Miloš Puzović

MC – modifikacija C okruženja prilagođena modularnom programiranju

U ovom radu je predstavljena nova metoda modifikacije prednjeg dela C prevodioca sposobnog da pruži osobine modularnog programiranja u C okruženju. Modularnost programskog jezika se ogleda u mogućnosti apstrakcije promenjivih i zavisnog prevođenja. Promene su izvršene na prednjem delu C prevodioca u vidu promena atributa tabele simbola, nadogradnji sintakse jezika i načinu prevođenja izvornih programa (tj. nezavisno prevođenje je zamenjeno zavisnim prevođenjem). Obavljene su pomoću programskih paketa Flex i YACC, za leksičku odnosno sintaksnu analizu, a kao krajnji rezultat dobijen je međukod sa mogućnošću generisanja mašinskog koda.

Principi modularnosti

U modularnim jezicima postoje tri vrste modula: lokalni, programski i globalni moduli. Lokalni moduli u blokovsko-struktuiranim programskim jezicima su implementirani pomoću funkcija i programskih blokova, dok je programski (kontrolni) modul prisutan u obliku glavne datoteke koja povezuje zavisne module. Globalni modul se sastoji iz para modula: definicionog i implementacionog modula. Definicioni modul definiše interfejs modula prema ostalim modulima, odnosno definiše i deklariše imena koja modul izvozi. Implementacioni modul implementira (tj. realizuje) sva imena iz definicionog modula, uz mogućnost definisanja novih. Sva imena deklarisana u implementacionom modulu (a ne nalaze se u definicionom modulu) imaju lokalni karakter, odnosno nisu vidljiva van implementacionog modula (Marković *et al*. 1990). Implementacioni i definicioni modul su dve fizički razdvojene datoteke, koje se odvojeno prevode. Definiciona datoteka i implementaciona datoteka moraju imati isto ime, jer određenom definicionom modulu odgovara istoimeni implementacioni modul. Osim pomenutih pravila, modularni jezik mora da podržava i sledeće principe modularnosti (Mašulović *et al*. 1997): (1) mogućnost nastavljanja pro-

Miloš Puzović (1983), Čačak, Gradsko šetalište 55/30, učenik 2. razreda Gimnazije u Čačku

grama u više datoteka; (2) zavisno prevođenje (*dependent compilation*); (3) sakrivanje informacija (*information hiding*); (4) inkrementalnost (posle izmene nekog modula automatski se prevode svi moduli koji zavise od tog modula). Od gore navedenih osobina, C prevodilac poseduje samo mogućnost rastavljanja programa u više datoteka, kao i inkrementalnost, ali u vidu posebnih programskih paketa (Makefile u UNIX ili .prj datoteke u Windows okruženju) nezavisnih od prevodioca. Prevodilac programskog jezika C ne podržava apstrakciju promenjivih na nivou jednog zasebnog modula i nije u stanju da automatski prevodi zavisne module. C prevodilac ne podržava navedene osobine zbog toga što on koristi nezavisno prevođenje, što je veoma slab koncept budući da sve promenjive deklarisane u jednoj programskoj datoteci moraju biti ispravno pozvane. Da bi C prevodilac imao potpunu kontrolu nad datim modulima, njih je potrebno prevoditi određenim redosledom zbog međuzavisnosti koda. Prema tome, programski jezik C nije u potpunosti modularan jezik.

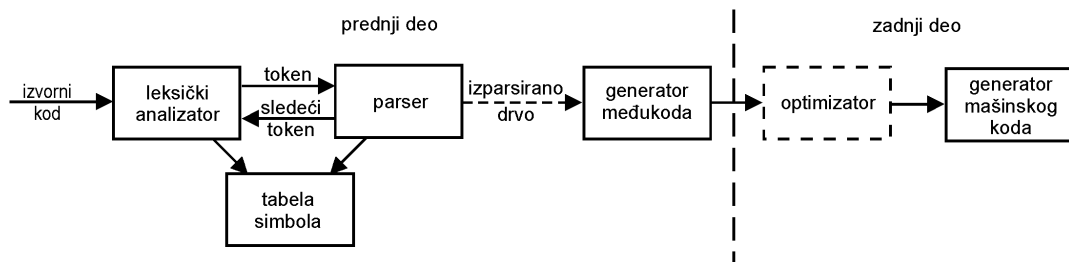
Cilj rada

Cilj ovog rada je promena prednjeg dela C prevodioca u okruženje koje bi imalo mogućnosti modularnog programiranja. Da bi C okruženje moglo da radi sa modulima bilo je potrebno izmeniti način opisa leksičkog života promenjive, nadograditi sintaksu jezika i promeniti način prevođenja izvornih programa (tj. koristiti zavisno prevođenje). Inače, korišćenjem principa izloženih u ovom radu, većina blokovsko-struktuiranih jezika može da prilagodi svoje okruženje za rad sa modulima.

Koncept prevodilaca

Problem sastavljanja prevodilaca je dobro poznat i nije se menjao od vremena kada su nastali prvi prevodioci. On predstavlja složenu celinu koja se može podeliti na prednji (*front end*) i zadnji (*back end*) deo (slika 1).

Uloga prednjeg dela je da izvrši analizu na izvornom kodu i generisanu gramatiku prosledi do zadnjeg dela koji generiše spreman mašinski kod za izvršavanje. Opšta podela prednjeg dela prevodioca je na leksičku analizu (*scanner*), sintaksnu (*parser*) i semantičku analizu (*semantic analy-*



Slika 1. Posebne celine prevodioca

Figure 1. Phases of a compiler

sis). Uloga leksičkog analizatora je da prođe kroz tekst (izvorni kod) programa i dodeli jednoznačne identifikatore ključnim rečima, promenjivim, relacionim operatorima itd. Kasnije se ti identifikatori koriste u određivanju njihovih sintaksnih vrednosti. Određivanje sintaksnih vrednosti vrši sintaksni analizator koji se zbog skupa pravila koje služe za određivanje konstrukcije jezika zove i kontekstno nezavisna gramatika (*context-free grammar*) ili samo gramatika. Zadnji deo prevodioca se može podeliti na optimizator i generator koda (*code generator*). Uloga optimizatora je da na osnovu posebnih kriterijuma (npr. dužina programa ili brzina izvršavanja) stvori ekvivalentan kod koji je po tim kriterijumima bolji od polaznog. U zavisnosti od dizajna prevodioca, ove celine manje-više mogu biti preplетene. Generator koda ima ulogu da na osnovu skupa semantičkih akcija koje su pridružene sintatičkim konstrukcijama stvori konačan oblik prevedenog programa.

Dizajn Modula-C prevodioca

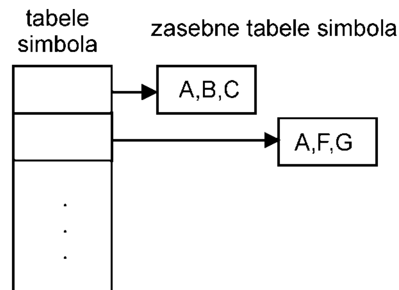
Implementacija

Realizovani prevodilac MC (Modula-C) je implementacija prednjeg dela prevodioca sposobnog da podrži rad sa modulima u C okruženju. Prevodilac se sastoji iz više celina. Kao leksički analizator korišćen je standardni UNIX program Flex, koji je kompatibilan sa Lex-om (Flex 1995). Za sintaksnu analizu je korišćen standardni UNIX program YACC (Johnson 1974). Proširenja u formalnoj gramatici programskog jezika C data su u prilogu rada. U analitičko-sintaksnom modelu prevodioca odrađena je i međukodna reprezentacija (*intermediate code representation*), zbog određenih dodatnih mogućnosti koje prevodilac dobija sa njom (opis dodatnih mogućnosti je dat u daljem delu teksta). Zadnji deo prevodioca nije implementiran, pošto cilj rada nije bio da se generiše i odgovarajući mašinski kod.

Tabela simbola

Vrlo važna opcija MC je čuvanje deklarisanih identifikatora koji kasnije moraju biti upotrebljeni u povezivanju zavisnih modula. Kao skladišni prostor, kako u prevodiocima tako i u MC se koristi tabela simbola. U odnosu na standardnu tabelu simbola koja se nalazi u C prevodiocima, nova tabela simbola morala je da sadrži nove atribute. Njih je bilo potrebno dodati da bi se potpuno opisao leksički život promenjive u drugim modulima. Pošto je C struktuiran jezik, broj tabela simbola zavisi od broja programskih blokova. Tabele simbola organizovane su u obliku steka. Pošto stek radi na principu *last-in-first-out* (LIFO), na vrhu steka se nalazi lokalni modul koji je u samoj unutrašnjosti (slika 2).

```
int A,B,C;
 {
  float D,E;
   {
    /* ... */
   }
  char A,F,G;
  {
   /* ... */
   /* trenutna pozicija i programu */
   /* ... */
  }
}
```



Slika 2. Implementacija tabele simbola blokovsko-struktuiranih jezika

Figure 2. Individual Table Implementation of Nested Scopes

Funkcije za rad sa tabelom simbola se mogu naći u literaturi (Fischer *et al.* 1991) i jedan takav primer je korišćen u implementaciji tabele simbola u MC.

Leksička analiza

Leksička analiza u MC poverena je Flex prevodiocu zbog njegove velike rasprostranjenosti. Flex prevodilac kao ulaz koristi notacije zasnovane na regularnim izrazima (Aho *et al.* 1986) – flex jezik. Ovaj programski paket je korišćen u ovom radu jer se regularni izrazi mogu kombinovati sa akcijama koje su potrebne da se izvrše npr. unošenje identifikatora u tabelu simbola, generisanje definicionih i implementacionih modula i dr. Flex jezik se može koristiti, iako se Flex prevodilac ne nalazi na sistemu; ulazna specifikacija se može ručno prevesti korišćenjem prelaznih dijagrama (*ibid.*), što je još jedna dodatna mogućnost portabilnosti MC prevodioca. Uloga leksičke analize je da pročita izvorni kod (tekst) i da ključne reči i operatore prenese parseru (sintaksnoj analizi), a identifikatore upiše u tabelu simbola. U slučaju da Flex naiđe na globalne tipove u modulu (definisane sa **extern**) on šalje sintaksnoj analizi informaciju o generisanju defincionog modula. Slično ovoj funkciji, Flex pravi listu modula koji zavise od modula na kome trenutno vrši leksičku analizu. Lista zavisnih modula je sortirana prema zavisnosti modula i prema zadnjem prevođenju zavisnih modula. Pošto su leksičke greške u programima veoma česte, Flex-u je dodata mogućnost da ispravi određene greške u kodu u cilju lakšeg parsiranja. Pored korišćenja Flex paketa za leksičku analizu, korišćen je i YACC, u funkciji sintaksne analize. Zbog zavisnosti između YACC i Flex potrebno je svaki token generisan regularnim izrazima preneti sintaksnoj analizi, odnosno parseru.

Sintaksna analiza

MC predstavlja prošireni skup programskog jezika C. U procesu rada MC, sintaksna zajedno sa semantičkom analizom predstavlja pretposlednji

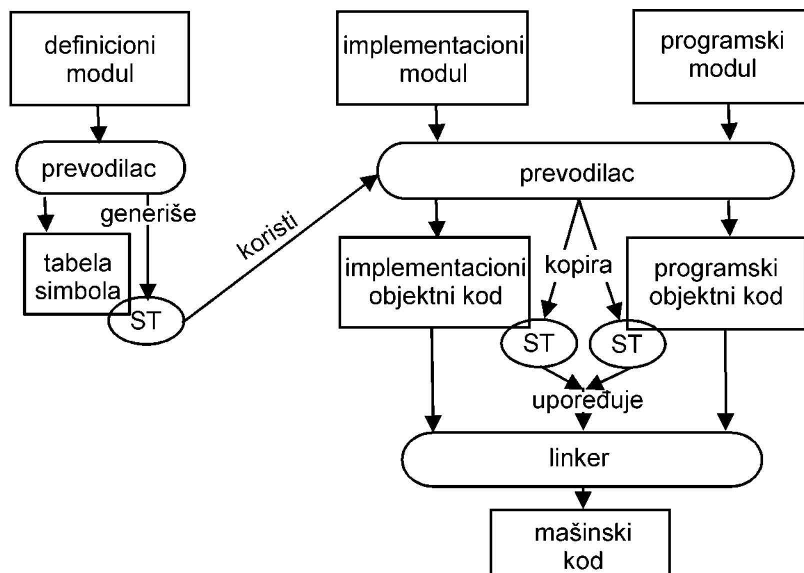
deo lanca prednjeg dela prevodioca. Pošto globalni moduli nisu implementirani u programskom jeziku C, bilo je potrebno izvršiti nadogradnju njegove formalne gramatike (BNF notacija nadogradnje gramatike je data u prilogu). Izvršena nadogradnja formalne gramatike sastojala se u opisivanju definicija globalnih modula, zato što MC mora da ima mogućnost razlikovanja definicionog od implementacionog modula. Nakon ovih promena, u zaglavlju definicionog modula potrebno je navesti njegovu deklaraciju: **definition module** *modul* {**from** *modul* **import** *tip promenjiva* }; (ključna reč **from** ima značenje poziva određenog defincionog modula; pomoću ključne reči **import** pozivaju se samo određene promenjive, ali ukoliko nije naznačena koriste se sve promenjive iz navedenog modula). Isto tako implementacioni modul u svom zaglavlju mora da ima deklaraciju: **implementation module** *modul* {**from** *modul* **import** *tip promenjiva* }; (ključne reči **from** i **import** imaju isto značenje kao i kod defincionong modula).

Ove promene formalne gramatike bile su neophodne da bi se omogućio rad sa modulima. Pored nadogradnje formalne gramatike, sintaksnoj analizi je dodata mogućnost sprečavanja kolizije između imena istih varijabli u različitim modulima. Za tu svrhu se koristi operator @, koji ima značenje poziva promenjive iz navedenog modula. Prateći semantiku poziva promenjivih iz struktura u programskom jeziku C, analogno njima pre operatora @ se navodi ime modula, pa potom promenjiva iz traženog modula (*modul@promenjiva*). Sa druge strane formalna gramatika je zadužena i za proveru ispravnosti poziva promenjivih iz definicionog modula. U slučaju da je promenjiva definisana u definicionom modulu, ali ne i za njega ekvivalentom implementacionom modulu, MC će javiti grešku definisanja i poziva promenjivih. Takođe, MC ima mogućnost da generiše definicione module za promenjive koje su definisane ključnom reči **extern** u implementacionom modulu, tako da programer nema obavezu da ih piše. Ali, zbog velike koristi definicionih modula ostavljena je mogućnost njihovog implicitnog definisanja. Zbog načina na koji paket YACC manipuliše greškama, MC može da prijavi samo osnovne sintaksne greške, a ne i druge greške koje su uslovljene sintaksom.

Redosled prevođenja modula

Da bi C programsko okruženje u potpunosti podržavalo module bilo je potrebno izvršiti promene u načinu prevođenja izvornog koda. Kao što je prikazano na slici 3, definicioni, implementacioni i programski modul se odvojeno prevode.

Proces prevođenja se odvija u dve faze. Prva faza je prevođenje izvornog koda u mašinski kod koji još nije spreman za izvršavanje (objektni



Slika 3.
Redosled prevođenja modula u mašinski kod

Figure 3.
Rule of translating modules into target language

kod). Potom sledi druga faza u kojoj se prevedni program povezuje (linkuje) sa mašinskim oblicima svih modula od kojih zavisi.

Prva faza prevođenja modula

Prvo se prevode definicioni moduli. Definicioni modul zavisi od svih definicionih modula koje uvozi, pa se stoga pre prevođenja nekog modula moraju prevesti svi moduli od kojih zavisi taj modul. Osobina MC je da se ne prevode svi moduli, već samo oni koji su kasnije menjani od modula koji se prevodi. Posle prevođenja definicionih modula, prevode se implementacioni moduli. Implementacioni modul zavisi od svog definicionog modula i od definicionih modula koje uvozi. Na kraju se prevodi programski modul. Programski modul zavisi samo od definicionih modula i pri njegovom prevođenju nisu potrebni gotovi implementacioni moduli.

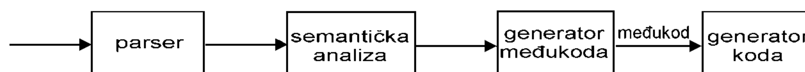
Druga faza prevođenja modula

Pre povezivanja programskog modula u izvršni program potrebno je: prevesti sve implementacione module koje programski modul uvozi i prevesti sam programski modul. Na kraju je potrebno da se oni povežu u izvršni program.

Generisanje međukoda

Zadnja faza u dizajnu prednjeg dela prevodioca je generisanje međukoda. Međukod je zapis koji se produkuje na osnovu semantičkih akcija. Mogućnost generisanja međukoda je jedna od značajnih osobina sadašnjih

prevodilaca. Iako se izvorni program može direktno prevesti u mašinski kod, prevodilac dobija dodatne mogućnosti u slučaju generisanja međukoda: (1) prevodilac za mašinu sa različitim resursima može biti kreiran dodavanjem novog zadnjeg dela na već postojeći prednji deo; (2) mašinsko nezavisna optimizacija koda može biti dodata na međukodnu reprezentaciju. U MC se podrazumeva da je pre generisanja međukoda izvršena kako sintaksna, tako i semantička analiza (slika 4).



Slika 4. Pozicija međukoda u dizajnu prevodioca

Figure 4. Position of intermediate code generation in the compiler design

Jedan od načina zapisivanja i implementacije međukoda je troadresni kod (Aho *et al.* 1986) koji je korišćen u ovom projektu.

Mogućnosti i nedostaci Modula-C

Mogućnosti postignute menjanjem nemodularnog C okruženja u modularno okruženje su sledeće: (1) lakše programiranje: svaki programer u okviru tima može da nezavisno od ostalih programira svoj modul; (2) lakše traženje grešaka: testiranje i traženje grešaka je ograničeno na samo jedan modul, jer on predstavlja logički nezavisnu programsku jedinicu; (3) smanjeno pravljenje grešaka: podacima jednog modula može se pristupiti samo posredstvom izvezenih operacija; (4) lakše menjanje programa; (5) veća čitljivost programa: modularno koncipiran program je pregledniji i jednostavniji za razumevanje i (6) ponovno korišćenje delova programa (*code reusability*): jedan dobro definisani modul se može koristiti i u drugim programima. Ove nove mogućnosti pridodate programskom jeziku C, preko prevodioca MC, svrstavaju ga u rang modularnih jezika kao što je Modula-2. Iako je dobio nadogradnju formalne gramatike, programski jezik C je i dalje zadržao svoje karakteristične osobine portabilnosti, generisanja mašinskog koda i prenosivosti. Dodavanjem modularnog okruženja na već postojeće, prenosivost C koda više ne zavisi samo od "spoljnih" faktora (npr. standardizacije).

MC nema mogućnosti generisanja mašinskog koda, pa zbog toga ne može biti korišćen kao zamena trenutno vodećim prevodiocima C jezika. Ipak, MC ima mogućnost generisanja međukoda koji kasnije pomoću asemblerskih listinga može biti preveden za bilo koju računarsku mašinu. Zbog korišćenja programskih paketa Flex i YACC, za leksičku odnosno sintaksnu analizu, prijavljivanje grešaka je svedeno samo na grešku leksičke ili sintaksne analize, bez većeg opisa greške.

Dalji razvoj

Na sadašnjem stepenu razvoja MC ima sve mogućnosti koje ima svaki prednji deo C prevodioca. Pored svih tih mogućnosti dodata je i mogućnost modularnog programiranja. Mogućnosti daljeg razvoja se kreću u smeru izrade potpuno novog sistema za prijavljanje grešaka pri prevođenju programa, jer se Flex i YACC nisu pokazali kao najbolje rešenje. Takođe, u daljem razvoju prevodioca postoji ideja da se pojednostavi pozivanje promenjivih iz različitih modula i još više smanji mogućnost kolizije između imena istih promenjivih. Pored nekoliko dodatnih funkcija koje MC dodaje standardnim C prevodiocima, u daljem stepenu razvoja projekta trebalo bi da bude i rada na "opismenjavanju" prevodioca u vidu *log* fajlova. Koristeći principe izložene u ovom radu, većina struktuiranih jezika može da modifikuje svoje programsko okruženje, kako bi podržavalo rad sa modularnim programiranjem.

Zahvalnost. Rad je svojim najvećim delom razvijan u Istraživačkoj stanici Petnica, koja mi je na raspologanje stavila veliku količinu stručne literatura, kao i pomoć stručnih saradnika. Sugestije Zorana Rilaka i Milana Gornika su predstavljale nezamenjivu pomoć u razvoju ovoga projekta. Posebnu pomoć u finalizaciji ovog projekta mi je pružio rukovodilac seminara računarstva Dragan Toroman.

Literatura

Aho A.V., Sethi R., Ulman J.D. 1986. *Compilers: Principles, Techniques and Tools*. Addision Wesley

Fischer C.N., LeBlanc R.J. 1991. *Crafting A Compiler With C*. The Benjamin/Cummings Publishing Company

Flex 1995. *The Fast Lexical Scanner Generator*. Berkeley: Lawrence Berkeley Labs.

Johnson S.C. 1975. YACC – Yet Another Compiler-Compiler. *Computing Science Technical Report*. Murray Hill (N.J): AT&T Bell Laboratories

Marković M., Simić R. 1990. *Struktuirano i modularno programiranje: iskustva, metode i primene*. Beograd: Naučna knjiga

Mašulovic D., Budimac Z. 1997. Da li je C dobar programski jezik. *Tangenta*, 1: 7-14.

Miloš Puzović

MC: Modification of C environment for supporting modular programming

Realized compiler Modula-C is implementation of modifided front end of programming language C. MC (Modula-C) is superset of programming language C, which includes modular programming in its environment.

Programming language is modular when he has next odds: dividing programs into modules, separate compilation, information hiding and incrementality. Of all these odds, programming language C has only opportunity of dividing programs into modules and incrementality using special programming packages. Instead of dependent, C uses independent compilation which is very soft concept because it doesn't maintain variable access level and doesn't have option of compiling dependent modules. Because programming language C doesn't have these odds, MC makes major changes to the front end of the compiler.

One of the demand changes is to the symbol table, where MC puts new attributes in it. New attributes have options of full describing variable lexical life, which is very important part in designing programming environment for modules. Except changes to the symbol table, MC makes appropriate modification at the formal grammar of C. With those modifications programming language C has now option to recognize which kind is module and to prevent collision between same variable names in different modules. MC doesn't change syntax of C, its only integrates these modification. Also, MC changes way of compiling C source code. Separate compilation is changed to the independent compilation. For lexical and syntax analysis MC uses, respectively, programming packages Flex and YACC.

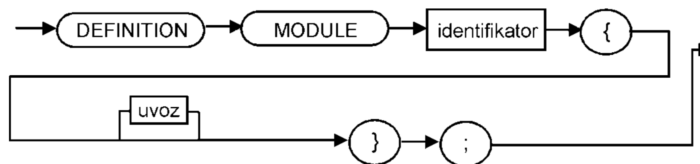
Finally, the last operation used in the front end of the compiler is intermediate code generation. Although, a source program can be translated directly into the target language, some benefits of using an independent representation are: a compiler for different machine can be created by attaching a back end for the new machine to an existing front end and later code optimizer can be applied to the intermediate representation. In this stage of development MC doesn't have possibility of generating target code, but he has opportunity to generate intermediate code which lately can be translated to machine language using assembly listings.

Future development of MC could present design of new lexical and syntax analysis because Flex and YACC didn't show much results on that field. Except changes to the analysis-synthesis part of the compilers front end, in the future will be made changes to the communication between the compiler and human in the form of *log* files. This project presents a new approach in changing non-modular programming language environment into environment which support modular programming.

Prilog: Nadogradnja sintakse progamskog jezika C

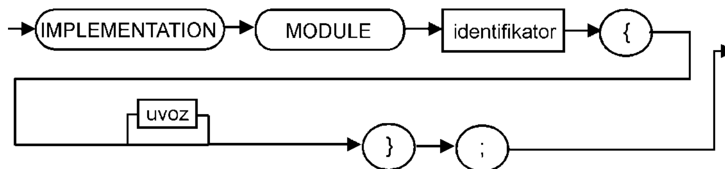
Sintaksa i sintaksni dijagram definicionog modula

ImePromenjive = “_” | slovo { slovo | cifra }
Tip = VOID | INT | CHAR | DOUBLE | SHORT | LONG | FLOAT
ImeModula = slovo { slovo | cifra }
ListaPromenjivih = Tip ImePromenjive { “,” Tip ImePromenjive }
Uvoz = [FROM ImeModula] IMPORT ListaPromenjivih “;”
DefModul = DEFINITION MODULE ImeModula “{” Uvoz “}”;



Sintaksa i sintaksni dijagram implementacionog modula

ImePromenjive = “_” | slovo { slovo | cifra }
Tip = VOID | INT | CHAR | DOUBLE | SHORT | LONG | FLOAT
ImeModula = slovo { slovo | cifra }
ListaPromenjivih = Tip ImePromenjive { “,” Tip ImePromenjive }
Uvoz = [FROM ImeModula] IMPORT ListaPromenjivih “;”
ImpModul = IMPLEMENTATION MODULE ImeModula “{” Uvoz “}”;



Sintaksa i sintaksni deklaracije poziva promenjive

ImeModula = slovo {slovo | cifra }
ImePromenjive = “_” | slovo { slovo | cifra }
ImeModula “@” ImePromenjive