**Elektrotehnički fakultet Univerziteta u Beogradu**

**Softversko inzenjerstvo**

**Kompajler za mikrojavu**

Andrej Jokic 2018/0247

**Opis zadatka**

Cilj projektnog zadatka je implementacija kompajlera za programski jezik mikrojava. Komplajler prijavljuje sintaksne i semantičke greške, a u slučaju ispravnog MJ programa generiše MJ kod koji se može izvršavati na MJ virtuelnoj mašini.

Projektni zadatak je podeljen na 4 faze:

1. Implementacija leksičkog analizatora

2. Implementacija sintaksnog analizatora

3. Implementacija semantičkog analizatora

3. Implementacija generatora koda

**Faze**

Projekat je moguće pokrenuti preko komandne linije ili korišćenjem Eclipse IDE u kojem je i pisan. Faze generisanja MJ parsera su sledeće:

**1. Generisanje leksičkog analizatora**

Na osnovu .flex fajla je potrebno izgenerisati klasu Yylex.java koja je zadužena za tokenizaciju ulaznog fajla. Ovo se postiže sledećom komandom:

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

-d [lokacija] predstavlja direktorijum u kojem ce biti generisan Yylex.java fajl generisan od ulaznog flex fajla. Pokretanjem ove komande se dobija sledeći ispis na konzoli:

lexerGen:

[java] Reading "spec\mjlexer.flex"

[java] Constructing NFA : 229 states in NFA

[java] Converting NFA to DFA :

[java] ....................................................................................................................................................................

[java] 142 states before minimization, 131 states in minimized DFA

Leksička analiza se pokreće preko MJParserTest.java fajla, tako da komande za njeno pokretanje nisu od naročitog značaja.

**2. i 3. Faze implementacije sintaksnog i semantičkog analizatora**

Za ove faze se koristi CUP biblioteka. Biblioteku ***cup\_v10k.jar*** je potrebno dodati u classpath projekta u Eclipse okruženju. Koristi se takođe ***symobltable-1.1.jar*** biblioteka koja predstavlja implementaciju tabele simbola.

Prilikom pokretanja iz Eclipse-a potrebno je napraviti novu Run configuraciju, ParserGenerator, i postaviti da je glavna klasa koja se koristi *java\_cup.Main.* Za argument komande linije treba staviti:

**-destdir src\rs\ac\bg\etf\pp1 -parser MJParser spec\mjparser.cup**

Naredbom ***-destdir*** se zadaje direktorijum u koji cup alat treba da smešta rezultate, a

*–* ***parser*** naredbom se zadaje ime izlaznog fajla, i na kraju se zadaje putanja ulaznog fajla koji sadrži specifikaciju gramatike.

Prilikom pokretanja iz komandne linije potrebno je uraditi sledeće:

Rezultat rada cup alata je novi MJParser.java fajl koji sadrži implementaciju parsera. Prilikom pokretanja CUPGenerator konfiguracije dobija se sledeći izlaz:

[java] ------- CUP v0.10k Parser Generation Summary -------

[java] 0 errors and 7 warnings

[java] 52 terminals, 52 non-terminals, and 109 productions declared,

[java] producing 190 unique parse states.

[java] 7 terminals declared but not used.

[java] 0 non-terminals declared but not used.

[java] 0 productions never reduced.

[java] 0 conflicts detected (0 expected).

[java] Code written to "MJParser.java", and "sym.java".

[java] ---------------------------------------------------- (v0.10k)

Zatim je potrebno napraviti novi Run Configuration u Eclipse-u čiji će main biti:

**rs.ac.bg.etf.pp1.Compiler.java**

Za pokretanje ovog main-a potrebni nikakvi parametri, a na izlazu pokrenutog programa se ispisuje tabela simbola kao i eventualne greske prilikom parsiranja MJ koda, odnosno proverava se njegova sintaksna i semantička ispravnost. Za potrebe zahteva projekta može se napraviti i Run Configuration za main.

**4. Faza implementacije generatora koda**

Za ovu fazu je potrebno koristiti ***mj-runtime-1.1.jar*** fajl koji predstavlja MicroJava virtualnu masinu, kao i apstrakciju nad bajtkodom u vidu Code klase.

Pokretanjem Compiler.java fajla, izvršiće se leksička, sintaksna i semantička analiza i ukoliko su sve one prošle bez greške, generisaće se batjkod u .objfajl.

MJ virtuelna mašina treba da pokrene *program.obj* fajl i da ga izvrši.

Potrebno je napraviti novu Run konfiguraciju, VM, gde će za main klasu postaviti:

**rs.etf.pp1.mj.runtime.Run**

kao argument je potrebno navesti: *test/obj/program.obj*

*-debug*je opcija kojom se uključuje debug režim rada.

Radi lakšeg debagovanja je moguće koristiti disasembler koji daje prikaz redom bajtkoda koji se izvršava. Da bi se on koristio treba napraviti još jedan Run Configuration, Disasm, gde će se za glavnu klasu staviti:

**rs.etf.pp1.mj.runtime.disasm**

a za argument putanja do ***obj*** fajla, odnosno: *test/obj/program.obj*.

*Ukratko za Eclipse:*

**LexerGenerator:**

-Main: *JFlex.Main*

-Argumenti: *-d rs\ac\bg\etf\anacompiler spec\analexer.flex*

**ParserGenerator**

-Main: *java\_cup.Main*

-Argumenti: *-destdir src\rs\ac\bg\etf\pp1 -parser MJParser spec\mjparser.cup*

**Compiler:**

-Main: *rs.ac.bg.etf.pp1.MJParserTest test/program.mj test/program.obj*

**VM**

-Main: *rs.etf.pp1.mj.runtime.Run*

-Argumenti: *-debug test\program.obj*

**Disasm**

-Main: *rs.etf.pp1.mj.runtime.disasm*

-Argumenti: *test\program.obj*

**Novouvedene klase**

# TabExtended.java:

Uveli smo klasu **rs.ac.bg.etf.pp1.*TabExtended*** koja proširuje klasu ***Tab*** iz *rs.ac.bg.etf.pp1.symboltable* paketa na takav način da se uvek pri pozivanju njene *init()* metode u tabelu simbola veštački ubacuje i boolean tip podataka koji inače nije bio predefinisan u osnovnoj ***Tab*** klasi.

# DumpSymbolTableVisitorExtended.java:

Uveli smo klasu **rs.ac.bg.etf.pp1.DumpSymbolTableVisitorExtended** koja proširuje klasu **DumpSymbolTableVisitor** iz *rs.ac.bg.etf.pp1.symboltable* da bi se omogucili tekstualni ispis i bool tipa.

# Stack.java:

Uveli smo klasu **rs.ac.bg.etg.pp1.Stack** koja predstavlja obican stek, koji se koristi pri ugnjezdavanju IF..ELSE struktura, Designatora i stvarnih parametara metoda.

# ConditionTree.java:

Uveli smo klasu **rs.ac.bg.etg.pp1.ConditionTree.java** koja predstavlja jedan Condition neterminal, odnosno stablo condition term i condition factor-a.