepravili jump potrebno je da sačuvamo i uslov za skok koji čuvamo upravo u polju code.

### IfElseCondition

Klasa IfElseCondition sadrži pet polja tipa int:

* ThenAddress – predstavlja u If-Else grananju adresu početka Then grane, u while i foreach petlji adresu početka tela petlje. Adresa na koju se skače ukoliko su uslovi ispunjeni.
* ElseAddress – predstavlja u If-Else grananju adresu početka Else grane.
* EndIfStmtAddress – predstavlja u If-Else grananju kao i u petljama while i foreach kraj tih struktura tj adresu na sledeći Statement koji se nalazi odmah nakon struktura. Na tu adresu skače se nakon završetka Then grane, a kod foreach i while petlji onda kada uslov za ulazak u petlju nije ispunjen.
* WhileStmtStart – predstavlja adresu uslova while petlje na koju se skače nakon završetka tela petlje.
* ForEachStmtStart – predstavlja adresu foreach petlje u kojoj se učitava sledeći element niza. Uslov foreach petlje se proverava nakon kraja tela foreachpetlje.

Klasa IfElseCondition sadrži i još četiri polja strukturnog tipa:

* ArrayList <Integer> NextConditionAddress – predstavlja niz adresa početka sledećih uslova koji su sa prethodnim uslovom odvojeni OR naredbom.
* Stack <Integer> JMPToEndIfStmt – predstavlja stack sa svim adresama gde je potrebno zakrpiti skok na adresu EndIfStmtAddress.
* Stack <JMPAddress> JMPAddresses – predstavlja stack sa svim adresama u uslovima gde je ubačen falsejump i kojima je potrebno definisati adresu skoka u zavisnosti od pozicije gde se nalaze.
* ArrayList<Stack<JMPAddress>> CondTermList – predstavlja niz stackova u koji su ubačeni svi uslovi koji čine jedan CondTerm tj svi uslovi koji su odvojeni od narednih sa OR naredbom.

### BreakContinue

Klasa BreakContinue sadrži dva polja tipa Stack <Integer> koji se koriste pri dodavanju adresa za skokove:

* BreakStatements – čuva adrese svih break statement-a pronađenih.
* ContinueStatements – čuva adrese svih continue statement-a pronađenih.