Sveučilište u Zagrebu
Fakultet elektrotehnike i računarstva

Seminarski rad iz predmeta
Prevođenje programskih jezika

Zadatak broj 77

Zagreb, siječanj 2009.

Seminarski rad iz predmeta Prevođenje programskih jezika

Student:

Matični broj studenta:

Zadatak broj 77: Pomoću programa YACC ostvariti sintaksnu analizu programskog koda za jezik ECMAScript. Potrebno je podržati osnovne konstrukte jezika poput ulaznih i izlaznih naredbi, blokove, naredbe grananja, programske petlje. Samostalno definirati ulazni niz (tablicu uniforminih znakova) koji će se prenijeti programskom alatu YACC. Nije potrebno ostvarivati i leksičku analizu.

Uvod

Sintaksna analiza je središnji korak rada jezičnog procesora. Sintaksni analizator provjerava da li niz leksičkih jedinki zadovoljava hijerarhijsku strukturu izvornog programa zadanu sintaksnim pravilima. Hijerarhijska struktura programa se najčešće zadaje primjenom formalne gramatike i opisuje stablom. Sintaksni analizator čita uniforme znakove leksičkih jedinki, grupira ih u sintaksne cjeline, provjerava sintaksna pravila, određuje mjesto sintaksnih pogrešaka i ispisuje ih, izvodi postupak oporavka od pogreške i gradi sintaksno stablo.

YACC

Sintaksnu analizu moguće je dijelom automatizirati pomoću generatora parsera. Generator parsera je program koji iz zadanih produkcija gramatike gradi sintaksni analizator za zadani jezik. YACC (Yet another compiler-compiler) je primjer generatora parsera. Osim za izgradnju sintaksnog analizatora, program YACC se koristi za rješavanje problema koje je moguće svesti na problem prihvaćanja kontekstno neovisnih jezika. Program YACC koristi LALR (LookAhead-LR) postupak gradnje tablice LR parsera. LALR postupak gradi znatno manju tablicu LR parsera od kanonskog LR postupka. Osnovna ideja LALR postupka je grupiranje stanja označenih istim LR(0) stavkama u jedinstveno stanje. Nedostatak ovog postupka je mogućnost nastanka proturječja.

Sintaksni analizator definira se u datoteci sa nastavkom .y koja se predaje programu YACC. Definicija sintaksnog analizatora sastoji se od tri osnovne cjeline:

```
Deklaracije
%%
Pravila Prevođenja
%%
Pomoćni potprogrami
```

Prva cjelina su deklaracije koje se sastoje od dva dijela. Dio ograđen oznakama %{ i %} predviđen je za deklaracije jezičnog procesora C. Drugi dio deklaracija predviđen je za definiranje završnih znakova gramatike. Središnji dio definicije sintaksnog analizatora su pravila prevođenja. Pravila prevođenja čine produkcije gramatike i njima pridružene akcije semantičkog analizatora. U trećem dijelu zadaju se razni pomoćni potprogrami sintaksnog analizatora kao što su postupak oporavka od pogreške i potprogram leksičkog analizatora yylex().

U dijelu deklaracija definiraju se leksičke jedinke u obliku tokena na sljedeći način: %token KON. Ako se završni znak sastoji od samo jednog znaka, kao što su na primjer operatori, tada te znakove nije potrebno posebno definirati. Pod deklaracije se mogu zadati naredbe koje služe za razrješavanje proturječja. Na primjer, naredba %left '+' definira da je operator + lijevo asocijativan. Redoslijed pisanja naredbi određuje prioritet završnih znakova. Ako je završni znak zadan prije nekog drugog, tada je njegova prednost manja. Ako su zadani u istoj deklaraciji to znači da su znakovi jednake prednosti. Ako prednost produkcije nije moguće pravilno odrediti na temelju krajnjeg desnog završnog znaka, onda YACC dozvoljava izravno definiranje prednosti sa naredbom %prec <ZavršniZnak>. Prednost produkcije tada je jednaka prednosti završnog znaka koje je definiran u dijelu deklaracija. Pod deklaracije spada i naredba %start <PočetniZnak> kojom se definira početni nezavršni znak gramatike.

Pravilo prevođenja sadrži produkciju gramatike i semantičku akciju koja se izvršava nakon primjene produkcije. Završni znakovi koji se sastoje od samo jednog simbola pišu se u jednostrukim navodnicima, na primjer '+'. Produkcija gramatike:

<izraz> → <produkcija1> | <produkcija2>

u programu YACC zapisuje se u sljedećem obliku:

```
izraz    : produkcija1 {SemantičkaAkcija1}
         | produkcija2 {SemantičkaAkcija2} ;
```

ECMAScript

Cilj ovog projekta bio je napraviti sintaksnu analizu za jezik ECMAScript. ECMAScript je skriptni jezik kojeg je standardizirao Ecma International u ECMA-262 specifikaciji. Jezik uključuje strukturna, dinamička, funkcijska i objektno orijentirana obilježja. ECMAScript podržavaju brojne aplikacije, a posebno web pretraživači kod kojih se koristi dijalekt JavaScript. Dijalekti uključuju svoje vlastite standardne biblioteke od kojih su neke posebno standardizirane kao što je na primjer DOM kojeg je standardizirao W3C. Aplikacije koje su pisane u različitim ECMAScript dijalektima vjerojatno neće raditi zajedno osim ako nisu dizajnirane da budu kompatibilne. Uz JavaScript najpoznatiji ECMAScript dijalekt je JScript. Cilj oba dijalekta je kompatibilnost sa ECMAScriptom ali uz to pružaju još i dodatna proširenja kojih nema u ECMA specifikaciji.

JavaScript je vrlo popularan klijentski skriptni jezik zbog jednostavnih pravila i procedura čime omogućava brzo učenje i lako korištenje. Glavna prednost JavaScripta je što može reagirati na različite događaje koji se odvijaju na web stranicama te tako omogućuje dizajn dinamičkih i interaktivnih stranica. Druga bitna karakteristika je da se izvodi u klijentskom web pretraživaču prije slanja podataka na server čime se ubrzava rad te smanjuje opterećenje servera. JavaScript omogućava HTML dizajnerima da efikasno i interaktivno dizajniraju web stranice na jednostavan i učinkovit način.

Primjer korištenja JavaScripta:

```
<script type="text/javascript">

//Write a "Morning" message if
//the time is less than 10
var d=new Date();
var time=d.getHours();

if (time<10)
{
document.write("<b>Morning</b>");
}

</script>
```

Ostvarenje

Sintaksna analiza pomoću generatora parsera YACC izvedena je korištenjem alata GPPG (Gardens Point Parser Generator). GPPG je generator za LALR (1) parsere koji prihvaća ulaz u YACC formatu i generira izlaznu datoteku u programskom jeziku C#. GPPG parseri su dizajnirani za korištenje sa leksičkim analizatorom GPLEX (Gardens Point Scanner Generator), ali se također mogu koristiti uz ručno pisane analizatore ili analizatore napravljene sa drugim alatima. GPPG i parseri generirani njime implementirani su isključivo u C# te koriste generičke tipove podataka definirane u C# 2.0. Parser implementira parsiranje od dna prema vrhu te koristi tehniku parsiranja Pomakni-Reduciraj. LR parser gradi se LALR postupkom (Look Ahead LR) koji je programski manje zahtjevan od kanonskog LR postupka, obuhvaća manji skup jezika od kanonskog LR postupka, ali ga je moguće primijeniti na veći skup jezika od SLR postupka. Rad parsera oslanja na komponentu "ShiftReduceParser.dll" čija glavna klasa se instancira sa dva parametra koji određuju tip i koji su definirani u specifikaciji gramatike. Pozivom naredbe "gppg.exe [opcije] ulazna_datoteka.y > izlazna_datoteka.cs" kreira se C# datoteka (.cs) koja predstavlja generirani parser. Generirana datoteka ubačena je u Visual Studio projekt te je povezana sa analizatorom LEX i grafičkim sućeljem.

Analizator generiran GPLEX-om i parser generiran GPPG-om povezani su zajedno preko forme koja se pokreće u datoteci Program.cs. Tamo se nalazi i metoda Ispisi koja služi za ispis rezultata nakon završene analize. U prozoru Izlaz ispisuje se greška u slučaju unosa nepoznate leksičke jedinke te sintaksne pogreške. Za ispis sintaksnih pogrešaka koristi se metoda yyerror koja je ugrađena u GPPG i koja vraća informaciju o nastaloj pogrešci. Leksička analiza izvedena je pomoću alata GPLEX kojim je kreirana C# datoteka koja predstavlja leksički analizator. Kao ulaz u program korišteno je grafičko sučelje napravljeno u C#-u u koje se unosi kod za analizu ili se učitava već gotova datoteka sa željenim kodom. Učitani kod prvo prolazi kroz leksičku analizu tijekom koje se leksičke jedinke pohranjuju u liste. Liste predstavljaju tablicu uniformnih znakova, te KROS tablicu, tablicu identifikatora i tablicu konstanti. U grafičkom sučelju prikazuje se sadržaj svih tablica. Sintaksna analiza koristi podatke iz tablica tako što nadjačava metodu yylex preko koje se dobavljaju tokeni za analizu. Glavna metoda sintaksnog analizatora Analiziraj() kreira instancu parsera te preko nje poziva metodu Parsiraj() preko koje se pokreće parsiranje. Parsiranje se izvodi pozivom metode Parse() koja se nasljeđuje od apstraktne klase ShiftReduceParser. U generiranoj cs datoteci kreira se enumeracija Tokens koja pridružuje brojevnu vrijednost svim zadanim tokenima, te klasa Parser: ShiftReduceParser koja predstavlja definiciju klase koja implementira parser.

Gramatika

Glavni dio konstrukcije sintaksnog analizatora bio je izgradnja gramatike za jezik ECMAScript. Gramatika konstruirana u ovom projektu prihvaća osnovne konstrukte jezika ECMAScript, te ne predstavlja potpunu gramatiku definiranu u ECMA-262 specifikaciji. U nastavku su opisane osnovna obilježja jezika implementirana u ovoj gramatici. Osnovni tipovi podataka su identifikatori i konstante koje mogu biti cijeli broj, decimalni broj ili string koji se sastoji od slova bez razmaka unutar jednostrukih navodnika. Koriste se sljedeće ključne riječi koje su definirane kao tokeni: while, do, if, else, for, switch, case, return, continue, break, default, var, new, this, function. Omogućeni su aritmetički operatori "+","-","*","/", logički operatori "&&", "||", "!" te relacijski operatori "==", "!=", "<"," >", "<=", ">=". Glavni elementi programa su naredbe i definicija funkcije. Naredba može biti blok u vitičastim zagradama, deklaracija varijable, pridruživanje, grananje if koje se može koristiti

sa ili bez nastavka else, petlje while, do while i for, switch case, continue, break i return. Funkcije se definiraju sa ključnom riječi function nakon koje slijede ime funkcije i argumenti. Izrazi su odvojeni sa znakom ';' te mogu sadržavati pridruživanje svih kombinacija aritmetičko logičkih izraza. U gramatici je implementirano korištenje naredbi this i new i izraza unutar uglatih zagrada. Slijedi popis svih produkcija konstruirane gramatike koja se nalazi u datoteci parser.y:

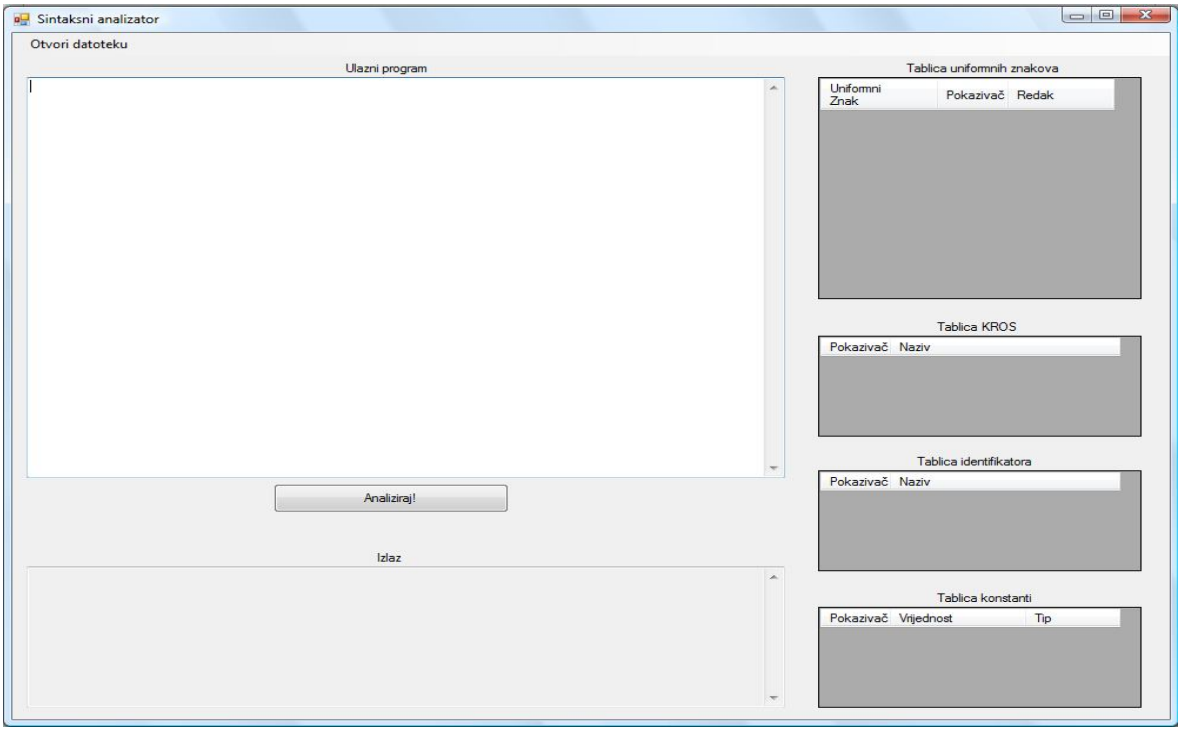
```
program  :  osnovni_elementi  ;
osnovni_elementi :      osnovni_element
                  |     osnovni_elementi osnovni_element  ;
osnovni_element  :      naredba
                  |     funkcija  ;
naredba  :  '{' '}'
          |  '{' naredbe '}'
          |  VAR deklaracija_var ';'
          |  ';'
          |  pridruzivanje ';'
          |  IF '(' izraz ')' naredba ELSE naredba
          |  IF '(' izraz ')' naredba %prec IFX
          |  DO naredba WHILE '(' izraz ')'
          |  WHILE '(' izraz ')' naredba
          |  FOR '(' izraz ';' izraz ';' izraz ')' naredba
          |  FOR '(' VAR deklaracija_var ';' izraz ';' izraz ')' naredba
          |  CONTINUE IDENT ';'
          |  CONTINUE ';'
          |  BREAK IDENT ';'
          |  BREAK ';'
          |  RETURN izraz ';'
          |  SWITCH '(' izraz ')' '{' case '}'  ;
naredbe  :  naredba
          |  naredbe naredba  ;
izraz    :  lijeva_strana '=' or
         |  or     ;
izraz_zagrade:  izraz
              |  izraz_zagrade ',' izraz
              |  ;
pridruzivanje  :  lijeva_strana '=' or
               |  lijeva_strana   ;
or             :  and
               |  or OR and    ;
and            :  jednakost
               |  and AND jednakost ;
jednakost      :  usporedbe
               |  jednakost EQ usporedbe
               |  jednakost NEQ usporedbe ;
usporedbe      :  arit_nize
               |  usporedbe '<' arit_nize
               |  usporedbe '>' arit_nize
               |  usporedbe LE arit_nize
               |  usporedbe GE arit_nize   ;
arit_nize      :  arit_vise
               |  arit_nize '+' arit_vise
               |  arit_nize '-' arit_vise          ;
arit_vise      :  unarni
               |  unarni '*' arit_vise
               |  unarni '/' arit_vise    ;
```

```
unarni          : lijeva_strana
                | '+' unarni
                | '-' unarni
                | '!' unarni      ;
lijeva_strana           :       NEW poziv
                        |       poziv   ;
osnovni_izraz           :       THIS
                        |       IDENT
                        |       KONST
                        |       '(' izraz ')'
                        |       funkcija_izraz ;
poziv   :       osnovni_izraz
        |       poziv argumenti
        |        poziv '[' izraz_zagrade ']'
        |       poziv '.' IDENT     ;
argumenti       :       '(' ')'
                |       '(' argumenti_lista ')'   ;
argumenti_lista         :       izraz
                        |       argumenti_lista ',' izraz   ;
funkcija        :       FUNCTION IDENT '(' parametri_fun ')' '{' tijelo_fun '}'
                |       FUNCTION IDENT '(' ')' '{' tijelo_fun '}'         ;
funkcija_izraz          :       FUNCTION '(' parametri_fun ')' '{' tijelo_fun '}'
                        |   FUNCTION '(' ')' '{' tijelo_fun '}'     ;
parametri_fun           :       IDENT
                        |       parametri_fun ',' IDENT        ;
tijelo_fun      :       osnovni_elementi
                |       ;
deklaracija_var         :       varijabla
                        |       deklaracija_var ',' varijabla    ;
varijabla       :       IDENT
                |       IDENT '=' or    ;
case    :       case_izrazi
        |       case_izrazi default_izraz
        |       case_izrazi default_izraz case_izrazi
        |       default_izraz case_izrazi
        |       default_izraz ;
case_izrazi     :       case_izraz
                |       case_izrazi case_izraz          ;
case_izraz      :   CASE IDENT ':' naredbe
                |   CASE IDENT ':'
                |   CASE KONST ':' naredbe
                |   CASE KONST ':'    ;
default_izraz   :       DEFAULT ':'
                |       DEFAULT ':' naredbe ;
```

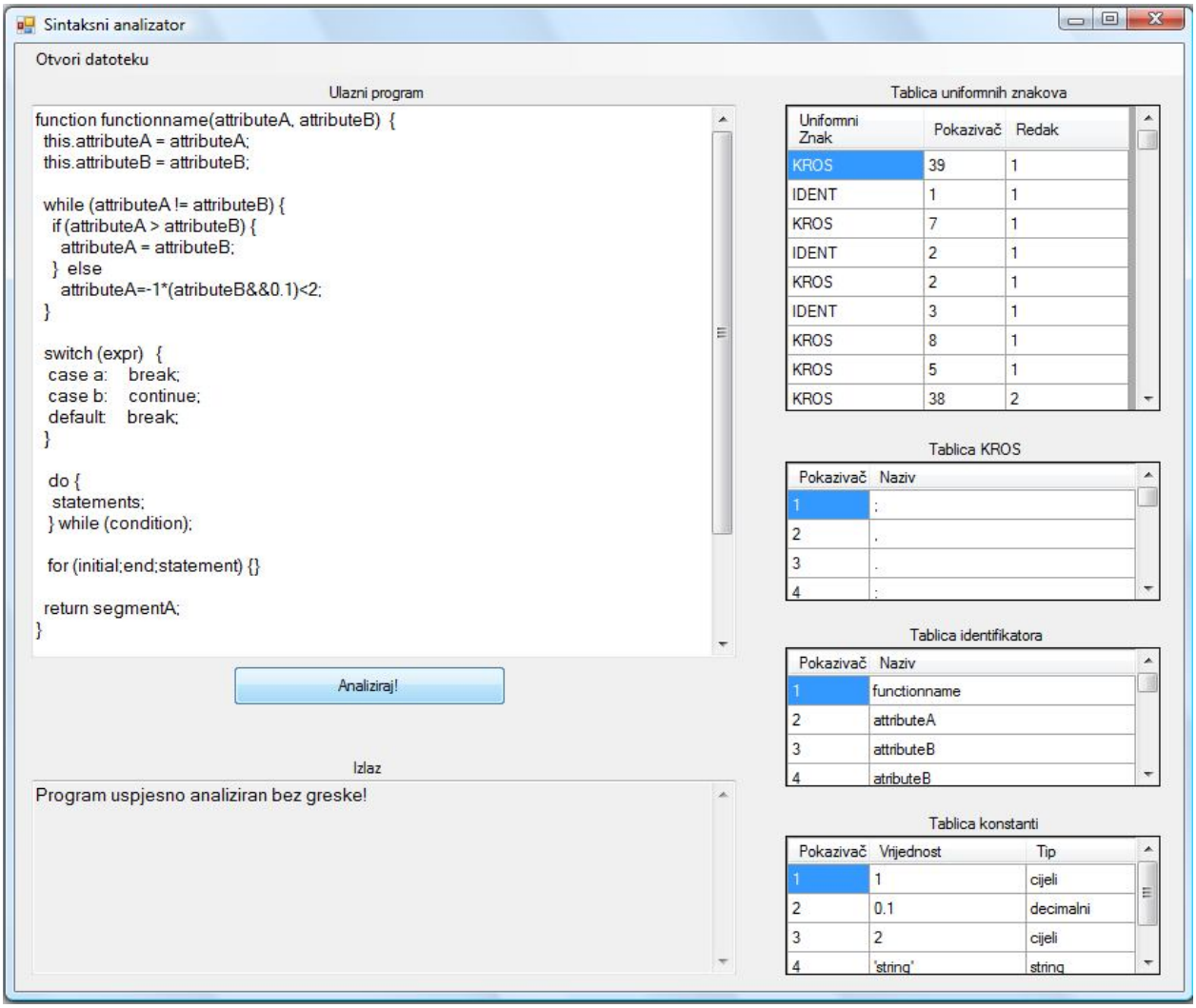
U analizatoru nije implementirana puna podrška za stringove nego je jedini oblik stringa koji se prihvaća jedna riječ bez razmaka unutar jednostrukih navodnika ('string'). Također nisu implementirani komentari. Te dvije stvari su dosta bitne u jeziku ali su se pokazale malo teže za implementaciju, a i vezane su uz leksičku analizu koja nije tema ovog zadatka.

Način korištenja programa i primjer izvođenja

Gotovu aplikaciju pokreće se iz generirane izvršne datoteke (ECMAScript.exe) koja se nalazi u glavnom direktoriju projekta. Također je moguće pokrenuti gotov projekt u Visual Studiu otvaranjem Visual C# projektne datoteke (ECMAScript.csproj) ili cijelog Visual Studio rješenja (SSP.sln) koji se nalaze u build direktoriju. Pokretanjem aplikacije otvara se grafičko sučelje koje sadrži prozor za unos teksta, tablice u kojima se pohranjuju učitani uniformni znakovi, te prozor u kojem se ispisuje rezultat analize. Tekst željenog ulaznog programa unosi se ručno ili se učitava iz datoteke pritiskom na "Otvori datoteku". Primjeri ulaznih datoteka nalaze se u direktoriju ulazne datoteke.



Analiza koda počinje pritiskom na tipku "Analiziraj" nakon čega se u tablicama ispisuju pročitani znakovi, a u polju "Izlaz" se ispisuje da je program uspješno analiziran ili da je naišao na pogrešku. Nakon analize jednog niza moguće je odmah unijeti novi niz te ponovno izvršiti analizu.



Uz projekt su još priloženi alati GPLEX i GPPG koji su korišteni za generiranje C# datoteka koje se nalaze u tools direktoriju. Uz alate se nalaze i ulazne .y i .lex datoteke iz kojih je generiran program.

Zaključak

Sintaksna analiza može se jednostavno izvesti uz pomoć alata koji generira parser kao što je YACC. YACC ili u ovom slučaju GPPG jednostavni su za korištenje i uvelike ubrzavaju i olakšavaju postupak izgradnje parsera. Prednost GPPG-a je što koristi C# koji je jednostavan i moderan objektno orijentirani jezik široke namjene. Nedostatak GPPG-a je prilično loša dokumentacija te je potrebno dosta vremena za shvatiti osnovni princip rada. Povezivanje sa LEX-om odnosno kreiranje vlastitog leksičkog analizatora također je komplicirano zbog nedostatka boljih primjera. Kad se shvati osnovni princip i kad se napravi poveznica sa leksičkom analizom GPPG je vrlo jednostavan za korištenje i sintaksna analiza svodi se na pisanje gramatike za zadani jezik. YACC u kombinaciji sa LEX-om (GPLEX-om) je moćan alat kojim se može većim dijelom automatizirati prve tri faze izgradnje jezičnog procesora to jest leksičku, sintaksnu i semantičku analizu.