**PROJEKAT PP1**

**Leksicka analiza**

**Src/sym.java**

Za sve leksicke structure pravimo konstantu kojoj dodeljujemo broj

public static final int LEKSICKE STRUKTURE = 1;

**spec/mjlexer.lex**

sastoji se iz 3 sekcije

* Import
* Direktiva
* Regularni izrazi

Generator na osnovu ove specifikacije genrise java implementaciju analizatora.

Zelimo da nasu impl.smestimo u nas paket – dopunjujemo import sekicju.

Nas analizator treba da bude CUP kompatibilan, treba da implementira cup skener interfejs.

%cup

Zelimo da brojimo linije i kolone:

%line

%column

Mozemo da u generisanu javu klasu ukljucimo nas usluzni kod, neke nase pomocne metode.

Pisemo u sekciji kod

%{

}%

I tu pisemo usluzne metode new\_symbol, koje su iz paketa java\_cup.runtime.Symbol

Nas lekser je kao automat, ima stanje u kom vrsi citanje ulaznih sekv i jedno za obradu komentara. Dodajemo %xstate

Kada lekser dodje do EOF moze da izvrsi neku akciju. Vratiti token EOF kad dodjemo do kraja:

%eofval{

return new\_symbol(sym.EOF);

%eofval}

Sekcija regularni izrazi:

Na osnovu njih omogucimo akcije kada se okine neki od regeksa

Zelimo da lekser ignorise sve vrste belih znakova u kodu

"\t" { }

"\r \n" { }

" " { }

"\b" { }

"\f" { }

Potom dodajemo ostale regularne izraze.

Za sve simbole je ok da bude <YYINITIAL> stanje, i ne mora da se pise.

Za komentar, kada naidje // treba da predjemo u COMMENT stanje, i da se vratimo u YYINITIAL tek kad naidjemo na \r\n

Na kraju svih regeksa, imamo hvatanje simbola koji nam ne predstavljaju nista – nedozvljene simbole:

. { System.err.println("Leksicka greska (" + yytext() + ")" u liniji + (yyline + 1)); }

U lib dodajemo cup.jar i jflex.jar, i dodajemo ih u java build path projekta.

New File -> build.xml -ant fajl koji ce nam generisati sta treba

Pisemo skriptu za bildovanje, doijamo Yylex.java

Imamo targete za delete, lexerGen i compile.

U test cemo dodati program.mj da testiramo.

Pisanje poruka u logove pomocu system.println cemo zameniti log4j Sa apache sajta

U src paket->dodajemo paket util za smestanje usluznih klasa za logovanje. Nju treba da konfigurisemo – config folder- u koji log fajl ispisujemo poruke i u kom format se ispisuje poruka.

Log4j.xml

**Sintaksna analiza**

**Spec-> mjparser.cup.** sastoji se iz 3 dela:

* Import
* Action core sekcija
* Sekcija gde pocinje gramatika sa deklarisanim terminalnim i neterminalnim simbolom.

Cilj da konstruisemo apstraktno sintaksno stablo

Neterminali – veliko pocetno slovo

Terminali – velika slova

::= odvaja levu i desnu stranu smene

Gramatika fajlova cup specifikacije sadrzi mogucnost da svakoj produkciji dodelimo ime koje ce se koristiti kao ime novogenerisane klase – na pocetku desne strane smene – NE SME DA SE IME KLASE ZOVE ISTO KAO I LEVA STRANA

Program ::= (Program) PROG IDENT VarDeclList LBRACE MethodDeclList RBRACE;

Deklarisati sve koriscene terminale i neterminale – iznad

terminal PROG, IDENT, LBRACE, RBRACE, ODENT, LPAREN, RPAREN;

nonterminal Program, VarDeclList, MerhodDeclList, VarDecl, Type, MethodDecl;

nonterminal FormPars, StatementList;

Leva rekurzija za varDeclList

VarDeclList ::= (VarDeclarations) VarDeclList VarDecl

|

(NoVarDeclarations) /\*epsilon\*/

;

VarDecl ::= (VarDecl) Type:varType IDENT:varName SEMI;

MerhodDeclList ::= (MethodDeclarations) MerhodDeclList MerhodDecl

|

(NoMethodDeclarations) /\*epsilon\*/

;

MerhodDecl ::= Type:retType IDENT:methName LPAREN FormPars RPAREN VarDeclList LBRACE StatementList RBRACE;

Sym.java ce biti izgenerisana automatski

Terminalne simbole nazivamo na isti nacin!!

Import sekcija:

Sve sto ce nasa klasa koristiti i od cega ce biti zavisna

Package rs.ac.bg.etf.pp1

Import java\_cup.runtime.\*;

Oporavak od sintakne greske – generator sintaksnih analizatora.

Parser ako naidje na gresku pokusace da se oporavi, ako ne-ispisuje fatal error.

Preklapamo metode koje parser poziva kada prijavljuje sintaksne greske.

Od nase specifikacije nastace fajl sa dve klase-jedna ce biti parser ii mace kod koji obradjuje smnene a druga Action koja izvrsava akcije

Parser kod- sve za parsiranje-utility/user defined kod

Dodajemo sekciju parser code:

Parser code {:

-report\_fatal\_error – kad ne mozemo da nastavimo parsiranje

-syntax\_error-definisemo kako cemo da prijavimo gresku

-unrecovered\_syntax\_error – kad ne moze da se oporavi od greske, samo prijavi gresku i zaustavi parsiranje

-report\_error – poruka za ispis

:}

Treba import sekciju popunit sa log: import org.apache.log4j.\*;

Takodje dopuniti sadrzajem klasa iz paketa koji ce biti generisan prilkom poziva cup generatora i u njemu ce se generisati klase za konstrukciju naseg apstraktnog sintaksnog stable:

Import rs.ac.bg.etf.pp1.ast.\*;

Treba da presretnemo svako dohvatanje tokena koji leksicki analizaror vrati. Nas parser poziva skener/leksicki analziator i trazi sledeci token. Ako ocemo da presretnemo :

Scan with {:

:}

Terminal Integer NUMBER – da izbegnemo konverziju tipova ako znamo da number ima value i int je

Terminal String IDENT

Sada generisemo kod.

Dopunjujemo build.xml

<target name=”parserGen”>

<java jar=”lib/cup\_v10k.jar” fork=”true”>

<Arg value=”-destdir”/>

Ostali arg:

Src/s=rs/ac/bg/etf/pp1

-ast //ime paketa za staticko generisanje klasa neophodnih za gradnju stabla

-src.rs.ac.bg.etf.pp1.ast //tu se smestaju generisane klase

-parser //parser koji bude generisan da se zove MJParser

MJParser

-buildtree //da parser ima mogucnost dinaimckog generisanja stable

Kreira se novi fajl gde ce biti dodate akcije za kreiranje stabla

-dump\_states

Spec/mjparser.cup

GENERISEMO SYM.JAVA KOJU SMO PROSLI PUT MI RUCNO PISALI

Build generise sym.java i mjparser.java

U klasama stoji package src.rs.ac.bg.etf.pp1 i onda se bune klase. Dodajemo u build.xml

<target name=”repackage” depeds=”parserGen”>

<replace dir = “src” value=”rs.ac.bg.etf.pp1.ast” token=”src.rs.ac.bg.etf.pp1.ast” summary=”true”>

Sad testiramo rad parsera: test->MJParserTest.java

RuleVisitor.java-obilazak sinkatsnog stable. Visit metoda za odg.klase smena kojim je potrebna neka obrada. Ostale imaju vec od visitor adapter.

**Oporavak od greske**: ignorisanje svih znakova do pojave “;”

Dodamo symbol error – specijalni predeklarisani terminal

Error SEMI:l

{: parser.report\_error(“Izvrsen oporavak do ; u liniji ” + lleft, null) :}

**Parserski konflikti – vieznacna gramatika**

Konstrukcija IF-ELSE:

Video 2 – 01:25:00

Build.xml -> posle mjparser – arg value=”dump\_state”

Statement ::= mached |unmatched

**Semanticka analiza**

Dopunjujem akcijama za analizu semantike. Da vidimo da li su izrazi i simanticki ispravni.

Tabela simbola jar sa sajta - dodamo u lib i u class path.

Klasa Obj:

Deklarise sva imena koje mozemo imati u kodu.

Interfejs tabele simbola:

Init-sva deklarisana i predefinisana imena i metode.

Int, char, eol, null, i metode char-> int i int->char i len.

Potom imamo open scope i close scope. Inser-pravi novi obj cvor i dodaje u tabelu simbola. Fin-> nadje i vrati cvor koji odgovara prosledjenom parametru. Metoda dump-ispis sadrzaja tabele simbole. Chain local symbols – za ulancavanje metoda i polja.

Obrada deklaracije:

Globalne, lokalne i dekl.fje maina

Detektovti svako naknadno koriscenje imena

Iskaz dodele i return mora da se slaze po tipu.

Struktura nase tabele: Prvo ime-ime programa, objektni cvor.

IDENT je ime programa. Moramo smenu da presecemo-da umetnemo akciju koja stavlja ime u tabelu simbola. Akcije ne dodajemo u specifikaciju, vec koristimo sintaksno stablo i u visit metodama pisemo akcije.

Src-pp1-SemanticPass extends VisitorAdapter

Build.xml-compile-pathelement lib/symboltable.jar

Prekopiramo deo iz ruleVisitor

Uvodimo ProgName ::= (ProgName) IDENT;

Gledamo Tab.java-insert

Tab.insert(Obj.Prog, progName.gerProgName(), Tab.noType())

Potom otvaramo scope – Tab.openScope();