Parametrizovani univerzalni editor programskog teksta

mr Zorica Suvajdžin

- doktorska disertacija -

Mentor:
prof. dr Miroslav Hajduković

Novi Sad
Novembar 2007.

Sadržaj

1 Uvod **1**
1.1 Pristup editovanju programskog teksta 3
1.2 Okruženje 4
1.3 Strukturno-orijentisano okruženje 6
1.4 Interna reprezentacija 8
1.5 Strukturne operacije 10
1.6 Formalna specifikacija 11
1.7 Primena jezički-zasnovanih sistema 13
1.8 Tok istraživanja 13

2 Srodni radovi **15**
2.1 Editori 16
2.2 Programska okruženja 19
2.2.1 *Cornell Program Synthesizer* 19
2.2.2 *ABC structure editor* 21
2.2.3 Uporedni prikaz programskih okruženja 23
2.3 Generatori programskih okruženja 23
2.3.1 *Mentor* 24
2.3.2 *Centaur* 25
2.3.3 *Synthesizer Generator* 28
2.3.4 *Programming System Generator (PSG)* 29
2.3.5 *ASF + SDF Meta-environment* 31
2.3.6 *SmartTools* 34
2.3.7 *Pan* 35
2.3.8 Uporedni prikaz generatora programskih okruženja 36
2.4 Generička programska okruženja 38
2.4.1 *Generic Syntax-directed Editor* (*GSE*) 38

2.4.2 *ORM Environment* 40
2.4.3 *Agora Structure Editor (ASE)* 43
2.4.4 *Muir* 45
2.4.5 Uporedni prikaz generičkih programskih okruženja . 47
2.5 Uporedni prikaz okruženja 47
2.6 Ostali alati 49
2.7 Opisi jezika 50
2.7.1 Definicija sintakse 50
2.7.2 Definicija semantike 51
2.7.2.1 *Operational semantics* 51
2.7.2.2 *Denotational semantics* 52
2.7.2.3 *Natural semantics* 52
2.7.2.4 *Axiomatic semantics* 53
2.7.2.5 *Attribute Grammar* 53

3 Cilj istraživanja **55**
3.1 Postavljeni ciljevi 56
3.2 Izbor dizajna editora 58
3.3 Izbor korisničkog interfejsa editora 60
3.3.1 Izbor strukturnih operacija 62
3.4 Izbor interne reprezentacije 64
3.5 Izbor formalnog opisa specifikacije 65
3.6 Izbor arhitekture okruženja 66
3.7 Razlike u odnosu na postojeće sisteme 67

4 Opis rešenja **69**
4.1 Arhitektura sistema 69
4.2 Specifikacija programskog jezika 70
4.2.1 Opis leksičke strukture jezika 72
4.2.2 Opis osnovnih tipova podataka jezika 73
4.2.3 Opis operacija i operatora jezika 74
4.2.4 Opis imena 75
4.2.4.1 *lvalue* referenca 75
4.2.5 Opis sintaksne strukture jezika 76
4.2.5.1 Akcije 77
4.2.6 Operatori 77
4.2.6.1 Opis statičke semantike jezika 78
4.2.6.2 Opis provere tipova 81
4.2.6.3 Opis formata prikaza 81

4.3 Specifikacioni parser 82
4.4 Specifikacione tabele 83
4.5 Tabela simbola 84
4.6 Editorska mašina i editorski interfejs 84
4.6.1 Funkcionalne osobine editorske mašine 86
4.6.2 Kontekstno editovanje 87
4.6.2.1 Strukture 87
4.6.2.2 Dijalozi 88
4.6.2.3 Kretanje po programskom tekstu i njegovo markiranje 90
4.6.2.4 Rukovanje strukturama i dijalozima 91
4.6.2.5 Prednosti kontekstnog editovanja 97
4.7 Neimplementirana funkcionalnost 97
4.7.1 Neimplementirane osobine jezika 97
4.7.2 Neimplementirane operacije 99

5 Zaključak 101
5.1 Procena ostvarenih ciljeva 101
5.2 Prednosti realizovanog strukturnog editora 102
5.3 Odnos *USE* editora i kompajlera 103
5.4 Budući rad 104

Bibliografija 107

A Rečnik pojmova 121

B Specifikacija programskog jezika *C* 127

Popis tablica

2.1 Sekvenca potrebnih operacija za primer sa slike 2.1 18
2.2 Uporedni prikaz programskih okruženja (? označava nedostupnost podataka) 23
2.3 Uporedni prikaz generatora programskih okruženja (? označava nedostupnost podataka) 37
2.4 Uporedni prikaz generatora programskih okruženja (nastavak) 38
2.5 Uporedni prikaz generičkih programskih okruženja (? označava nedostupnost podataka) 47
2.6 Uporedni prikaz okruženja na osnovu sadržanih alata . . . 48
2.7 Uporedni prikaz okruženja na osnovu sadržanih alata(nastavak) 49

3.1 Pregled editora na osnovu operacija koje nude 59
3.2 Pregled editora 60

Poglavlje 1

Uvod

Problem zadavanja, pregledanja i modifikacije programskog teksta umnogome zaslužuje pažnju, iako nije jedan od ključnih problema procesa razvoja softvera. Rešavanje ovog problema je bitnije za programere početnike i neprofesionalne programere nego za profesionalce. Ali, čak ni za njih ovaj aspekt nije nebitan, pošto tehnologija editovanja može znatno doprineti njihovoj većoj produktivnosti. Jezički-zasnovani sistemi (*language-based systems*) predstavljaju važan napredak u tehnologiji softverskog inženjerstva. Ovi sistemi omogućuju rukovanje programskim dokumentima u smislu formalnog jezika, kao i na nivou njihovih tekstualnih karakteristika. Jezički-zasnovani sistemi se baziraju na formalnoj strukturi programskog jezika. Nekoliko generacija eksperimentalnih [8, 9, 7, 87] i praktičnih [65] jezički-zasnovanih sistema za editovanje je napravilo bitan progres u interaktivnoj jezičkoj tehnologiji (*interactive language technology*).

Jezički-zasnovani sistemi su izgrađeni na strukturnoj perspektivi razvoja programa, jer programi, po svojoj strukturi predstavljaju drvo, i njima treba rukovati na taj način. Osnovni cilj jezički-zasnovanih sistema je da pruže podršku razvoju i održavanju programskih dokumenata. Centralno mesto u svakom od ovih sistema zauzima strukturni editor. Jezički-zasnovani sistemi za editovanje su interaktivni alati, i njihova namena je značajno povećanje produktivnosti korisnika [80]. Ovi sistemi koriste informacije dobijene primenom jezičke (lingvističke) analize, da bi ponudili šire mogućnosti

(usluge) od tradicionalnih editora. Uprkos dugotrajnim naporima u razvoju jezički-zasnovanih sistema za editovanje, još uvek ne postoji sistem koji je u širokoj praktičnoj upotrebi.

U mnogim okruženjima, čitanje je i dalje dominantan posao programera, čak i kada piše kod. Istraživanja pokazuju da se razumevanje koda putem čitanja bitno unapređuje pomoću visokokvalitetne, jezički-vođene tipografije (*linguistically-driven typography*) [92]. S druge strane, programeri imaju "strukturni" način razumevanja programa. Međutim, većina programera nije baš voljna da menja svoje duboko ukorenjene navike editovanja. Oni će prihvatiti samo onaj alat koji im nudi naprednije usluge editovanja, udobne za korišćenje, i koji je u toj meri intuitivan da se može koristiti bez dodatne obuke. Zato se usluge editovanja koje pruža alat moraju osmisliti pažljivo, tako da ne uvode ograničenja (ili da ta ograničenja budu jedva primetna) [27]. Specijalizovana unapređivanja editorskih usluga (operacija) su veoma važna, ali je još važnije da ona ne sputavaju postupak editovanja. Bilo kakvo nametanje u postupku editovanja mora biti izbalansirano između korisnika i alata, a to ume da bude veoma delikatan zahtev. Da bi bio efikasan, jezički-zasnovan sistem za editovanje treba da ponudi širok spektar editorskih operacija, visokokvalitetnu grafiku i visok stepen prilagodljivosti [89, 92]. Danas se, prilikom razvoja softvera, stavlja akcenat na primenu novih tehnologija, sa ciljem povećanja kvaliteta razvoja programa i povećanja produktivnosti programera. Kvalitet ovih interaktivnih sistema mora biti što je moguće bliži komercijalnim (kao što su, u današnje vreme, *Visual Studio Environment* [98] ili *Eclipse* [99]).

Proučeno je četrnaest jezički-zasnovanih sistema i njihovih (strukturnih) editora. Ovaj proces nije bio ni malo lak, iz dva razloga. Prvo, neke implementacije jezički-zasnovanih sistema nisu javno dostupne, a one koje su dostupne nisu izvršive na *Windows* ili *Linux* platformama. Drugo, većina dokumenata opisuje dizajn editora, ali ne i korisnički interfejs, koji je predmet interesovanja ove teze. Iz istraživanja svojstava pomenutih sistema i editora, zaključeno je da funkcionalna struktura postojećih strukturnih editora ostavlja prostora za poboljšanja. Glavni cilj istraživanja je prepoznati probleme od kojih pate postojeći strukturno-orijentisani editori, i u implementaciji prototipa ih ukloniti (ili ih bar smanjiti).

Tradicionalni pristup editovanju programskog teksta je zasnovan na znakovno orijentisanom korisničkom interfejsu. On je formiran pod

uticajem prvih znakovno orijentisanih ulazno/izlaznih uređaja i imao je za cilj podršku interakciji čovek-računar. Iako je kasnija pojava grafičkih terminala i pokazivačkih uređaja (*pointing devices*) izazvala radikalne izmene korisničkog interfejsa, to nije esencijalno izmenilo editore programskog teksta. Zato se pojavila dilema da li programski editori nisu menjani jer ne postoji način da se to uradi, ili je u pitanju samo inercija. Ovo istraživanje pokušava da odgovori na ovu dilemu, implementirajući novi pristup editovanju programskog teksta.

1.1 Pristup editovanju programskog teksta

Editovanje i kompajliranje su aktivnosti u razvoju programa, koje se nadovezuju jedna na drugu. One su međusobno povezane. Tokom editovanja nastaju greške editovanja, koje se otkrivaju u procesu kompajliranja a otklanjaju tokom ponovljenog editovanja. Ispravljanje grešaka editovanja neretko uzrokuje višestruka naizmenična ponavljanja editovanja i kompajliranja. Ona se mogu izbeći, a postupak razvoja programa skratiti, ako se spreči pravljenje grešaka editovanja u toku komponovanja programskog teksta. Programski tekst ne sadrži greške editovanja, ako uspešno prolazi leksičku, sintaksnu i semantičku analizu u toku kompajliranja.

Tekstualni editori ne sprečavaju pojavu grešaka editovanja, jer programski tekst posmatraju kao običan tekst. Oni, zbog toga, dozvoljavaju ubacivanje proizvoljnih nizova znakova u programski tekst. To znači i nizova znakova koji nisu legalni sa stanovišta leksike, sintakse i semantike programskog jezika. Isto važi i za sintaksne editore (*syntax-oriented editors*), koji samo olakšavaju uočavanje nelegalnih nizova znakova. Sintaksni editori to postižu tako što prepoznaju i označavaju rezervisane reči programskog jezika, čime povećavaju preglednost programskog teksta. Ni neki strukturni editori ne sprečavaju pojave grešaka editovanja, nego eventualno ukazuju na njihovu pojavu. Ovi strukturni editori to postižu tako što programski tekst posmatraju kao niz znakova, na koga primenjuju leksičku, sintaksnu i semantičku analizu.

Greške editovanja se mogu sprečiti, ako, u toku komponovanja programskog teksta, korisniku stoje na raspolaganju samo ispravne komponente, koje korisnik može kombinovati samo na ispravan način. To se može postići, ako se uvedu posebni obrasci za pojedine sintaksne

konstrukcije, i ako se korisniku dozvoli da komponuje programski tekst samo od ovakvih obrazaca. Popunjavanjem obarazaca, korisnik kompletira sintaksnu konstrukciju. Obrasci se popunjavaju na način koji sprečava unos pogrešnih sadržaja.

Prototip, nastao kao rezultat ovog istraživanja, implementira ovakav pristup sintezi programa.

1.2 Okruženje

Okruženje (*environment*) predstavlja skup hardverskih i softverskih alata koje korisnik (*system developer*) koristi za razvoj nekog softverskog sistema [14]. Kako tehnologija napreduje, tako se povećevaju očekivanja korisinika, pa funkcionalnost okruženja mora da se menja. Tokom poslednje dve do tri decenije, softverski alati koji se nude korisniku su značajno unpaređeni. Razlog pravljenja integrisanog okruženja, umesto skupa nezavisnih specifičnih alata, je u tome što integracija pruža mogućnost korišćenja zajedničkih servisa/usluga.

Programsko okruženje (*programming environment*) i razvojno okruženje (*software development environment*) su pojmovi koji se često koriste kao sinonimi, ali između njih se može napraviti razlika [14]. Pod programskim okruženjem misli se na okruženje koje podržava samo fazu kodiranja u razvojnom ciklusu softvera. To uključuje podršku *programming-in-the-small* zadacima, kao što su editovanje i kompajliranje. Pod razvojnim okruženjem podrazumeva se okruženje koje povećava ili automatizuje aktivnosti softverskog razvojnog ciklusa. Ono uključuje *programming-in-the-large* zadatke kao što je upravljanje izgledom i oblikom projekta (*configuration management*) i *programming-in-the-many* zadatke kao što su upravljanje projektom i ljudima (*project and team management*). Pojam razvojnog okruženja uključuje i okruženja koja omogućavaju dugoročno održavanje softvera (*long-term software maintenance*).

Trendovi koji imaju veliki uticaj na okruženja (i njihove alate, korisničke interfejse i arhitekture) se mogu razvrstati u četiri kategorije [14]:

Language-centered environments Ova okruženja su napravljena za jedan jezik, i stoga imaju alate koji su prilagođeni tom jeziku. Odlikuju se visokim stepenom interakcije, ali nude ograničenu

podršku *programming-in-the-large* zadacima. Jedno ovakvo okruženje služi i za razvoj kao i za izvršavanje, testiranje, dibagiranje i izmenu koda.

Structure-oriented environments Ova okruženja objedinjuju tehnike koje omogućuju korisniku da direktno manipuliše strukturama. Tehnike su postale nezavisne od jezika, što je dovelo do ideje o generatorima okruženja, a kasnije i generičkim okruženjima. Ova okruženja će kasnije u tekstu biti detaljnije razmotrena.

Toolkit environments Ova okruženja se sastoje od skupa malih alata koji podržavaju fazu kodiranja u razvoju softvera. Ovaj pristup kao osnovu koristi operativni sistem, i dodaje alate za razvoj programa kao što su kompajler, editor, asembler, linker i dibager, ali i alate za jezički nezavisnu podršku *programming-in-th-large* potrebama, kao što su upravljanje projektom (*configuration management*) i kontrola verzija (*version control*). Namera je da se odgovarajućim alatima obezbedi jezički nezavisno okruženje koje podržava više jezika. Motivacija ovog pristupa je proširivost (*extensibility*) i prenosivost (*portability*) alata. Primer komercijalnog *toolkit* sistema je *Unix Programmer's Workbench* (*Unix/PWB*).

Method-based environments Ova okruženja nude podršku za određenu metodu razvoja softvera. Metode se mogu podeliti u dve klase: metode razvoja pojedinih faza u procesu razvoja softvera i metode za upravljanje procesom razvoja. Metode za razvoj koriste se u fazama kao što su analiza zahteva (*requirements analysis*), specifikacija sistema (*system specification*) i dizajn (*design*). Metode za upravljanje procesom pružaju mogućnost uređenog razvoja softvera (koji razvija više ljudi) i organizovanje i upravljanje ljudima i aktivnostima (*team and project management*, *programming-in-the-many*). Primeri alata koji nude metode razvoja su *CASE* alati, koji imaju velike grafičke mogućnosti editovanja.

Smisao ove taksonomije je prikaz trendova, pre nego kategorizacija pojedinih okruženja. Pojedinačno okruženje može imati osobine iz nekoliko kategorija. Ove kategorije ne predstavljaju konkurentske tačke gledišta, već naprotiv, predstavljaju glavne tehničke pravce i oblasti koje su inspirisale dalja istraživanja i razvoj okruženja.

1.3 Strukturno-orijentisano okruženje

Inicijalna motivacija za struktuno-orijentisana okruženja je bila da pruže korisniku interaktivni alat - sintaksno-vođeni editor - za unos programa u obliku jezičkih konstrukcija. Mogućnosti su kasnije proširene, pa je obezbeđeno *single-user* programsko okruženje koje podržava interaktivnu semantičku analizu, izvršavanje programa i dibagiranje. Editor je centralna komponenta takvih okruženja. On je interfejs kroz koji korisnik interaguje i kroz koji se modifikuju sve strukture. Vremenom je pojam *syntax-directed* postepeno bio zamenjen pojmom *structure-oriented*.

Strukturno-orijentisana okruženja su napravila nekoliko doprinosa tehnologiji okruženja: ponudila su direktnu manipulaciju programskim strukturama, višestruko prikazivanje programa iz iste programske strukture, inkrementalnu proveru statičke semantike i sematičkih informacija dostupnih korisniku, i najvažnije, mogućnost da se formalno opiše sintaksa i statička semantika jezika iz kojeg se može napraviti (izgenerisati) instanca strukturnog editora.

U strukturno-orijentisanim okruženjima je realizovan koncept direktne manipulacije strukturom. Korisnik direktno menja programske (jezičke) konstrukcije i time je lišen napora oko pamćenja detalja sintakse programskog jezika. Većina ovakvih okruženja, kao internu reprezentaciju, koristi apstraktno sintaksno drvo (*abstract syntax tree*). U nekim strukturno-orijentisanim okruženjima zastupljeno je čisto strukturno editovanje, u kome ne postoji mogućnost da korisnik napravi sintaksno neispravan program. U drugim strukturno-orijentisanim okruženjima korisnik može raditi i sa tekstualnom i sa strukturnom reprezentacijom. Korisnik unese delove programa u obliku teksta, a okruženje inkrementalnim parsiranjem pretvara tekst u programsku strukturu. Korisnik može editovati program korišćenjem komandi koje se mogu primeniti i na tekst i na strukturu. Da bi ovo bilo moguće, okruženje mora obe reprezentacije držati uvek ažurne.

Strukturno-orijentisana okruženja generišu tekstualnu reprezentaciju programa iz strukture. Tako se iz iste strukture mogu generisati različite reprezentacije. Ova osobina omogućuje korisnicima ovih okruženja prikaz programa na različitim nivoima apstrakcije i sa razeličitim nivoima detalja.

Posle prvih sintaksno-vođenih editora, brzo se ispostavilo da

je primoravanje na ispravnu sintaksu samo jedan način podrške programeru. Kao nova podrška, editorima je dodata analiza statičke semantike. Semantički analizator procesira strukturu programa i dopunjuje pozadinsko drvo semantičkim informacijama. Korisnik može pristupiti semantičkim informacijama (kao što su definicija identifikatora i lokacija njegove upotrebe) i informacijama o tipu, kroz editor. Na primer, dok korisnik edituje poziv funkcije, editor može prikazati zaglavlje funkcije i, čim se unese ime funkcije, ponuditi templejt za unos parametara.

Strukturno-orijentisano okruženje je interaktivan alat. Zato, ono treba da odmah korisniku pruži povratnu informaciju o sintaksnim greškama, i da ga obavesti o semantičkim greškama. Ovo znači da okruženje mora pratiti izmene programa (poznavati u svakom trenutku strukturu programa) i ponovo analizirati samo one delove programa koji su afektirani editovanjem. Dosadašnja tehnologija pravljenja kompajlera je uspešno proširena da podrži takvu inkrementalnu analizu. Različiti tipovi analize se mogu vršiti na različitim nivoima granularnosti. Na primer, sintaksna ispravnost se može forsirati na nivou jezičkih konstrukcija, dok se statička semantika može proveravati na nivou programske jedinice (na primer na nivou procedure), a kod se može generisati na nivou modula.

Jedan od najvažnijih doprinosa strukturno-orijentisanih okruženja je mogućnost manipulisanja programskim strukturama na način koji je nezavisan od jezika. Dizajneri okruženja postižu ovo tako što enkapsuliraju sintaksne i semantičke osobine jezika u gramatiku. Kada im se ponudi deklarativni opis jezika, alati za generisanje mogu automatski izgenerisati instancu strukturno-orijentisanog okruženja. Ovo je mnogo efikasnije od pravljenja okruženja od početka. Informacije o jeziku se mogu čuvati u formi koja se može učitati u jezgro jezički-nezavisnog okruženja u toku izvršavanja, dozvoljavajući time da jedna instanca okruženja razume nekoliko programskih struktura istovremeno.

Prednosti strukturnih okruženja su veoma značajne:

- podržavaju direktnu manipulaciju strukturama
- omogućen je višestruki istovremeni prikaz iste strukture
- informacije o statičkoj semantici se mogu vezati za strukturu programa, i time postati dostupne korisniku

- moguća je inkrementalna analiza ovih informacija
- instance okruženja se mogu automatski generisati iz formalne specifikacije jezika
- formalne specifikacije jezika opisuju i sintaksu i statičku semantiku na deklarativan način.

Strukturno-orijentisana okruženja su uglavno prihvaćena kao pomoć u nastavi (*teaching aids*). Univerziteti ih koriste u procesu nastave na kursevima uvodnog programiranja. Uprkos njihovoj dostupnosti, slabo su prihvaćeni u industriji. Strukturni editori se koriste kao podrška u fazi kodiranja i smatraju se alatima za podršku *programming-in-the-small* zadacima. Tehnike koje se trenutno koriste u mnogim strukturno-orijentisanim okruženjima imaju nedostatke u smislu obezbeđivanja efikasnog, trajnog smeštanja velikih struktura i u koordiniranju konkurentnog pristupa strukturama za više korisnika ili alata. Dalje, da bi se različiti alati integrisali u strukturno-orijentisano okruženje, moraju biti ili prilagođeni zajedničkoj strukturnoj reprezentaciji ili mora postojati mehanizam za stalno ažuriranje strukture kroz višestukre prikaze (*views*).

1.4 Interna reprezentacija

Tradicionalni tekstualni editori kao internu reprezentaciju koriste niz znakova. Za razliku od njih, strukturni editori kao internu reprezentaciju koriste drvo (*tree*). Ova struktura podataka kod većine strukturnih editora reprezentuje apstraktnu sintaksu programskog jezika. Zato se i zove apstraktno sintaksno drvo.

Drvo [18] je hijerarhijska struktura podataka i sastoji se od čvorova. Čvor, ispod sebe, može da sadrži jedan ili više čvorova. Ti čvorovi se nazivaju potomci (*child node*, *descendant*). Čvor koji sadrži potomke se naziva roditeljski čvor (*parent node*). Čvor koji se nalazi na vrhu drveta se zove korenski čvor (*root node*). Korenski čvor nema roditeljski čvor. Svi ostali čvorovi imaju tačno jedan roditeljski čvor. Korenski čvor je predak svim čvorovima u drvetu.

U usmerenom drvetu, na istom hijerarhijskom nivou (u odnosu na neki posmatrani čvor) razlikuju se naredni čvor (*successor node*)

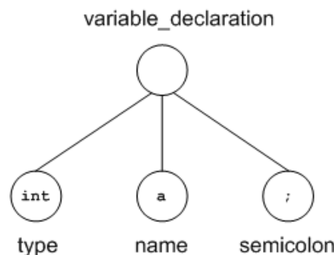
i prethodni čvor (*previous node*). Takvi čvorovi na istom hijerarhijskom nivou imaju zajednički roditeljski čvor.

Čvor drveta koji ima potomke se naziva unutrašnji čvor (*internal node*), a čvor koji nema potomke se naziva list (*leaf*).

Veza (*edge*) povezuje dva čvora u drvetu, od kojih je jedan roditelj drugom. Putanja (*path*) u drvetu je skup veza koje povezuju dva čvora, od kojih je jedan predak onom drugom. Dužina putanje (*length*) predstavlja broj veza u putanji. Visina čvora (*height*) je dužina najduže putanje od tog čvora do lista. Dubina čvora (*depth*) je dužina (jedinstvene) putanje od tog čvora do korenskog čvora. Poddrvo (*subtree*) drveta sa korenskim čvorom *n* je drvo koje je napravljeno od originalnog drveta tako da se sastoji od čvora *n* kao korenskog čvora i svih njegovih potomaka.

Kao i većina tekstualnih editora, i strukturni editori imaju pojam selekcije (*current selection*) unutar strukture koja se edituje. Selekcija je poddrvo (čvor sa svojim potomcima) na koje se većina editorskih operacija odnosi.

Ovakva interna reprezentacija je zajednička (sa manjim ili većim varijacijama) za sve alate koji se bave analizom jezika. Na primer, za deklaraciju promenljive (na programskom jeziku *C*), struktura drveta bi mogla da izgleda kao na slici 1.1:



Slika 1.1: Moguća interna reprezentacija deklaracije promenljive na *C* programskom jeziku

Čvor koji opisuje deklaraciju promenljive se sastoji od tri potomka. Prvi potomak opisuje tip promenljive (*type*) i sadrži string `int`. Drugi potomak opisuje ime promenljive (*name*) i sadrži string `a`. Treći potomak opisuje delimiter (*semicolon*) i sadrži string `;`.

Postoji nekoliko mogućih implementacija drveta [2]:

niz
: Svaki čvor je numerisan. Broj čvora se koristi kao indeks niza. Element niza sadrži oznaku roditeljskog čvora ili *NULL* za koren. Mana ovog pristupa je što ne postoji direktan pristup informaciji o potomcima.

lista
: Za svaki čvor se formira lista njegovih potomaka. Mana ovog pristupa je što je teško doći do informacije o roditeljskom čvoru.

leftmost-child right-sibling representation
: Svaki čvor poseduje vezu sa prvim potomkom (*leftmost-child*) i vezu sa sledbenikom (*right-sibling*). Sve operacije se jasno implementiraju, osim operacije *parent* koja zahteva pretraživanje cele reprezentacije.

1.5 Strukturne operacije

Sve strukturne operacije odnose se na hijerarhijsku strukturu koja predstavlja internu reprezentaciju programskog teksta koji se edituje. Strukturne operacije direktno rukuju strukturom. Sve modifikacije programskog teksta, odnosno strukture, dešavaju se relativno u odnosu na trenutnu poziciju kursora. Pozicija kursora definiše selektovanu (označenu, trenutno aktivnu) strukturu. To je struktura na koju se primenjuje aktivirana strukturna operacija.

Osnovne operacije koje strukturni editor može ponuditi korisniku su one koje se koriste za sintezu programa. U osnovi, ove operacije se mogu podeliti u dve kategorije: navigacija i modifikacija.

Operacije kretanja kroz strukturu istovremeno imaju i ulogu pozicioniranja, odnosno, selektovanja (označavanja) strukture. Za navigaciju kroz strukturu drveta koriste se četiri osnovne operacije:

left
: premešta selekciju sa trenutno selektovanog čvora na čvor koji mu prethodi, ukoliko takav čvor postoji

right
: premešta selekciju sa trenutno selektovanog čvora na sledeći čvor, ukoliko takav čvor postoji

up
: premešta selekciju sa trenutno selektovanog čvora na roditeljski čvor, ukoliko takav čvor postoji

down premešta selekciju sa trenutno selektovanog čvora na prvog potomka, ukoliko takav čvor postoji.

Kombinacijom ove četiri osnovne navigacione operacije nastaju kompozitne navigacione operacije. Na primer, operacija *last* selektuje poslednjeg potomka (ukoliko takav čvor postoji), tako što prvo pozove operaciju *down*, a zatim nekoliko puta operaciju *right*. Za navigaciju kroz strukturu drveta koristi se još i operacija *root* koja premešta selekciju sa trenutno selektovanog čvora na korenski čvor drveta.

Operacije koje modifikuju strukturu drveta su:

insertion dodavanje nove strukture, pre ili posle označene strukture (ubacivanje novog čvora ili poddrveta)

replace izmena označene strukture (zamena postojećeg čvora/poddrveta novim čvorom/poddrvetom)

deletion uklanjanje označene strukture (brisanje postojećeg čvora ili poddrveta).

Kako se ove operacije oslanjaju na specifikaciju programskog jezika, tako se, njihovom upotrebom, i modifikacija strukture programskog teksta vrši na način koji obezbeđuje sintaksnu ispravnost. Zato je, u čisto strukturnom editoru, praktično nemoguće napraviti sintaksno neispravni program.

Dodatne strukturne operacije su one koje proširuju mogućnosti strukturnog editora. One su specifične za svaku implementaciju strukturnog editora. U ove operacije spadaju: pronalaženje čvora sa zadatim osobinama (*find*), automatska indentacija, sintaksni *highlighting*, rukovanje komentarima (retko se nalazi u strukturnim editorima [16]), proširenje selekcije, razni semantički upiti, višestruka selekcija (mogućnost istovremenog postojanja više selektovanih struktura), strukturna transformacija (zamena jednog fragmenta programa drugim) i slične operacije.

1.6 Formalna specifikacija

Programski jezik ima dva fundamentalna svojstva: sintaksu i semantiku. Sintaksa se odnosi na izgled ispravnih programa, a semantika na značenje tih programa. Sintaksa jezika se može

formalizovati pomoću gramatike ili sintaksnog dijagrama (i takva formalizacija se uglavnom nalazi na kraju priručnika za programski jezik). Semantika bi, takođe, trebala da bude formalizovana, ali uglavnom nije (i u priručnicima se opisuje neformalno) [66].

Prednosti formalne specifikacije sintakse programskih jezika su dobro poznate:

- formalizacijom sintakse se standardizuje zvanična sintaksa jezika. Ovo je bitno korisnicima, kojima je potrebno vođenje u pisanju sintaksno ispravnih programa, i onima koji implementiraju jezik, jer moraju napraviti ispravan parser za kompajler.
- formalizacija sintakse omogućuje formalnu analizu njenih svojstava (da li je definicija *LL*, *LR* ili dvosmislena)
- definicija sintakse može biti i ulaz alatima (kao što je *Yacc*) koji generišu *front-end* kompajlera (ili interpretera).

Obezbeđivanje formalne specifikacije semantike programskog jezika, nudi slične prednosti:

- formalizacijom semantike se standardizuje zvanična semantika jezika. Ovo je bitno korisnicima, kojima je potrebno vođenje u razumevanju programa koje pišu, i onima koji implementiraju jezik, jer moraju napraviti ispravan generator koda za kompajler.
- formalizacija semantike omogućuje formalnu analizu njenih svojstava (na primer, da li je definicija *strongly-typed*)
- definicija semantike može biti i ulaz alatima koji generišu *back-end* kompajlera (ili interpretera).

Svi navedeni faktori doprinose unapređenju dizajna i razvoju programskih jezika. Programski jezici koji su dizajnirani sa jednom od raznih formalnih metoda imaju precizniju sintaksu i semantiku, manje izuzetaka i lakše ih je naučiti.

Štaviše, iz formalne specifikacije mnogi drugi jezički alati mogu biti automatski izgenerisani. To su: *pretty-printers*, *syntax-directed editors*, *type checkers*, *dataflow analyzer*s, *partial evaluators*, *debuggers*, *profilers*, *test case generators*, *visualizers*, *animators*, *documentation generators*, itd [30]. Kod većine generatora, definicija samog jezika

nije dovoljna, već se specifikacija mora proširiti informacijama koje su specifične za alate.

1.7 Primena jezički-zasnovanih sistema

Primena jezički-zasnovanih sistema je raznovrsna i mnogobrojna. U nastavku slede primeri mogućih primena ovakvih sistema:

- za razvoj ili generisanje sintaksno-vođenih ili strukturnih editora
- za razvoj ili generisanje okruženja za kreiranje, editovanje, izvršavanje i dibagiranje programa
- za razvoj ili generisanje raznih alata koji rukuju strukturnim podacima (*desk-calculator*, *proof-checker*, *text-fotmatting editor*, *active document applications* [63], ...)
- za brzi razvoj prototipa okruženja
- za razvoj ili generisanje interaktivnih sistema za pravljenje (formalne) specifikacije jezika i generisanje alata za te jezike
- za brzi razvoj jezika (*language prototyping*), što podrazumeva dizajn, analizu i razvoj jezika
- za razvoj alata za analizu i transformisanje programa (u oblasti *reverse engineering* i *system renovation*) [86]
- u edukativne svrhe, na kursevima uvodnog programiranja i na kursevima pravljenja kompajlera (*compiler construction*)

1.8 Tok istraživanja

Istraživanje jezički-zasnovanih sitema, autor je započeo početkom 1999. godine, radom na svojoj magistarskoj tezi [71]. U tu svrhu proučeni su postojeći sintaksno-vođeni i strukturni editori. Kao konkretan rezultat istraživanja, nastao je prototip sintaksno-vođenog editora za programski jezik *ST* (*Structured Text*, *International Standard IEC 1131-3*, *Programmable Controllers*, *Part 3: Programming Languages* [1, 24]) [75, 76, 74]. Ovaj programski jezik je namenjen programiranju

programabilnih kontrolera (*PLC*, *Programmable Logical Controlers*). Razvijeno programsko okruženje obuhvata editor za *ST* programski jezik (*STeditor*), translator sa programskog jezika *ST* na programski jezik *C* (*ST2C* translator), *C* kompajler, linker, *PLC* punilac i *PLC* izvršnu podršku. Za razvoj okruženja, korišteni su postojeći *C* kompajler, linker i *PLC* izvršna podrška, a razvijeni su (ne u punoj funkcionalnosti) *STeditor*, *ST2C* translator i *PLC* punilac. Editor je namenjen isključivo *ST* programskom jeziku, i sve sintaksne i strukturne operacije su ugrađene u editor [25, 26].

Razvijeno programsko okruženje je korišćeno za razvoj složene aplikacije, namenjene za upravljanje velikim industrijskim pogonom. Konačna korisnička reakcija na *STeditor* je bila veoma povoljna (nakon početnih nedoumica i nesporazuma i uprkos delimičnoj funkcionalnosti editora). Korisnici su bili programeri, vični jedino programiranju programabilnih kontrolera, pa tako ostaje pitanje da li se korisnički utisci mogu generalizovati.

Istraživanje je nastavljeno stazom uopštavanja. Cilj je bio napraviti generator ovakvih strukturnih editora. Generatoru bi trebalo saopštiti u nekoj formi opis programskog jezika, i on bi proizveo editor za dati programski jezik. Da bi se, krećući od prvobitnog prototipa, postigao ovaj cilj, bilo je potrebno uopštiti strukturne operacije iz *STeditora*-a.

Radom na uopštavanju operacija, zaključeno je da nema potrebe praviti generator editora, već je moguće napraviti generički editor. To znači, da ne moraju postojati posebno generator i posebno editor, već može postojati samo jedan softverski alat - editor, koji prepoznaje više programskih jezika. Njemu je potrebno saopštiti formalni opis programskih jezika, i to samo jednom, jer editor te informacije čuva u svojoj bazi programskih jezika. Od tog momenta, pa nadalje, pomoću editora je moguće editovati programe u datim programskim jezicima [73, 72].

U nastavku rada opisan je prototip ovakvog generičkog strukturnog editora.

Drugo poglavlje sadrži detaljan pregled istraženih postojećih, jezički-zasnovanih sistema. U trećem poglavlju je predstavljen zaključak o stanju ovih sistema i mestima gde je moguć napredak. Četvrto poglavlje opisuje implementaciju prototipa. U petom poglavlju su izneseni zaključci istraživanja, i pravci budućeg razvoja.

Poglavlje 2

Srodni radovi

Razvoj prvih kompajlera, krajem 50-tih godina prošlog veka, bez adekvatnih alata je bio vrlo komplikovan i zahtevan posao. Na primer, za implementaciju kompajlera za programski jezik *Fortran* trebalo je 18 čovek-godina [31]. Kasnije su razvijene formalne metode kao što su: *operational semantics*, *attribute grammar*, *denotational semantics*, *action semantics*, *algebraic semantics* i *abstract state machines*. One su olakšale implementaciju programskih jezika i doprinele su automatizaciji generisanja kompajlera, odnosno interpretera. Napravljeni su mnogi alati, i svi su zasnovani na različitim formalnim metodama i procesiraju različite delove specifikacije jezika. Takvi alati su: generatori skenera (*scanner generator*), generatori parsera (*parser generator*) i generatori kompajlera (*compiler generator*). Primarni cilj takvih sistema je bio automatsko generisanje kompletnog kompajlera. Međutim, istraživači su uskoro prepoznali mogućnost da se i mnogi drugi jezički-zasnovani alati mogu izgenerisati iz formalne specifikacije jezika. Sada postoje mnogi alati koji ne samo da generišu kompajler, već kompletna jezički-zasnovana programska okruženja. Ovako izgenerisana okruženja sadrže editor, *type-checker*, dibager, razne analizatore i slične alate.

U početku, programeri su za pravljenje programa koristili samo tradicionalni tekstualni editor i kompajler, kao dva zasebna alata. Da bi se postupak pisanja programskog koda olakšao, nastali su strukturni editori. Ovi editori su posmatrali programski tekst kao strukturu sastavljenu od podstruktura, a ne kao niz znakova. Kako

bi mogli raditi na nivou struktura, ovi editori su interno morali imati znanje o sintaksi a kasnije i semantici programskog jezika kojem su namenjeni. Nastale su razne varijante strukturnih editora, sa širokim spektrom strukturnih operacija. Programiranje je u mnogome bilo olakšano nastankom programskih okruženja, koja su dizajnirana specijalno za neki jezik. Ovakva okruženja su nudila specijalizovane alate za kreiranje, modifikaciju i prikaz programa, kao i za njihovu analizu, prevođenje i izvršavanje. U ovakvim sistemima, centralna komponenta bio je strukturni ili hibridni editor. Ubrzo, nastala je ideja o generatorima takvih okruženja. Generatorima je trebalo saopštiti formalnu specifikaciju jezika, da bi on proizveo programsko okruženje za taj programski jezik. U ovom periodu, istraživanja su bila fokusirana na formalizme kojima se mogu opisati osobine jezika (apstraktna i konkretna sintaksa, statička i dinamička semantika, generisanje koda, formati prikaza), kao i na mogućnost generisanja raznovrsnih alata za dati jezik. Dalja istraživanja u ovoj oblasti dovela su do razvoja generičkih programskih okruženja, koja ne generišu okruženja, već se ponašaju u skladu sa formalnom definicijom jezika koja joj se prosledi.

U nastavku je opisan razvoj editora, a zatim je dat pregled najvažnijih predstavnika programskih okruženja, generatora programskih okruženja i generičkih programskih okruženja.

2.1 Editori

U zavisnosti od načina interakcije sa korisnikom, postoji nekoliko vrsta editora. Ako korisnik menja tekst, koji se zatim parsira spoljašnjim alatom da bi se napravilo odgovarajuće drvo parsiranja, onda je editor tekstualni. Ukoliko korisnik menja drvo, koje se zatim upotrebom *pretty-printer*-a pretvara u odgovarajući tekst, onda se editor zove strukturni editor. Ako je korisniku dozvoljeno da menja i tekst i drvo parsiranja, takav editor se naziva hibridni editor [9, 42].

Jednostavni tekstualni editori nemaju jezičku (lingvističku) podršku. Oni nude jednostavno editovanje, bez specijalizacije za programski kod. Informacije o strukturi programskog teksta koje ovakav editor može da prikupi su nekompletne i neprecizne. Zato, on ne može pružiti usluge koje zahtevaju pravu analizu programskog koda.

Sintaksni editori (*syntax-oriented editors, code-oriented editors*), kao

što je *Emacs* [69], koriste čisto tekstualnu reprezentaciju, uz dodatno prepoznavanje određenih jezičkih konstrukcija upotrebom regularnih izraza. Sintaksni editori obezbeđuju usluge editovanja kao što su indentacija i *highlighting*.

Strukturno-orijentisani način editovanja programskog teksta je zasnovan na konceptu direktne manipulacije strukturama. Dok je programski tekst prikazan na ekranu, korisnik direktno menja internu strukturu [14]. Interna reprezentacija programskog teksta je drvo. Ovo u mnogome pojednostavljuje neke vrste jezički-zasnovanih operacija, ali zahteva da korisnik edituje pomoću strukturnih komandi. Korisnik rukuje direktno programskim strukturama i zbog toga nema potrebu za detaljnim poznavanjem sintakse programskog jezika.

Strukturni editor nudi operacije editovanja samo za jezičke strukture, i ne dozvoljava korisniku da napravi sintaksno neispravan program. Problem sa nekim strukturno zasnovanim editorima je to što forsiraju silazni (*top-down*) pristup prilikom unošenja teksta, zbog izgradnje i obilaska korespondentnog apstraktnog sintaksnog drveta. Ovo sprečava prirodni način unosa teksta i može otežati izmenu teksta. Na primer, u ranim danima strukturnih editora, korisnici su se žalili na nezgodan način unošenja izraza sa infiksnim operatorima. Za ilustraciju koristiće se primer (preuzet iz [70]):

$$a * b + c * d$$

Slika 2.1: Primer programskog teksta

Za unošenje prethodnog izraza u običnom tekstualnom editoru potrebno je pritisnuti dirku tastature za svaki od 7 znakova izraza, znači ukupno 7 elementarnih editorskih operacija. U silaznom strukturnom editoru, ove elementarne operacije ubacivanja su pomešane sa navigacionim komandama. Sekvenca potrebnih operacija bi mogla izgledati kao na slici 2.1 ($\downarrow$, $\rightarrow$ i $\uparrow$ respektivno služe za navigaciju na dole do prvog potomka, na desno do sledbenika, i na gore do roditelja).:

	?
$+$	
	?+?
$\downarrow$	
	?+?
$*$	
	?*?+?
$\downarrow$	
	?*?+?
a	
	a*?+?
$\rightarrow$	
	a*?+?
b	
	$a * b$+?
$\uparrow$	
	$a * b$+?
$\rightarrow$	
	$a * b$+?
$*$	
	$a * b$+?*?
$\downarrow$	
	$a * b$+?*?
c	
	$a * b + c$*?
$\rightarrow$	
	$a * b + c$*?
d	
	$a * b + c * d$

Tablica 2.1: Sekvenca potrebnih operacija za primer sa slike 2.1

Mane silaznih strukturnih editora, prethodno ilustrovane relativno komplikovanim načinom unošenja teksta i njegove modifikacije, mogu da se reše hibridnim pristupom. Ovaj pristup podržava i strukturne i nestrukturne operacije, a pri tome se očuvava struktura apstraktnog sintaksnog drveta. Korisnik unosi delove programa u obliku teksta, dok okruženje kompletira obradu teksta koliko je to

moguće. Upotrebom tehnika inkrementalnog parsiranja, okruženje konvertuje deo teksta u apstraktno sintaksno drvo. U većini hibridnih sistema postoje dva režima editovanja: strukturni i nestrukturni. U jednom pristupu, korisnik može da se prebacuje iz jednog u drugi režim. Ova dva režima su potpuno različita, pa se u nestrukturnom režimu gube sve prednosti strukturnog pristupa. U drugom pristupu, režim editovanja se bira automatski, na osnovu gramatike dela teksta koji se edituje (npr. ispod određenog nivoa, čvorovi u drvetu se sastoje od nestrukturiranog teksta). Hibridni pristup, u obe varijante, krši zahtev modalnosti. To se dešava jer hibridni pristup zahteva eksplicitnu promenu režima (moda), što znači da tekstualno i strukturno editovanje nisu glatko integrisani.

Koncept strukturnog editovanja je primenljiv kako na programske editore tako i na grafičke editore [64, 82, 83]. Oni nude podršku grafičkim (dvodimenzionalnim) jezicima umesto tekstualnim (jednodimenzionalnim) jezicima [58].

2.2 Programska okruženja

U nastavku se kao najznačajniji predstavnici programskih okruženja razmatraju *Cornel Program Synthesizer* [79, 77] i *ABC structure editor* [51].

2.2.1 *Cornell Program Synthesizer*

Cornel Program Synthesizer je interaktivno programsko okruženje sa mogućnostima za kreiranje, editovanje, izvršavanje i dibagiranje programa. I editovanje i izvršavanje su vođeni sintaksnom strukturom programskog jezika. Gramatika programskog jezika je predstavljenja skupom unapred definisanih templejta (*template*) za sve jezičke konstrukcije, osim za najjednostavnije iskaze. Programi se kreiraju od gore na dole, ubacivanjem novih iskaza i izraza na mesto koje je označeno kursorom. Templejti naglašavaju strukturni način razmišljanja - da je program hijerarhijska kompozicija sintaksnih struktura, a ne samo sekvenca znakova.

Editor

Editor u ovom sistemu je hibrid između strukturnog i tekstualnog editora. Templejti se generišu izvršavanjem editorske komande, a izrazi i iskazi pridruživanja se unose tekstualno jedan po jedan znak. U templejtima se ne može desiti greška jer su oni predefinisani. Parser se poziva iz editora za frazu po frazu (*phrase-by-phrase*), a greške u tekstu koji unosi korisnik se detektuju odmah.

Hijerarhijska struktura se sastoji od dve vrste elemenata: templejta i fraze (*phrases*). Templejt je unapred definisan, formatiran niz znakova. Ključne reči, znakovi punktuacije i indentacija templejta se ne mogu menjati. Templejt, tako, predstavlja nepromenljiv okvir za ubacivanje dodatnih programskih struktura. Ležišta fraze (*placeholders*) identifikuju lokacije gde su ova ubacivanja dozvoljena. Svako ležište određuje sintaksnu klasu dozvoljenih ubacivanja. Fraza je proizvoljna sekvenca simbola (znakova). Fraze i templejti mogu biti ubačeni u druge templejte na lokacijama koje određuju ležišta, tako što zamene dato ležište. Sve modifikacije programskog teksta se dešavaju relativno u odnosu na trenutnu poziciju kursora (*editing cursor*). Iako se kursor može pomerati bilo gde unutar fraze, može se pozicionirati samo na krajnje levom simbolu (znaku) templejta ili ležišta fraze. Gornji krajnje levi simbol označava kompletan templejt (uključujući sve njegove podstrukture). Kursor se nikada ne pojavljuje u okviru ključnih reči u templejtu, niti na marginama. Kursor se može pomerati sa jednog na drugi templejt, i sa jednog templejta na njegove podstrukture, a ne samo sa jedne linije teksta na drugu. Kursor je moguće pozicionirati samo na onim mestima gde je ubacivanje ili brisanje dozvoljeno.

Izmena strukture programskog teksta se postiže ubacivanjem i brisanjem celih templejta ili fraze. Ovakav disciplinovani režim modifikacije programskog teksta garantuje strukturni integritet programa u svakom koraku editovanja. Izmena fraze se radi jednostavnom izmenom znakova od kojih se fraza sastoji. Nakon svake modifikacije, proverava se sintaksna ispravnost fraze.

Početni cilj *Synthesizer* sistema bio je da garantuje ispravnost programa u bilo kojoj tački njegovog razvoja, time što će svaka modifikacija koja uvodi grešku biti sprečena. Međutim, kontekstna ograničenja sintakse su primorala *Synthesizer* na tolerisanje neispravne fraze, pa ih on samo vizuelno naglašava (*highlight*) dok ne

postanu ispravne.

Izvršavanje

Tokom editovanja, programi se prevode i čuvaju u formi koja se može interpretirati, stoga ne postoji kašnjenje zbog prevođenja između editovanja i izvršavanja. Svaki put kada je izvršavanje suspendovano, kontrola se vraća editoru, kursor se postavlja na tačku suspenzije u izvornom kodu i ispisuje se poruka koja objašnjava zašto je izvršavanje suspendovano. Moguće je izvršavati nezavršene programe. Svaki put kada se u izvršavanju naiđe na ležište, izvršavanje se suspenduje. Izvršavanje se može nastaviti od te tačke nadalje, kada se doda deo programa koji nedostaje.

Istorijat

Dizajn i implementacija *Program Synthesizer* sistema započeli su u maju 1978. godine. Do decembra 1978. godine razvijene su verzije prototipa za demonstraciju, koje su bile operativne na *UNIX* sistemima i *Terak* mikroračunarima (*microcomputer*). Od juna 1979. godine, sistem je počeo da se koristi na *Cornell* univerzitetu, i godišnje ga je koristilo po 1500 studenata (programera). Prvi jezik koji je implementiran za *Synthesizer* je bio *PL/CS* (*instructional dialect of PL/I*), a kasnije je implementiran i *Pascal* programski jezik.

2.2.2 *ABC structure editor*

ABC structure editor [51] je deo programskog okruženja namenjenog interaktivnom programskom jeziku *ABC*.

Editor

Okruženje uključuje hibridni editor u kojem ne postoje različiti režimi editovanja. Editovanje se uglavnom sastoji od operacija koje vrše akciju nad selektovanom strukturom. Akcija može biti ubacivanje, brisanje ili zamena, a selektovana struktura može biti znak, reč, linija ili neka druga struktura. *ABC* editor podržava direktnu manipulaciju, pa selektovanje strukture prethodi odabiru akcije.

Trenutna selekcija u dokumentu se naziva još i fokus. Na ekranu, fokus se prikazuje naglašeno (*highlihted*). U čisto nestrukturnom editoru, selekcija se sastoji od bilo kog dela teksta. U strogo strukturnom editoru, selekcija se sastoji od poddrveta, pa se operacija selektovanja zato može posmatrati kao operacija navigacije kroz drvo. Struktura se može selektovati upotrebom miša (direktna i brza selekcija). Takva selekcija obuhvata najveću strukturu koja počinje na poziciji miša. *ABC* editor podržava operacije navigacije (pomeranja selekcije) koje se odnose na jedan čvor ili skup čvorova. Za uzlazno pomeranje po drvetu, koristi se operacija *widen*. Za silazno pomeranje po drvetu koriste se operacije *first* (za selektovanje prvog čvora naslednika) i *last* (za selektovanje poslednjeg čvora naslednika). Operacije *next* i *previous* vrše pomeranje selekcije na naredni i prethodni čvor, respektivno. Proširenje selekcije na naredni segment koji se sastoji od više čvorova je moguća pomoću operacije *extend*. Ova operacija proširuje trenutnu selekciju susednim čvorom s desne strane. Zbog zahteva korisnika, u editor su dodate i operacije koje selektuju celu narednu ili prethodnu liniju (*downline* i *upline*).

Prilikom unosa teksta editor pomaže korisniku tako što vrši predikciju i nudi korisniku predloge programskog teksta koji treba da usledi. Korisnik može da prihvati predloge ili da ih ignoriše i nastavi sa unosom teksta. Ovi predlozi tada predstavljaju templejte, sa parametrizovanim mestima koja treba dopuniti delovima programskog teksta.

Akcije koje nudi *ABC* editor uključuju još i unos teksta sa tastature, kopiranje i premeštanje postojećeg dela programskog teksta. U paru sa operacijom *undo*, postoji operacija *redo* koja poništava efekat *undo* operacije. Operacije *record* i *playback* omogućavaju korisniku da određeni niz operacija sačuva i da ih kasnije ponovi.

Istorijat

Pilot projekat *ABC* strukturnog editora je razvijen nad programabilnim editorom *Emacs*. Posle toga, napravljene su dve glavne iteracije. Prva je 1984. godine rezultovala prototipom koji je imao implementirane sve bitne ideje, i koji je tada javno objavljen. Druga iteracija je 1981. godine uvela neznatne promene u funkcionalnost editora. Sadašnji *ABC* editor koriste različite grupe korisnika, od studenata do profesionalaca.

2.2.3 Uporedni prikaz programskih okruženja

U tabeli 2.2 su uporedno prikazana bitna svojstva opisanih programskih okruženja.

	editor	alati	jezik	sintaksa	semantika
Cornell Program Synthesizer	hibridni	debugger, execution	*PL/CS*	skup templejta *hard-coded*	*hard-coded*
ABC structure editor	hibridni	?	*ABC*	*hard-coded*	*hard-coded*

Tablica 2.2: Uporedni prikaz programskih okruženja (? označava nedostupnost podataka)

2.3 Generatori programskih okruženja

Uloga generatora je da generišu interaktivna programska okruženja na osnovu formalne definicije jezika. Integrisano programsko okruženje se proizvodi iz formalne definicije sintakse, semantike i nekih dodatnih informacija (na primer: opis korisničkog interfejsa).

Kapacitet generatora okruženja potiče od postojanja zajedničkog kernela okruženja i posebnog deklarativnog opisa kojim se specificira okruženje. Zajednički kernel koriste (*reuse*) svi alati u okviru izgenerisanog okruženja. Novo okruženje se može specificirati na vrlo koncizan način, pomoću deklarativnog opisa. S druge strane, izgenerisano okruženje može da sadrži i neke osobine koje nisu tražene specifikacijom, jer postoje neke odluke generatora nad kojima specifikacija nema kontrolu.

Slika 2.2: Model generisanja programskih okruženja

Česta je praksa da se u formalnoj definiciji koja služi za generisanje strukturnih editora koristi apstraktna sintaksa i sintaksa koja opisuje format prikazivanja apstraktnog sintaksnog drveta (*unparsing rules*, *pretty-printing rules*).

U nastavku se razmatraju najznačajniji predstavnici generatora: *Mentor* [20], *Centaur* [13], *Synthesizer Generator* [44], *Programming System Generator* (*PSG*) [8], *ASF + SDF Meta-Environment* (*Algebraic Specification Formalism plus Syntax Definition Formalism*) [87, 85], *SmartTools* [7, 6, 60] i *Pan* [9, 89, 91, 44].

2.3.1 *Mentor*

Mentor je jedan od najranijih projekata koji su se bavili generisanjem programskih okruženja. Osnovne osobine sistema proizašlog iz ovog projekta su:

- programabilnost: komandni jezik koji se koristi za manipulaciju objektima sistema je stvarni interaktivni programski jezik *Mentol*. Ovaj jezik se može koristiti za razvoj biblioteka programa koje rukuju drvetom i transformišu drvo.

- nezavisnost od jezika: informacije specifične za jezik su smeštene u tabelama koje su eksterne za sistem. Kada se javi potreba za prebacivanjem na novi jezik, odgovarajuće tabele se učitaju, i sistem je vođen ovim informacijama specifičnim za jezik. Korisnik može definisati jezik upotrebom meta-jezika visokog nivoa *Metal*. Kada se *Metal* program koji sadrži formalizam F prevede pomoću *Mentor* sistema, dobijaju se tabele koje sadrže specifične informacije za F.

- višejezičnost (*multilingual*): moguće je manipulisati, istovremeno, sa više stabala koji pripadaju različitim formalizmima. Moguće je čak mešati više različitih formalizama unutar jednog drveta, tako što se čvorovima drveta pridruže anotacije.

Editor

Mentor sistem koristi hibridni editor. Prebacivanje iz režima strukturnog editovanja u režim tekstualnog editovanja je ekspilicitan. Za tradicionalno tekstualno editovanje koristi se *Emacs* editor. Nakon

editovanja teksta, korisnik mora da pokrene komandu *parse* koja izmenjeni tekst isparsira i zameni selektovano poddrvo.

Specifikacija jezika

Mentor sistem koristi *Metal* jezik za specifikaciju programskog jezika. *Metal* je jezik pomoću kojeg se definišu novi formalizmi. *Metal* specifikacija je skup gramatičkih pravila, dopunjenih anotacijama koje opisuju kakvo sintaksno drvo treba da se sintetiše. *Metal* specifikacija se sastoji od apstraktne sintakse, konkretne sintakse i skupa funkcija za sintezu drveta. Obrada semantike se opisuje *Mentol* jezikom.

Istorijat

Na početku projekta, 1974. godine, *Mentor* je bio programsko okruženje specijalizovano za programski jezik *Pascal*. Zatim je, 1975. godine, *bootstrap* metodom, pomoću *Mentor-Pascal* programskog okruženja razvijen *Mentor* za *Pascal*. Implementacija je završena 1977. godine. Od tada, *Mentor* umesto *Pascal* koda koristi apstraktnu sintaksu. 1980. godine napravljeno je kompletno *Pascal* okruženje koje je tada objavljeno i van *INRIA* instituta. *Metal* jezik je dizajniran i implementiran 1981. godine.

2.3.2 *Centaur*

Centaur sistem predstavlja interaktivno okruženje. Kada primi formalnu specifikaciju nekog programskog jezika (koja sadrži opis sintakse i semantike) *Centaur* generiše okruženje koje je namenjeno tom programskog jeziku. *Centaur* je naslednik *Mentor* sistema.

Arhitektura sistema

Arhitektura *Centaur* generatora se može posmatrati po slojevima:

- kernel: sloj za reprezentaciju i manipulaciju strukturnim objektima
- sloj specifikacije: za podršku i sintaksnih i semantičkih aspekata jezika

- korisnički interfejs: sloj koji obezbeđuje svu interaktivnu komunikaciju između sistema i korisnika.

Kernel ima glavnu ulogu u ovakvoj arhitekturi jer je namenjen da se koristi, direktno ili indirektno, od strane svih ostalih komponenti u sistemu. Svaki strukturni objekat u sistemu je reprezentovan apstraktnim sintaksnim drvetom, kojim rukuju funkcije kernela. Kernel se sastoji od apstraktne mašine (*abstract machine*) i logičke mašine (*logical machine*). Apstraktna mašina rukuje sintaksnim aspektima i zove se *Virtual Tree Processor* (*VTP*). Logička mašina rukuje semantičkim aspektima (svim vrstama evaluacija) i realizovana je kao *Prolog* interpreter. *VTP* definiše i implementira protokol za stvaranje, manipulaciju i trajno čuvanje apstraktnog sintaksnog drveta. Organizovan je oko dva koncepta: konteksti (*contexts*) i anotacije (*annotations*). Konteksti su vrednosti koje opisuju jednu ili više lokacija u drvetu. Anotacije su liste vrednosti (*tag*s), koje mogu biti vezane za apstraktno sintaksno drvo da bi sačuvale dodatne informacije.

Specifikacija jezika

Iz specifikacije sintakse generiše se strukturno-orijantisani editor, skener, (*multi-entry*) parser, *pretty-printer* i *abstract syntax tables*. Skener i parser se koriste za transformaciju tekstualne reprezentacije u strukturnu. *Unparser* (*pretty-printer*) se koristi za transformaciju strukturne reprezentacije u tekstualnu. *Abstract syntax tables* se koriste za proveru ispravnosti editorskih operacija.

Specifikacija konkretne i apstraktne sintakse, i njihova veza se piše upotrebom *Metal* formalizma (koji je razvijen za *Mentor* sistem [20]). Prevođenjem *Metal* specifikacije dobija se 1) *VTP* formalizam, 2) parser za konkretnu sintaksu i 3) generator drveta koji koristi *VTP* primitive da bi napravio apstraktno sintaksno drvo. Parser je rezultat prevođenja konkretne sintakse u format koji se predaje *YACC* ([36, 3]) kompajleru.

Personalized Print Markup Language (*PPML*) je namenski formalizam, koji se koristi za povezivanje konkretnog prikaza (*layout*) sa apstraktnom strukturom. Grubo govoreći, *PPML* specifikacija je uređena lista *pretty-printing* pravila, koja koristi *box language* za opis konkretnih prikaza. Nakon provere konzistentnosti sa definicijom jezika, *PPML* programi se prevode u *List* programe. Kada za neku

jezičku konstrukciju ne postoji *pretty-printing* pravilo, sistem koristi podrazumevana (*default*) pravila generičkog *pretty-printer*-a. *PPML* prevodilac je inkrementalni: kada se neko pravilo izmeni, samo to pravilo i pravila koja se u vezi sa njim se ponovo prevode.

Specifikacija semantike se može pisati na dva načina: na *Typol* jeziku (verzija *Natural Semantics*), ili upotrebom *ASF* formalizma (*Algebraic Specification Formalism*). Specifikacije se prevode na *Prolog*. U *Centaur* sistemu postoji okruženje za pisanje *Typol* specifikacija.

Korisnički interfejs

Korisnički interfejs nudi mogućnost više različitih prikaza (*views*) jednog apstraktnog sintaksnog drveta. Izmena u jednom prikazu se reflektuje u ostalim. Svi prikazi su tekstualni (grafičke reprezentacije nisu implementirane). Strukturno i tekstualno editovanje su realizovani kao posebni prikazi pozadinskog drveta. U prikazu je uvek selektovan neki podizraz. On je prikazan naglašeno - podebljanim (*bold*) slovima da bi se razlikovao od ostatka teksta. Sve komande se odnose na ovaj podizraz.

Implementacija

Centaur je implementiran upotrebom programskih jezika *Lisp* i *Prolog*. Dizajn i implementacija sistema zasnovani su na iskustvima u vezi dizajna, implementacije i upotrebe *Mentor* sistema ([20]). *Mentor* sistem je implementiran upotrebom programskog jezika *Pascal*.

Komponente sistema koje koriste *VTP* su: 1) *Metal* kompajler, koji pravi novi formalizam i parser za njega, i 2) *PPML* kompajler, koji iz simboličkog opisa generiše *pretty-printer*. Kombinovanjem ove dve komponente nastaje jednostavni strukturno-orijentisani editor.

Primena

U radovima koji opisuju sistem je saopšteno da je samo nekoliko odabranih korisnika zaista imalo iskustvo sa sistemom.

2.3.3 *Synthesizer Generator*

Uspeh sistema *Program Synthesizer*, koji je inicijalno ručno razvijen, podstakao je na kreiranje *Synthesizer Generator* sistema za generisanje takvih okruženja. *Synthesizer Generator* je verovatno najšire rasprostranjeni sistem za generisanje programskih okruženja. Sastoji se od sintaksno-vođenog (*syntax-directed*) editora za blok-struktuirane (*block-structured*) jezike, kompajlera, interpretera i dibagera.

Editor

Izgenerisano okruženje sadrži hibridni editor, u kojem postoji implicitno prebacivanje iz tekstualnog režima editovanja u strukturni režim editovanja. Moguće je postojanje više od jedne selekcije u editoru kada je on u tekstualnom režimu. Pravila editovanja definišu koja jezička struktura može biti editovana u kom režimu (tekstualnom, strukturnom ili oba).

Specifikacija jezika

Synthesizer Generator je zasnovan na *attribute grammars*. Korisnik sistema piše specifikaciju programskog jezika na *Synthesizer Specification Language* (*SSL*) jeziku. U prvoj verziji *SSL* jezika, za pisanje funkcija za izračunavanje atributa (*attribute evaluation*) korišćen je *C* programski jezik. Sadašnja verzija generatora podržava definisanje *semantic equations and functions* upotrebom *SSL* jezika.

Dizajner editora (*editor designer*) priprema *attribute grammar* specifikaciju koja sadrži pravila sa opisom apstraktne sintakse, dodele atributa (*attribution*), formata prikaza (*display format*) i konkretne sintakse. U editoru koji se izgeneriše iz ove specifikacije, program je reprezentovan drvetom, čiji čvorovi sadrže atribute. Inkrementalna analiza se sprovodi ažuriranjem vrednosti atributa kroz drvo, kao odgovor na modifikaciju.

Primena

Osim što se koristi za generisanje editora za programske jezike, *Synthesizer Generator* je iskorišćen za pravljenje nekoliko drugih alata koji rukuju strukturnim podacima. Primeri tih alata su: kalkulator

(*desk calculator*), *proof checker* i nekoliko editora za formatiranje teksta (*text-formatting editor*).

2.3.4 *Programming System Generator (PSG)*

PSG (*Programming System Generator*) je sistem koji generiše programska okruženja.

Editor

Editori u izgenerisanim okruženjima su hibridnog tipa. Prebacivanja iz strukturnog u tekstualni režim su implicitna, ali obrnut smer prebacivanja je eksplicitan. Može postojati više od jedne selektovane strukture u tekstualnom režimu.

Specifikacija jezika

Specifikacija jezika se sastoji iz 3 dela. U prvom delu se definiše sintaksa jezika. Ovaj deo sadrži:

- opis leksičke strukture (definicija ključnih reči i delimitera),
- opis apstraktne sintakse (definicija apstraktnog sintaksnog drveta, koje će služiti kao interna reprezentacija fragmenata programa),
- opis konkretne sintakse (definicija transformacija pojedinih fragmenata programa iz tekstualne reprezentacije u apstraktno sintaksno drvo); Pošto *PSG* sistem koristi silazno parsiranje, konkretna sintaksa je ograničena na *LL(1)* gramatike,
- opis formata prikaza (definicija gramatike koja se koristi za transformaciju drveta u string, odnosno za konstruisanje eksterne tekstualne reprezentacije apstraktnog drveta; ovaj deo uključuje i *pretty-printing* informacije kao što su znak za novi red i indentacija podstruktura).

U drugom delu se definišu uslovi za kontekstnu analizu jezika (*context conditions*). Ova analiza treba da ispuni nekoliko zahteva:

- mora biti u stanju da analizira nekompletne fragmente programskog teksta
- mora garantovati momentalnu detekciju semantičkih grešaka čak i u nekompletiranim fragmetima
- mora biti efikasna u interaktivnim programskim okruženjima, pa mora raditi u inkrementalnom režimu
- pošto je *PSG* generator, kontekstna analiza mora biti izgenerisana iz (drugog dela) formalne specifikacije.

Fragment programa je ispravan ako je on ispravan program ili ako može biti ubačen u ispravan program kao njegov deo. U svrhu kontekstne analize, koriste se atributi. Međutim, u nezavršenim fragmentima, ne postoji jedinstven način dodele vrednosti atributa čvorovima drveta, zato što je moguće da nedostaju neke važne informacije. Zbog ovog defekta, *PSG* sistem umesto vrednosti atributa koristi skup mogućih vrednosti atributa. Umesto da se koristi nekoliko atributa za jedan čvor drveta, koristi se najviše jedan atribut za svaki čvor, ali on može biti struktuiran. Skup dodele atributa se tada može posmatrati kao relacija (*context relation*). Ako je fragment završen i ispravan, relacija će sadržati tačno jednu n-torku (*tuple*) jer postoji samo jedna moguća dodela atributa za kompletirane programe. U slučaju semantičke greške, relacija će postati prazna, jer ne postoji ni jedna ispravna dodela vrednosti atributa čvoru.

Skup svih vrednosti atributa je opisan apstraktnom sintaksom koja se zove *data attribute grammar* (DAG). U njoj se definišu strukture atributa koje se koriste u datom jeziku. DAG definiše projektant jezika (*language definer*).

U trećem delu se definiše dinamička semantika upotrebom *denotational semantics*. Korisnik mora da definiše semantičke funkcije (*semantic function*) za svaku sintaksnu konstrukciju, tako što će da definiše značenje te konstrukcije unutar semantičkog domena (*semantic domain*), u zavisnosti od značenja svojih podkomponenti. Semantičke funkcije se pišu na funkcionalnom jeziku (*functional language*).

PSG definicije jezika su sigurne (*safe*), jer se sve nekonzistentnosti detektuju u vreme generisanja. S obzirom na kompleksnost i mogućnosti izgenerisanih okruženja, *PSG* specifikacija jezika je

relativno kratka (240 linija za opis *ALGOL 60* okruženja, 3600 linija za opis *MODULA-2* okruženja).

Istorijat

Rad na *PSG* sistemu započeo je 1980. godine. Prototip (*BLKS* sistem) je bio operativan krajem 1981. Kompletan *PSG* sistem, implementiran na *Pascal* programskom jeziku i *Siemens BS2000* računarima, je završen 1983. Kasnije su iskorišćene grafičke mogućnosti personalnih računara, pa je korisnički interfejs kompletno redizajniran, a *PSG* sistem u celini je prilagođen radu na *UNIX* sistemima.

Iskustvo sa *PSG* je pokazalo da se svi delovi okruženja koji zavise od jezika mogu formalno opisati i automatski izgenerisati, bar za jezike čija kompleksnost nije veća od jezika kao što su *Pascal* ili *MODULA-2*.

2.3.5 *ASF + SDF Meta-environment*

ASF + SDF Meta-Environment (Algebraic Specification Formalism plus Syntax Definition Formalism) je interaktivno razvojno okruženje koje automatski generiše interaktivne sisteme za pravljenje specifikacije jezika i generisanje alata za te jezike. Ono što pravi razliku između ovog i ostalih sistema je činjenica da se isti editor koristi za editovanje definicija jezika i editovanje programa. Editor je hibridni.

Specifikacija jezika

Specifikacija jezika sadrži opis sintaksnih i semantičkih aspekata jezika [29, 93]. Može se koristiti za definisanje programskih jezika, jezika za pisanje specifikacije, jezika za upite nad bazom podataka ili za tekst procesiranje, itd. *ASF + SDF* formalizam nudi:

- formalizam algebarske transformacije (*algebraic specification formalism*)
- modularnost specifikacije jezika
- integraciju definicija leksičke, kontekstno nezavisne i apstraktne sintakse
- mogućnost da korisnik definiše sintaksu specifikacije, tj. da piše specifikaciju koristeći svoju notaciju

- objedinjavanje definicija sintakse i semantike.

Editor

U sistemu postoje dva editora: tekstualni i strukturni. Strukturni editor rukuje drvetom parsiranja, a ne rukuje leksičkim sadržajem čvorova u drvetu parsiranja. Tekstualni editor radi na nivou znakova, i time menja sadržaj čvora u drvetu. Ako editovani tekst nije sintaksno ispravan (što je neizbežno tokom editovanja) najmanje poddrvo koje sadrži neispravan fragment programa će biti zadržano u fokusu.

Ova dva editora su implementirana kao dva zasebna alata, a međusobno su povezana preko *ToolBus* skripta. Ovaj skript obezbeđuje jednoznačno mapiranje iz tekstualne forme u strukturnu i obrnuto. Strukturni editor je specificiran upotrebom *ASF + SDF* formalizma, a implementiran je upotrebom *Java* programskog jezika. Tekstualni editor koji se koristi u sistemu je *jedit*, javno dostupan tekstualni editor napisan na *Tcl/Tk* programskom jeziku.

Interpeter je ranije bio definisan pomoću *ASF + SDF* formalizma, a kasnije je na osnovu ove specifikacije napravljena implementacija upotrebom *C* programskog jezika.

Arhitektura

ASF + SDF Meta-Environment sistem se sastoji od nekoliko komponenti. Jedna od njih (*ToolBus*) koordinira radom preostalih. Ostale komponente su:

- korisnički interfejs (u osnovi, služi za prikazivanje grafa i za preuzimanje grafa specifikacije, *graph browser*)
- tekstualni editor (prilagođena verzija *XEmacs* editora za tekstualno editovanje)
- strukturni editor (sintaksno-vođeni editor, koji je u tesnoj vezi sa tekstualnim editorom)
- parser (*scannerless generalized LR parser, SGLR*, koji je parametrizovan tabelom parsiranja)
- generator tabele parsiranja (preuzima *SDF* definiciju sintakse i iz nje generiše tabelu parsiranja za *SGLR* parser)

- repozitorijum drveta (*tree repository*), sadrži reprezentaciju apstraktnog sintaksnog drveta i svih modula specifikacije; u ranijoj verziji sistema, apstraktno sintaksno drvo je bilo reprezentovano pomoću *VTP* formalizma (*Virtual Tree Processing formalism*), a novija verzija koristi alternativni mehanizam *AsFix* koji predstavlja specijalizaciju generičkog formata *ATerms, Annotated Term*).
- kompajler (kompajler koji prevodi *ASF* specifikaciju na *C* kod, *ASF2C*)
- interpreter (izvršava specifikacije direktnom interpretacijom)
- *unparser* generator (generiše *pretty-printer* alate).

Primena

Dosadašnja primena *ASF* + *SDF* formalizma se može podeliti u nekoliko grupa:

- za pisanje formalne specifikacije bilo kog problema za koju je potrebna interaktivna podrška
- u oblasti brzog razvoja jezika (*language prototyping*) je korišćen za opis sintakse i semantike namenskih jezika (*domain specific languages*) i pravljenje interaktivnog okruženja za te jezike
- za analizu ili transformisanje programa napisanih na nekom programskom jeziku; u oblasti *reverse engineering* i *system renovation* je korišćen za analizu i transformaciju nasleđenih COBOL programa (*COBOL legacy code*)
- kao formalizam za specifikaciju alata za rukovanje programskim kodom (*language processing tools*). Nekoliko komponenti samog *Meta-Environment* sistema je specificirano upotrebom *ASF* + *SDF* formalizma (*ASF2C* kompajler, *unparser* generator i neki delovi generatora tabele parsiranja).

Komponente sistema se mogu iskoristiti kao samostalni (*stand-alone*) alati (primer je *Stratego* kompajler [94]).

Specifikacija za strukturno editovanje je testirana samo na malim primerima. U radovima se navodi da će editor u budućim verzijama biti zamenjen *Emacs* editorom.

2.3.6 *SmartTools*

SmartTools je generator programskih okruženja (*semantic framework generator*). *SmartTools* framework je naslednik *Centaur* sistema, te stoga koristi apstraktnu sintaksnu definiciju (*AST*) kao strogo tipizirani (*strongly-typed)* formalizam za sve svoje alate. Iz *AST* specifikacije moguće je izgenerisati više drugih specifikacija odnosno generatora (parser, *pretty-printer*, konfiguracionu datoteku (*resource file*), *Data Type Definition, DTD,* ili *Schema* specifikaciju). Glavni cilj ovog generatora je da pomogne dizajnerima namenskih (*domain-specific*) jezika. U tom smislu, dovoljna je jedna specifikacija napisana u standardnom *DTD* ili *Schema* formatu, ili u internom *AST* obliku, da bi se brzo proizvelo namensko razvojno okruženje.

Editor

Strukturni editor namenjen nekom jeziku *L* se automatski generiše iz *AST* specifikacije koja opisuje jezik *L*. Upotrebom ovog strukturnog editora krajnji korisnik može editovati bilo koju *L* datoteku.

Korisnički interfejs

Grafički interfejs (*GUI*) je zasnovan na *document/view* konceptu. To znači da se korisnički interfejs može posmatrati kao okvir u kome se može raditi sa raznim 'pogledima' na dokument (odnosno *AST* drvo). Za svaki otvoren dokument, moguće je napraviti jedan ili više prikaza koji prikazuju različite aspekte drveta u skladu sa različitim formatima. Na primer, moguće je napraviti jedan prikaz tekstualne reprezentacije, jedan prikaz strukturnog editovanja i još jedan prikaz grafičke reprezentacije. Grafički interfejs je implementiran upotrebom *Java* i *XML* (*EXtensible Markup Language*) tehnologija, i ima mogućnost konfigurisanja. Pojedinačni prikaz dokumenta se pravi tako što se na *AST* drvo primeni *XSLT* transformacija (*XSL Transformation*) da bi se proizveo *BML* (*Bean Markup Language*) koji sadrži opis grafičkih komponenti koje treba napraviti. Iznad *XSLT* jezika definisan je jezik za transformaciju, višeg nivoa, koji se zove *Xpp*. Njegova namena je transformisanje *XML* dokumenta u neki drugi format (*XSLT*, *BML*, *HTML* ili tekstualni). Sličan je *XSLT* jeziku, i sastoji se od skupa pravila. Sa desne strane pravila se mogu koristiti i postojeće funkcije

za formatiranje (na primer: horizontalno i vertikalno poravnanje, indentacija).

Semantika

Semantička analiza je razvijena korišćenjem *Visitor Design Pattern* tehnike programiranja i korišćenjem aspekt-orijentisanog programiranja (*aspect-oriented programming*).

Arhitektura

SmartTools sistem se sastoji od nekoliko nezavisnih modula koji međusobno komuniciraju asinhronom razmenom poruka. Informacija koja se prenosi porukama ima *XML* format.

Primena

SmartTools sistem je korišćen za brzo razvijanje prototipa okruženja za nekoliko različitih namenskih (*domain-specific*) jezika. Napravljeno je razvojno okruženje za *Java* programski jezik, ali bez parsera, zbog velike kompleksnosti ovog jezika.

2.3.7 *Pan*

Pan je sistem za editovanje i pregledanje (*browse*). Nastao je iz istraživanja o tome na koje načine se može iskoristiti *language-based* tehnologija da bi ponudila integrisanu podršku za razvoj softvera i dokumenata sa kojima softver rukuje. Između korisnika i softvera se nalazi inteligentni interfejs koji prilikom editovanja nudi uopštene operacije tradicionalnih interaktivnih editora za podršku pregledanju (*browsing*), modifikaciji i radu sa dokumentima.

Editor

Editor je hibridni, dozvoljava tekstualno editovanje bez ograničenja, koristi inkrementalnu analizu na zahtev i obezbeđuje povratne informacije (*feedback*).

Implementacija

Između jednostavnog korisničkog modela koji podržava čisto tekstualno editovanje, s jedne strane i strukturne reprezentacije koja podržava strukturno editovanje, s druge strane, *Pan* je svoju implementaciju zasnovao na leksičkoj reprezentaciji koja se nalazi na sredini između ova dva pristupa.

Pan podržava tri kategorije korisnika: klijenti (*clients*), oni koji obavljaju podešavanje sistema (*customizers*) i autori specifikacije jezika (*language description authors*). Za klijente, *Pan* je interfejs za pregledanje i editovanje dokumenata. Klijenti koriste delove sistema i opis jezika, ali ne moraju puno da znaju o njima. Autori specifikacije jezika moraju poznavati sintaksnu strukturu i kontekstna ograničenja jezika, kao i tehnike koje *Pan* sistem koristi za analizu dokumenata.

Specifikacija jezika

Analiza dokumenata zasniva se na dve komponente: *Ladle* i *Colander*. *Ladle* obuhvata inkrementalnu leksičku i sintaksnu analizu; generiše tabele koje su zavisne od jezika (*language-specific tables*) i *run-time analyzer* koji revidira internu reprezentaciju dokumenta da bi odslikao tekstualne izmene. *Colander* rukuje specifikacijom i inkrementalnom proverom kontekstnih ograničenja. Postupkom analize koordinira interfejs za editovanje (*editing interface*) preko koga se korisnicima pružaju informacije.

2.3.8 Uporedni prikaz generatora programskih okruženja

U tabelama 2.3 i 2.4 su uporedno prikazana bitna svojstva opisanih generatora programskih okruženja.

	editor	alati
Mentor	hibridni	?
Centaur	strukturni	skener/parser (*LALR*), interpreter, dibager, *pretty-printer*, *type-checker*
Synthesizer Generator	hibridni	skener/parser (*LALR*), kompajler, interpreter, dibager, *pretty-printer*, inkrementalni *type-checker*
PSG	hibridni	skener/parser, interpreter, inkrementalni *type-checker*
ASF+SDF Meta-Environment	hibridni	skener/parser (*generalized LR*), interpreter, kompajler, dibager, generator parsera, *prettyprinter*, *type-checker*
SmartTools	strukturni	kompajler, interpreter, *XML* related tools
Pan	hibridni	*browser*

Tablica 2.3: Uporedni prikaz generatora programskih okruženja (? označava nedostupnost podataka)

	sintaksa	semantika	*unparsing* pravila
Mentor	*Metal*	*Mentol*	?
Centaur	*Metal*	*Typol* (*Natural Action*), *ASF* (*Structural operational*)	*PPML*
Synthesizer Generator	*SSL* (*Attribute Grammar*)	*SSL* (*Attribute Grammar*)	*unparsing schemes* (*Attribute Grammar*)
PSG	interni format	static semantic: context relations (*DAG*), dynamic semantic: Denotational	?
ASF+SDF Meta-Environment		*Algebraic Specification*	
SmartTools	*AST*, *DTD*, *Schema*	*Visitor Pattern Design*	?
Pan	*Ladle*	*Colander*	?

Tablica 2.4: Uporedni prikaz generatora programskih okruženja (nastavak)

2.4 Generička programska okruženja

Kao najznačajniji predstavnici generičkih programskih okruženja proučeni su: *Generic Syntax-directed Editor* (*GSE*) [41, 96], *ORM* Environment [53, 28, 47, 48], *Agora Structure Editor* (*ASE*) [32] i *Muir* [97].

2.4.1 *Generic Syntax-directed Editor* (*GSE*)

Generic Syntax-directed Editor (*GSE*) je razvijen u sklopu *Esprit* projekta "*Generation of Interactive Programming Environments*" (*GIPE*).

GSE je generički u smislu da je parametrizovan definicijom sintakse i opcionim skupom semantičkih alata.

Editor

Editor je hibridni, što znači da je moguće i tekstualno i strukturno editovanje. Editor je izgrađen na konceptu da sve operacije na programskom tekstu jesu tekstualne izmene, a da je ažuriranje interne reprezentacije drveta sporedni efekat (*side-effect*) te izmene. Posvećeni zagovornici čisto strukturnog editovanja, kao argument iznose činjenicu da u slučaju strukturnog editovanja ne postoji potreba za parserom. Kao kontra-argument, autori *GSE* editora iznose da kod hibridnog editovanja nema potrebe za *pretty-printer*-om. Prednosti oslanjanja na parser a ne na *pretty-printer* su: 1) ne postoji potreba za definicijom *pretty-printer*-a, 2) korisnik ne mora da podešava *pretty-printer* i 3) editor rukuje komentarima na predvidljiv način. Mane ovakvog pristupa su: 1) i tekst i apstraktno sintaksno drvo moraju biti smešteni u strukturi editora i 2) mora postojati dvosmerno mapiranje između drveta i delova teksta. Autori smatraju da su prednosti veće od mana. Održavanje dve interne reprezentacije zahteva prostor, ali ubrzava editor, i mapiranje može da se implementira efikasno.

Najvažniji koncept u modelu editovanja je fokus (*focus*). Fokus je par [deo_teksta, poddrvo], gde prvi element para 'deo_teksta' predstavlja deo teksta koji odgovara drugom elementu para 'poddrvo'. Tekstualno editovanje je dozvoljeno samo unutar fokusa. Kursor označava trenutnu poziciju unutar teksta. *GSE* garantuje da je sav tekst van fokusa sintaksno ispravan. Premeštanje fokusa na neki drugi deo teksta se radi pomoću navigacionih komandi ('*go to next child*', '*go to previous child*' ili '*go to parent*') ili pomoću miša. Svako pomeranje fokusa zahteva parsiranje teksta unutar fokusa. Ako se parsiranje uspešno završi, poddrvo koje odgovara fokusu se menja novim drvetom parsiranja. U suprotnom se postavlja pitanje da li dozvoliti pomeranje fokusa. Ispostavlja se da bi bilo suviše restriktivno odbiti zahtev za pomeranjem fokusa, jer bi korisnik tada morao da ispravi sve sintaksne greške pre nego što se može nastaviti editovanje. *GSE* je usvojio malo pristupačniju metodu: ako parsiranje prethodnog fokusa ne uspe, fokus se proširi na minimalno potreban deo teksta koji obuhvata i prethodni fokus i novi fokus. Ovo proširenje fokusa je automatsko. Mana ovakvog pristupa je tendencija rasta fokusa, jer se time povećava

vreme parsiranja fokusa.

GSE ima dva specifična režima, jedan za tekstualno i drugi za strukturno editovanje. Kada se jednom pređe iz strukturnog u tekstualni režim, strukturalna informacija je izgubljena. *GSE* editoru mogu biti pridruženi razni alati, kao što su *type-checker*, *pretty-printer* ili interpreter.

Interna reprezentacija je apstraktno sintaksno drvo.

Implementacija

Mapiranje između teksta i drveta obavlja parser. Implementacija je zasnovana na čuvanju informacija o poziciji strukture u čvorovima apstraktnog sintaksnog drveta (informacija se čuva u anotacijama). Editoru je potrebna informacija o početnim i završnim koordinatama teksta koji odgovara nekom poddrvetu. Ovakva informacija (koja se čuva u velikom broju čvorova) mora da se ažurira svaki put kada se tekst u fokusu promeni. U *GSE* editoru se zato čuvaju relativne pozicije početka teksta u odnosu na prethodni čvor ili roditeljski čvor i relativna pozicija kraja teksta u odnosu na njegov početak. Osnovna prednost ovakve reprezentacije je da oni čvorovi koji ne pripadaju fokusu ne menjaju ove informacije tokom editovanja.

Primena

GSE editor se koristi u *ASF + SDF Meta-Environment* sistemu za editovanje *SDF* specifikacija, ali i kao editor u izgenerisanom okruženju.

2.4.2 *ORM Environment*

ORM Environment je generičko programsko okruženje.

Korisnički interfejs

Korisnički interfejs je zasnovan na dva bitna principa: 1) direktna manipulacija (*direct manipulation*) i 2) prikaz konteksta pomoću hijerarhijskog sistema prozora (*hierarchical window system*). Ovi principi primenjeni su na interfejs prema svim bitnim objektima u sistemu: programima, izvršavanju programa i gramatici. Bitni objekti

su predstavljeni prozorima na ekranu. Korisnik može identifikovati prozor objektom koji on predstavlja. Ova identifikacija je osnovni preduslov za direktnu manipulaciju. Fizička struktura objekata odgovara strukturi prozora na ekranu. Svaki prozor ima svoj skup operacija koje su relevantne za taj objekat. Korisnik se, osim kroz sistem prozora, može kretati i kroz strukturne relacije. Primeri relacija su: *super/subclass* relacija između klasa ili *declare/use* relacija između identifikatora u programu. Navigacija kroz ovakve relacije je moguća pomoću linkova (*hyperlink*). Kada korisnik odabere link, ciljni prozor postaje vidljiv.

ORM okruženje implementirano za programski jezik *Simula* [28] pruža mogućnost kontekstnog editovanja. Ova vrsta editovanja je izgrađena oko menija *Names*. On obezbeđuje listu svih deklarisanih imena, vidljivih u fokusu editovanja. Neke od stavki u meniju označavaju procedure, nizove i klase. Čak i nedeklarisana imena koja su korištena negde u programskog tekstu, biće prisutna u meniju. Tako kada korisnik odluči da deklariše to ime, on može do njega da stigne preko menija. Ovim se dobija efekat da se imena unose samo jednom. Meni *Names* prikazuje sva vidljiva imena, bez obzira da li će njihovo ubacivanje u fokus editovanja izazvati *type-checking* grešku ili ne. Prvi nedostatak kontekstnog editovanja u *ORM* okruženju je činjenica da *Names* meni osim liste dostupnih imena, sadrži i razne druge informacije. Time se vreme stizanja do željene stavke (imena) produžuje. Takođe, sve informacije u tom meniju su strukturno predstavljene, pa se time gubi preglednost. I, na kraju, ne postoji mogućnost filtriranja imena na osnovu nekog kriterijuma (na primer: tip podatka datog imena).

Interna reprezentacija

Program se, interno, sastoji od blokova (na primer, klasa i procedura). Izvršavanje programa je sastavljeno od instanci blokova i aktivacija procedura. Blokovi programa su reprezentovani prozorima i ikonicama, i mogu se slobodno raspoređivati na ekranu. Svaki blok sadrži svoje lokalne informacije (*locals,* na primer, parametre i deklaracije promenljivih) i svoje telo (*body*, odnosno niz iskaza). Oni su predstavljeni kao tekst i edituju se na strukturni način. Tekst je dobijen transformacijom sintaksnog drveta (*unparsing*).

Semantika

Analiza statičke semantike se vrši inkrementalno dok korisnik edituje program. Omogućena je stalna kontrola semantičkih grešaka. Greške su neupadljivo označene i mogu se ispraviti u bilo kom trenutku. Objašnjenje greške se može dobiti preko komande menija.

I neispravni i nedovršeni programi se mogu izvršavati, tako što će izazvati *breakpoint* u tački programa gde se nalazi greška.

Specifikacija jezika

ORM je parametrizovan gramatikom jezika i on interpretira tu gramatiku dinamički. Posledica je da okruženje ne mora da bude ponovo izgenerisano da bi prihvatalo druge jezike ili da bi prihvatilo izmene gramatike jezika. Svaka izmena gramatike jezika odmah ima efekat na program.

Gramatike u *ORM* sistemu su predstavljene kao objekat (kao i sve ostalo u sistemu). Objekat gramatike se sastoji od četiri aspekta jezika, pri čemu se svaki definiše posebnim formalizmom.

- apstraktna gramatika (*abstract grammar*) definiše strukturu jezika. Opisana je upotrebom strukturne varijante proširene *BNF* notacije (*extended Backus Naur Form*) u kojoj su izostavljeni svi elementi konkrente sintakse

- konkretna gramatika (*concrete grammar*) definiše tekstualnu reprezentaciju za jezičke konstrukcije koje su definisane u apstraktnoj gramatici. Ovo uključuje opis *syntactical sugar* i format prikaza (*presentation layout*) za svaku jezičku konstrukciju

- semantička gramatika (*semantic grammar*) definiše statičku semantiku, kao što su pravila opsega važenja (*scope rules*), *identifier bindings* i provera tipova (*type checking*). Opisana je objektno-orijentisanom varijantom *attribute grammar*, dopuštajući da pravila, međusobno, nasleđuju atribute i jednačine

- gramatika koja opisuje generisanje koda (*code-generation grammar)* translira jezičke konstrukcije u međujezik (*intermediate language*) koji izražava dinamičku semantiku jezika. Opisana je objektno-orijentisanom varijantom *attribute grammar.*

Implementacija

Implementacija svih komponenti *ORM* okruženja je zasnovana na blok strukturi. Korisnički interfejs ovog okruženja je implementiran upotrebom *HOPE* interfejsa (*object-oriented programming interface to window system*). *HOPE* je interfejs opšte namene, napisan na *Simula* jeziku, i podržava *event-driven* komunikaciju.

Izvorni program se reprezentuje kao drvo izgrađeno od blokova (*block tree*). Svaki blok (*block object*) sadrži lokalne informacije kao što su tabela simbola, apstraktno sintaksno drvo za lokalne informacije (*locals*) i *telo* (*body*), lokalne informacije za generisanje koda, itd.

Komponenta sistema za prezentaciju (*presentation component*) je zadužena za prikaz sintaksog drveta. Na svaku izmenu sintaksnog drveta, ova komponenta inkrementalno ažurira tekstualnu reprezentaciju. Ovo je omogućeno interpretiranjem konkretne gramatike (koja sadrži format prikaza za svaku jezičku konstrukciju).

Program se smešta u bazu programa (*program database*) na bazi bloka. Individualna komponenta bloka, kao što je tabela simbola ili apstraktno sintaktno drvo za *locals*, se može dobiti iz baze programa pomoću jedne operacije. Baza ima strukturu drveta.

ORM okruženje je implementirano na *Simula* jeziku i izvršava se na *SUN* radnim stanicama (*SUN workstations*).

Primena

ORM okruženje je korišćeno u edukativne svrhe, na kursevima *compiler construction*.

2.4.3 *Agora Structure Editor* (*ASE*)

Agora Structure Editor je generički strukturni editor. To znači da je nezavisan od programskog jezika. Za opis gramatike jezika koristi se *structured grammar* formalizam. *ASE*, tokom procesa editovanja, interpretira pravila gramatike. Zbog toga, dovoljno je da se *ASE* editoru saopšti (*plug-in*) proizvoljna *structured grammar* gramatika, da bi se on mogao koristiti za pisanje programa u tom programskom jeziku.

Editor

ASE je hibridni editor. Njegova strukturna komponenta predviđa osnovne operacije za sintezu programa:

- *replace*, zamenjuje proizvoljnu označenu strukturu, strukturom odabranom iz skupa alternativnih struktura
- *expand*, zamenjuje nepopunjenu strukturu (*placeholder*) templejtom (*template*), a *unexpand* (*shrink*) zamenjuje templejt nepopunjenom strukturom
- *insert* (*before/after*), ubacuje templejt (proizvoljni fragment programa) odabran iz skupa alternativnih struktura
- *remove*, uklanja templejt ili nepopunjenu strukturu.

ASE editor, za razliku od prethodnih sličnih editora, nudi i operaciju strukturne transformacije (*structural transformation*). Ova operacija zamenjuje jedan fragment programa drugim, transformišući izvorni fragment ciljnim fragmentom. Strukturna transformacija predstavlja vezu između dva templejta, izvornog i ciljnog. Izvorni opisuje na koji fragment programa se transformacija može primeniti, a ciljni opisuje kako izvorni templejt izgleda posle transformacije. Editorska komanda strukturne transformacije nudi meni koji sadrži sve strukturne transformacije koje su moguće za trenutno označenu strukturu. Skup strukturnih transformacija nije ugrađen u editor (*hard-coded*), već ga korisnik može menjati (interaktivno, kroz poseban interfejs). *ASE* editor nudi i operaciju *drag&drop*, koja omogućuje zamenu (*replace*) i ubacivanje novih fragmenata programa (*insert*), a suštinski predstavlja *cut/copy* i *paste* operacije. Ova operacija (*drag*) se može izvršiti sa grafičke palete u programski tekst, ali ne i unutar samog programskog teksta.

Programi su interno reprezentovani kao apstraktno sintaksno drvo, koje se proširuje nepopunjenim strukturama.

Specifikacija jezika

Definicija programskog jezika opisuje se strukturnom gramatikom (*structured grammar*). Specifikacija sadrži opis: alfabeta (identifikatori i literali), *surface syntax* (rezervisane reči i znaci punktuacije),

skup neterminalnih simbola, početni neterminalni simbol i pravila gramatike. Postoji nekoliko različitih vsta pravila (*alternation rules*, *construction rules*, *repetition rules*, *list rules* i *lexeme rules*).

Primena

Postoji nekoliko prototipova *ASE* editora. Prvi prototip je 1992. godine implementiran na *Pascal* programskom jeziku. Podržava sve navedene operacije, osim strukturne transformacije i strukturne operacije *search & replace*. Drugi prototip je implementiran upotrebom *VisualWorks\Smalltalk* jezika. Integrisan je u programsko okruženje za *Agora* programski jezik. Manji tehnički problemi koji postoje su vezani za implementaciju *drag&drop* operacije, hibridne aspekte editovanja i mogućnosti pretrage (*search*).

2.4.4 *Muir*

Muir je okruženje za razvoj jezika. Namenjen je za kreiranje i razvoj jezika kao što su programski jezici, jezici za specifikacije i gramatički formalizmi, pa se zato naziva *language development environment* (*LDE*).

Specifikacija jezika

Muir okruženje rukuje strukturama koje su opisane u specifikaciji jezika. Ove strukture su organizovane oko apstraktnog sintaksnog drveta (*AST*). Čvorovima ovog drveta dodeljuju se osobine (*property*). Osobine mogu biti definisane u deklaraciji (*property declarations*) koja se vezuje za određeni čvor u specifikaciji jezika, ili se njihove vrednosti dobijaju na osnovu izračunavanja atributa (*attribute equations*) koja su navedena u specifikaciji jezika. Specifikacija jezika dozvoljava definisanje skupa atributa (*attribute set*) koji opisuje skup imena osobina i skup jednačina pomoću kojih se računaju njihove vrednosti.

Strukture se mapiraju u vizuelnu reprezentaciju (ekran ili papir) na osnovu formata prikaza (*presentation schemes*) koji je naveden u specifikaciji jezika. Jedan jezik može sadržati nekoliko različitih formata prikaza (tekstualni, grafički).

Parser

Muir je strukturno okruženje, stoga parser ima sasvim drugačiju ulogu od one koju ima u uobičajenim okruženjima. U strukturnim okruženjuma, *unparser* (koji koristi formate prikaza definisane u specifikaciji jezika) se poziva svaki put kada struktura treba da se prikaže, dok se parser ne poziva. U strogo strukturnom editoru, jedina stvar kojom se ne rukuje preko strukturnih operacija je unos simbola koji odgovaraju terminalnim čvorovima drveta (na primer, idektifikatora). Međutim, za ovo nije potreban kompletan parser, već samo algoritam prepoznavanja ovih simbola.

Interaktivno okruženje

Muir nudi interaktivno okruženje koje se izvršava na *Xerox* (serija 1100) radnim stanicama. Okruženje se naslanja na *Interlisp-D* okruženje, i u velikoj meri nasleđuje njegove karakteristike (način zauzimanja prostora na ekranu, komande za selekciju realizovane preko *pop-up* menija i prikaz drveta). Strukturni editor je zasnovan na tekst editoru *TEDIT*.

Strukturni editor

Akcije korisnika u radu sa strukturama su selekcija podstrukture i aktivacija editorske operacije. Editorska operacija se može aktivirati iz menija ili preko komande sa tastature. Meni prikazuje skup alternativnih operacija koje se mogu primeniti na selektovanu strukturu.

Editorske operacije su implementirane upotrebom uopštenog formalizma transformacije. Ovaj formalizam je namenjen translaciji programa sa jednog jezika na drugi, ili sa ranijih verzija istog jezika na novije verzije. Implementacija operacija transformacije je zasnovana na *pattern-match* i *replace* funkcijama.

Manipulacija atributima je realizovana kao poseban alat. Izračunavanja atributa se vrše na zahtev, što dozvoljava vrednostima atributa nekog čvora da zavise i od vrednosti svojih naslednika i od svojih roditelja. Postoji lista atributa čije vrednosti zavise od vrednosti koje još nisu dostupne, i svaki put kada je atribut izračunat, proverava

se da li u ovoj listi ima vrednosti koje se sad mogu izračunati. Ovo nije najefikasnije rešenje, ali zato nudi maksimalnu fleksibilnost.

Istorijat

Muir okruženje je započeto kao deo razvoja *Aleph* sistema za specifikaciju jezika. Dosad je najčešće korišćena specifikacija samog jezika za specifikaciju. Ona je razvijena *bootstraping* procesom. Na kraju procesa, dobijena je *MetaGrammar* specifikacija. Neke od osobina ove specifikacije su da se pravila koja opisuju format prikaza mogu definisati kombinacijom dijagrama i tekstualne forme, i da se nudi poseban mehanizam za definisanje liste i elmenata drveta.

2.4.5 Uporedni prikaz generičkih programskih okruženja

U tabeli 2.5 su uporedno prikazana bitna svojstva opisanih generičkih programskih okruženja.

	editor	alati	specifikacija jezika
GSE	hibridni	?	*ASF + SDF*
ORM	hibridni	?	interni format (koristi *EBNF*)
ASE	hibridni	?	*structured grammar*
Muir	hibridni	parser	interni format

Tablica 2.5: Uporedni prikaz generičkih programskih okruženja (? označava nedostupnost podataka)

2.5 Uporedni prikaz okruženja

Na osnovu prethodno izloženih podataka, zanimljivo je pogledati i pregled svih okruženja na osnovu alata koji se generišu iz sistema (u slučaju generatora okruženja) i alata koje poseduju (u slučaju generičkih i namenskih okruženja).

	editor	skener/ parser	kompajler	interpreter
Cornell Program Synthesizer	hibridni			
ABC structure editor	hibridni			
Mentor	hibridni			
Centaur	strukturni	*		*
Synthesizer Generator	hibridni		*	*
PSG	hibridni			*
ASF+SDF Meta-Environment	hibridni	*	*	*
SmartTools	strukturni		*	*
Pan	hibridni			
GSE	hibridni			
ORM	hibridni			
ASE	hibridni			
Muir	hibridni	*		

Tablica 2.6: Uporedni prikaz okruženja na osnovu sadržanih alata

	dibager	*pretty-printer*	*type-checker*	ostalo
Cornell Program Synthesizer	*			execution
ABC structure editor				
Mentor				
Centaur	*	*	*	
Synthesizer Generator	*			
PSG				
ASF+SDF Meta-Environment	*	*	*	generator parsera
SmartTools				
Pan				browser
GSE		iz parsera		
ORM		komponenta za prikaz		
ASE				
Muir		poseban *unparser*		

Tablica 2.7: Uporedni prikaz okruženja na osnovu sadržanih alata(nastavak)

2.6 Ostali alati

Postoji veliki broj alata, okruženja i generatora okruženja koji se bave temom razvoja programskih jezika, pravljenja njihovih prototipova, generisanjem okruženja za brzi razvoj prototipa jezika i sličnih namena. Neki od tih alata, koji nisu navedeni u prethodnom tekstu su:

- *Aloe* generator programskih okruženja (*Gandalf* projekat) [96]
- *GNOME* okruženje, familija strukturnih editora (*Gandalf* projekat) [23]

- *Gem-Mex*, generator programskih okruženja [5]
- *MUPE-2*, integrisano programsko okruženje za programski jezik *Modula-2* [46]
- *APPLAB* (*A Laboratory for Application Languages*), okruženje za interaktivni razvoj domenskih jezika, koristi se u istraživanju o programiranju industrijskih robota [11, 12]
- *UQ** razvojno okruženje [37]
- *ParaScope editor*, *intelilgent interactive editor for parallel Fortran programs (ParaScope project)* [38]

Ovi alati nisu detaljno analizirani jer nisu bliski postavljenim ciljevima istraživanja doktorske teze.

2.7 Opisi jezika

Formalna specifikacija programskih jezika može da sadrži opis leksičke i sintaksne strukture jezika, i opis (statičke i dinamičke) semantike jezika [17]. Pomoću opisa leksičke i sintaksne komponente programskog jezika korisnik se upoznaje sa strukturom programskog jezika i mogućim kombinacijama sintaksnih elemenata. Na osnovu ovog opisa, on može napraviti (ispravan) program. Značenje programa, opisuje se semantikom programskog jezika.

U nekim jezički-zasnovanim sistemima, formalni opis jezika sadrži i opis korisničkog interfejsa konkretnog sistema (okruženja, editora). Taj deo specifikacije uglavnom sadrži definiciju komandi u sintaksno-vođenom interfejsu, pravila za transformaciju struktura ili mogućnosti različitih prikaza istog drveta (*views*).

2.7.1 Definicija sintakse

Opisom sintakse se definiše interna reprezentacija programa, kao i njegova spoljašnja prezentacija. Interna reprezentacija je obično apstraktno sintaksno drvo, dok je spoljašnja prezentacija obično tekst. Iz praktičnih razloga, potrebno je i mapiranje između ove dve reprezentacije. Mapiranje iz spoljašnje prezentacije u internu reprezentaciju se koristi u svrhe parsiranja (*parsing*). Obrnuti

smer mapiranja se koristi u svrhe prikazivanja programa, i zove se *unparsing*. Ova mapiranja se mogu i izvesti iz datih gramatika.

Leksička komponenta savremenih programskih jezika se sastoji od kjučnih reči jezika i strukturnih elemenata kao što su stringovi, komentari i identifikatori. Alati, kao što je *Lex* [45, 61], obično traže spisak opisa leksičkih elemenata programskog jezika i odgovarajućih tokena. Iz ovog opisa oni generišu skener.

Sintaksni elementi se obično definišu pomoću alata kao što je *Yacc* [36, 19]. Oni za definiciju sintaksne strukture jezika koriste *BNF* formu ili neku njenu varijantu [35]. Iz ove definicije oni generišu parser. Sintaksa programskog jezika je intenzivno proučavana tokom 1960-tih i 1970-tih, i *BNF* forma je prihvaćena kao standardna metoda za definiciju sintakse. Ona je lako razumljiva i vrlo je pogodna za definiciju sintakse programskih jezika.

2.7.2 Definicija semantike

Opisom semantike obično se definišu semantička pravila jezika (na primer: provera tipova, mehanizam nasleđivanja). Ovaj opis može da sadrži i opis semantike okruženja (na primer: stil kodiranja (*coding style*)).

Za definciju semantike postoji nekoliko tehnika. Međutim, ni za jednu se ne može reći da je u opštem slučaju bolja od drugih, iako svaka ima svoje prednosti u određenom kontekstu.

Metode za definiciju semantike se grubo mogu podeliti u nekoliko grupa [66]: *operational semantics*, *denotational semantics*, *axiomatic semantics* i *attribute grammar*.

2.7.2.1 *Operational semantics*

Značenje ispravnog programa je trag tokom procesa računanja, koji rezultuje iz procesiranja programskog ulaza. *Operational semantics* [66] se zove još i *intensional semantics*, jer je sekvenca internih koraka računanja (*intensions*) najvažnija. Na primer, dva različito napisana programa, koji računaju faktorijel, imaju različitu *operational semantics*.

2.7.2.2 *Denotational semantics*

Denotational Semantics [66] opisuje semantiku programskog jezika manipulišući strukturnim modelom. Model se sastoji od skupa mapiranja. Značenje ispravnog programa je matematička funkcija, od ulaznih podataka do izlaznih podataka. Koraci koji su potrebni da se izračuna izlaz nisu važni, važna je relacija između ulaza i izlaza. *Denotational semantics* se zove još i *extensional semantics*, jer je važna jedino vidljiva relacija između ulaza i izlaza (*extension*). Tako, dva različito napisana programa, koji računaju faktorijel, imaju istu *denotational semantics*. Ova tehnika je pogodna za neke domene primene, ali zahteva visok nivo matematičke sofisticiranosti za čitanje i pisanje specifikacije. Nedostatak *denotational semantics* je njena zavisnost od funkcija kojima se opisuju sve forme izračunavanja. Kao posledica toga, *denotational semantics* velikog jezika je često vrlo teško čitljiva, a suviše niskog nivoa da bi se modifikovala.

Action semantics [54, 56, 88] je varijanta *denotational semantics*, koja je lako čitljiva. Ona ispravlja problem čitljivosti upotrebom familije standardnih operatora da bi opisala standardne forme izračunavanja u standardnim jezicima.

2.7.2.3 *Natural semantics*

Natural Semantics [21] predstavlja metodu potvrđivanja pravila izvođenja (*inference rules*) koja definišu semantiku programskog jezika. Ova pravila se mogu koristiti za dokazivanje korektnosti programa ili za simboličko izvršavanje programa (kao u slučaju *Centaur* sistema [13]).

Natural semantics se nalazi između *operational semantics* i *denotational semantics*. Kao *structural operational semantics*, ona prikazuje kontekst u kom se pojavljuju koraci izračunavanja, a kao *denotational semantics*, naglašava da je izračunavanje fraze sastavljeno od izračunavanja podfraza. *Natural semantics* je skup *inference rules*, a kompletno izračunavanje u *natural semantics* je jedna, velika derivacija. Ove tehnike su moćne i značajne u odgovarajućem kontekstu, ali nisu dovoljno jednostavne.

2.7.2.4 *Axiomatic semantics*

Značenje ispravnog programa je logička tvrdnja (*logical proposition*) koja iznosi neko svojstvo ulaza i izlaza. *Axiomatic semantics* proizvodi osobine (*property*) pre nego značenje. Izvođenje ovih osobina se radi pomoću *inference rule set* koji izgleda slično kao u *natural semantics*.

2.7.2.5 *Attribute Grammar*

Attribute Grammar [68, 78] opisuje sintaksu programskog jezika, zajedno sa statičkom semantikom. *Attribute grammar* je u osnovi kontekstno-nezavisna gramatika koja opisuje sintaksne elemente jezika, a proširena je informacijom o semantici. Opis semantike ima formu atributa (*attribute*). Svaki simbol kontekstno-nezavisne gramatike ima skup njemu pridruženih atributa. Vrednosti atributa su definisane pravilima za izračunavanje atributa (*attribute evaluation rules*) koja su pridružena pravilima kontekstno-nezavisne gramatike. Ova pravila, obično definišu vrednost određenog atributa kao funkciju koja zavisi od vrednosti nekog drugog atributa unutar pravila. Takođe, postoji i skup atributskih tvrdnji (*attribute assertion*) koje su povezane sa pravilom koje ograničava opseg validnih vrednosti atributa unutar pravila. Da bi neki program bio semantički ispravan, sve ove tvrdnje moraju biti tačne.

Programi opisani upotrebom *attribute grammar* se parsiraju i dobija se drvo parsiranja sa atributima (*attributed tree*). Svaki čvor ovog drveta će korespondirati pravilu u *attribute grammar*, i biće označen skupom atributa (dodeljenih pravilu). Atributima se dodeljuje vrednost prolaskom kroz drvo i izvršavanjem pravila za izračunavanje atributa, koja su vezana za pravilo, u svakom čvoru. Ove vrednosti se zatim proveravaju pomoću atributskih tvrdnji, da bi se osigurala semantička ispravnost programa.

Vrednostima atributa u čvoru drveta se može pristupiti kroz vrednost atributa jednog od čvorova potomaka, pri čemu se omogućava tok informacije nagore, kroz drvo, prema korenu drveta. Ova vrsta atributa se naziva sintetizovani atribut (*synthesized attribute*). Vrednosti atributa se takođe mogu propagirati i nadole, kroz drvo, od čvora roditelja, ka njegovim potomcima. Ovakav atribut se naziva nasleđeni atribut (*inhereted attribute*). Obično, simbol u gramatici će imati i sintetizovane i nasleđene atribute. Sintetizovani atribut dodeljen

čvoru sadrži informaciju koja opisuje poddrvo tog čvora. Nasleđeni atributi se koriste da bi izrazili zavisnost jezičkih konstrukcija od konteksta u kom se pojavljuju.

Attribute gramar je jednostavna za razumevanje i ne zahteva puno vremena za obuku. Osoba koja piše specifikaciju jezika je oslobođena potrebe da specificira sintaksne komponente i mapiranje iz apstraktne sintakse u konkretnu sintaksu, kao što je to slučaj kod nekih formalizama (kao što je *denotational semantics*). Ovaj formalizam je našao primenu u mnogim sistemima [17, 31, 65, 22, 62, 15, 95].

Attribute grammar pristup nije adekvatan za opis dinamičke semantike programskog jezika.

U pristupe za definiciju semantike svrstavaju se i *abstract state machines* (ranije poznate kao *evolving algebra*), *coalgebra semantics* i *program algebra* [30]. Postoje i drugi pristupi i alati: *algebraic specification* [84], *Montages specification* [43, 4], *Casl* [55], [50].

Poglavlje 3

Cilj istraživanja

Jezički-zasnovani sistemi za editovanje su interaktivni alati. Njihov cilj je da specijalizacijom tehnologije programskih jezika povećaju produktivnost korisnika (programera) [80]. Da bi postigli ovaj cilj, okruženja (sistemi) moraju ponuditi širok spektar editorskih operacija i usluga, vrlo kvalitetnu vizualizaciju, odnosno grafički prikaz i korisna i praktična podešavanja (*customizability*) [89, 92].

Činjenica je da programeri čitaju programe tekstualno, ali i da, takođe imaju strukturan način razumevanja programa. Sa druge strane, većina programera ima duboko ukorenjene navike editovanja i nisu voljni da ih menjaju. Oni će prihvatiti samo onaj alat koji im nudi naprednije operacije editovanja i njihovo udobno i intuitivno korišćenje u smislu da ne moraju da se specijalno obučavaju za njihovu upotrebu. Napredna funkcionalnost editorskih operacija neće biti odbačena ili ignorisana ako se one pažljivo dizajniraju, tako da ponude nikakva, ili jedva primetna, ograničenja [27]. Specijalizovana unapređenja operacija editovanja su bitna, ali je još bitnije da ona ne umanje značaj i efikasnost postojećih osobina. Bilo kakva nefleksibilnost editora odbija korisnike. Zato je neophodno pronaći pravi balans između udobnosti korisnika i doslednosti alata.

Efikasan jezički-zasnovan sistem mora pružati mogućnost podešavanja, kako bi se prilagodio individualnim korisnicima. Jezički-zasnovan sistem mora imati vizuelni stil koji se lako podešava za različite jezike i zadatke.

Jezički-zasnovani sistemi za editovanje se sastoje od alata kao

što su editor, analizator, generator, kompajler, dibager. Ovi alati mogu biti efikasno integrisani, ako međusobno dele informacije. Svi oni koriste strukturnu reprezentaciju programskog koda (apstraktno sintaksno drvo). Ovakav sistem mora biti sposoban da koristi mnoštvo informacija, dobijenih iz više različitih alata u okruženju. Međutim, pošto editor predstavlja alat, koji prikazuje i dozvoljava modifikaciju programskog koda u obliku teksta, njegova implementacija je najizazovnija.

Izveštaj *SIGCHI* grupe o strukturnim editorima [58] je dao pregled strukturnih editora iz perspektive interakcije čovek-računar (*human-computer interaction*). Što se tiče izbora dizajna editora, zaključeno je da ne treba ograničavati korisnika *top-down* pristupom konstrukcije programa, već da treba da bude podržan *bottom-up* pristup razvoju programa. Izveštaj o greškama ne sme biti nametljiv, i bilo bi dobro da koristi boje za efektivniji prikaz informacije. Učesnici su postigli koncenzus da strukturni editori nude vrlo korisne usluge. Da bi ih korisnici lakše prihvatili, potrebno je da novi strukturni editori budu laki za učenje i korišćenje, i da ponude one usluge koje će motivisati korisnike da ih koriste. Strukturni editori su često bili usmeravani prema programerima početnicima [49]. Međutim, pošto početnici ne ostaju uvek početnici, bitno je da novi strukturni editori ponude i spektar usluga namenjenih iskusnim programerima. Strukturni editor za početnike bi se mogao posmatrati kao podskup usluga strukturnog editora za eksperta. Strukturni editori bi mogli da budu i mnogo više od editora, ako bi podržali razvoj projekata i razvoj sistema. Editor bi mogao ponuditi usluge kao što su kontrola verzija (*version control*) i *software configuration management*.

Pregled u poglavlju 2 pokazuje da su strukturni editori predmet dugotrajnog proučavanja. Ipak, trenutno ne postoji ni jedan strukturni editor koji je u širokoj praktičnoj upotrebi. U savremenoj upotrebi postoje programska okruženja sa tekstualnim editorom koja nude korisniku neke operacije sa strukturama, ali u rudimentarnom obliku (na primer: *Eclipse* [99], *VS .NET C# editor* [98]).

3.1 Postavljeni ciljevi

Ciljevi kojima je vođeno ovo istraživanje i proizveden početni prototip (koji će biti razmotren u sledećem poglavlju) se mogu svrstati u nekoliko

kategorija:

Oživeti ideju strukturnog editovanja Iako strukturni editori nisu doživeli široku komercijalnu upotrebu, njihove prednosti su neosporne. Mane ovih editora mogu se locirati u oblasti njihovog korisničnog interfejsa, a ne editorskih operacija koje nude. Savremena komercijalna okruženja, koja svoj repertoar editorskih operacija proširuju strukturnim, su bitan pokazatelj da korisnici imaju potrebu za takvim operacijama.

Intuitivni korisnički interfejs Mane strukturnih editora se ogledaju u načinu interakcije sa korisnikom, i mogle bi se izbeći izmenom korisničkog interfejsa. Jedna od bitnih prednosti strukturnih editora jeste da korisnik ne mora detaljno da poznaje sintaksu programskog jezika. Ova osobina je bitno uticala na odluku da prototip bude prevashodno namenjen programerima početnicima i neprofesionalnim programerima.

Generičko okruženje Cilj uopštavanja je prirodna posledica toka istraživanja (videti 1.8). Prvo je nastao *STeditor* (samo za programski jezik *Structure Text*) a zatim generator takvih editora. Postavljeni cilj je da se napravi generičko okruženje koje će ponuditi strukturno editovanje programa pisanih različitim programskim jezicima (za koje postoji formalna specifikacija). Za početak, potrebno je napraviti prototip okruženja namenjenog klasi proceduralnih programskih jezika (*C*, *Pascal*, *Modula*, ...).

Kvalitetna vizualizacija Komercijalna okruženja prate savremene trendove razvoja i softvera i hardvera, pa time postavljaju nove granice u kvalitetu usluga koje nudi editor. Kvalitet, u smislu grafike, zahteva upotrebu interaktivnih elemenata (dijalozi, meniji), vizuelnih elemenata (različiti jezički elementi se vizuelno prikazuju različitim fontovima ili u različitim prozorima) i upotrebljivost proizvoda (u smislu brzine i memorijskih zahteva). Prototip bi trebao da prati savremene trendove u vizualizaciji programskog koda.

Proširiva arhitektura sistema Arhitekturu sistema treba postaviti tako da se kasnije lako može proširiti: a) da podrži druge klase programskih jezika (ne primer, objektno orijentisane), odnosno njihove osobine sintakse i semantike, b) da se editoru mogu lako

dodati nove operacije i c) da se korisničkom interfejsu mogu lako dodati nove komande.

***Compile-time* analize** Strukturni editor treba da podržava sintaksu i statičku semantiku programskog jezika, odnosno *compile-time* analize. Pošto neće biti podržana *run-time* funkcionalnost, ne postoji potreba za pravljenjem kompajlera, interpretera, dibagera ili alata za izvršavanje.

3.2 Izbor dizajna editora

Jednostavni tekstualni editori nude editovanje, bez jezičke podrške. Strukturne informacije, ako ih ima, u ovim editorima su nekompletne i neprecizne.

Sintaksni editori koriste tekstualnu reprezentaciju, i podržavaju prepoznavanje nekih jezičkih konstrukcija. Ovakvi editori nude neke od strukturnih operacija editovanja, kao što su indentacija i *highlighting*.

Strukturni editori koriste strukturnu reprezentaciju (drvo). Ova struktura je pogodna za realizaciju raznovrsnih strukturnih operacija editovanja, ali s druge strane, zahteva od programera da edituje programski tekst preko strukturnih (a ne preko tekstualnih) komandi. Korisnik je uvek u direktnoj interakciji sa programskim strukturama i zato ne mora detaljno da poznaje sintaksu programskog jezika. Strukturni editori nude jedino operacije nad strukturnim elementima programa i ne dozvoljavaju korisniku da napravi sintaksno neispravan program. Ovo je moguće zbog pristupa da program nije samo običan niz znakova, već hijerarhijska struktura. Struktura programa određena je formalnom strukturom programskog jezika. To znači da strukturni editor u toku svog rada modifikuje strukturu drveta koja odgovara konkretnom programu, i da pri tom konsultuje formalni opis strukture tog programskog jezika. Prvi strukturni editori koji su ponuđeni korisnicima nisu dobro prihvaćeni jer su nametali *top-down* pristup unosa programskog teksta. Korisnici su ovo smatrali neprirodnim načinom unosa, nezgodnim kada su u pitanju izrazi.

Hibridni editori su nastali kao odgovor na primedbe prvobitnim strukturnim editorima. Ovakvi editori podržavaju i strukturne i nestrukturne operacije. Korisnik može raditi u strukturnom ili

tekstualnom režimu. Dok korisnik radi u tekstualnom režimu, okruženje uz pomoć inkrementalnih tehnika parsiranja, pravi strukturu programa. Prebacivanje iz tekstualnog režima u strukturni, i obrnuto, je u nekim editorima eksplicitno, a u nekim implicitno.

Izbor u dizajnu editora treba da bude takav da predstavlja ravnotežu između želje da editor sadrži jezički ispravnu reprezentaciju programa, i želje da editor ponudi pristupačne, napredne, jezički-vođene operacije editovanja, da bi postigao povećanje produktivnosti korisnika.

U nastavku teksta sledi uporedni prikaz postojećih editora. Tabela 3.1 nudi pregled editora na osnovu njihovih operacija, a tabela 3.2 sumira prednosti i mane razmatranih editora.

vrsta editora	strukturne operacije	tekstualne operacije
tekstualni	ne	da
sintaksni	indentacija, *highlighting*	da
strukturni	da	ne
hibridni	da	da

Tablica 3.1: Pregled editora na osnovu operacija koje nude

vrsta editora	prednosti	mane
tekstualni	jednostavna interna reprezentacija	nepostojanje strukturnih operacija
sintaksni	jednostavna interna reprezentacija, delimično prepoznavanje strukture	nepostojanje strukturnih operacija
strukturni	postojanje strukturnih operacija	nefleksibilan korisnički interfejs
hibridni	mogućnost i tekstualnog i strukturnog unosa teksta	postojanje režima (modova) rada

Tablica 3.2: Pregled editora

Posle proučavanja postojećih vrsta editora, i u skladu sa zadatim ciljevima, izbor je pao na čisto strukturni editor. Strukturno editovanje je poslednjih nekoliko decenija privlačilo pažnju istraživača. To pokazuju i nova komercijalna okruženja koja su prepoznala prednosti strukturnog editovanja. Ova okruženja ([98, 99]) su namenski implementirana za jedan programski jezik, pa time nude samo one strukturne operacije koje su korisne i primenljive za taj programski jezik. Ipak, širenjem skupa svojih tekstualnih i nestrukturnih operacija strukturnim operacijama, ova okruženja podržavaju ideju strukturnih editora. Samim tim, podržavaju i strukturni način razmišljanja programera.

3.3 Izbor korisničkog interfejsa editora

Svi strukturni editori podrazumevaju postojanje operacija koja omogućavaju editovanje na nivou struktura. Međutim, korisnički interfejs koji podržava te operacije, se razlikuje od editora do editora. Najraniji strukturni editori su koristili samo tastaturu, odnosno strukturne operacije su mogle da se pokrenu samo pomoću tastature.

Kasnija pojava grafičkih terminala i pokazivačkih uređaja (*pointing devices*) uvela je izmene korisničkog interfejsa, pa su strukturne operacije editora bile dostupne preko menija i miša. Značajnim razvojem hardvera, razvile su se i grafičke mogućnosti računara, pa je i vizuelni prikaz programskog teksta postao bolji.

Većina strukturnih editora predstavljenih u prethodnom poglavlju je razvijena 80-tih godina. U poslednjih deset godina razvijen je samo jedan alat [7]. To znači da većina postojećih strukturnih editora ne pruža grafičke i vizuelne mogućnosti, koje nudi sadašnji softver i hardver.

Proučavanje korisničkog interfejsa postojećih strukturnih editora je bilo otežano iz nekoliko razloga: (a) implementacija editora nije javna, (b) implementacija nije izvršiva na *Windows* i *Linux* platformama i (c) mnogi dokumenti opisuju dizajn editora, ali ne opisuju korisnički interfejs. Istraživanje svakog od predstavljenih sistema izvršeno je na osnovu nekoliko radova koji opisuju sistem. Iz istraživanja je vrlo jasno zaključeno da dizajn postojećih strukturno orijentisanih editora ostavlja mesta za poboljšanja. Cilj ovog istraživanja je da se drugačijim pristupom u oblikovanju korisničkog interfejsa izbegnu mane postojećih strukturnih editora.

Strukturno-orijentisana okruženja su obično namenjena programerima početnicima. U njima je naglašena podrška za konstruisanje sintaksno ispravnih programa i postojanje povratne informacije o *compile-time* greškama tokom editovanja. Međutim, ukoliko iskoriste ažurne kontekstno senzitivne informacije, ova okruženja imaju veliki potencijal da podrže napredno editovanje, koje može biti vrlo korisno i iskusnim programerima.

Kontekstno editovanje je implementirano u nekim pomenutim jezički-zasnovanim sistemima [28], ali i u sistemima drugačije namene (kao što je tekst procesor *Microsoft Office 2007* [59]). Ovakvo editovanje podrazumeva da se korisniku u svakom koraku editovanja, na jednostavan i prihvatljiv način, nude informacije koje su mu u tom trenutku potrebne [81, 90]. Da bi se realizovalo kontekstno editovanje, potrebne su informacije i o sintaksi i o statičkoj semantici jezika.

Korisnički interfejs editora mora biti (maksimalno) usklađen sa načinom na koji korisnici razmišljaju. Komande i alati moraju biti lako dostupni, onda kada su korisniku potrebni. Do ovog zaključka su došli i dizajneri novog *Microsoft Office 2007* tekst procesora [57], nakon temeljnih istraživanja i testiranja. *Microsoft Office Word 1.0* je

imao samo oko 100 komandi, a *Word 2003* ima više od 1500 komandi, pa je jasno da je sistem menija postao opterećen. Time je postalo komplikovano i iritirajuće pronaći neku komandu koja se baš i ne koristi tako često. Novi interfejs *Office 2007* paketa sam nudi komande i sam pretpostavlja koje su komande korisniku potrebne i kada su mu potrebne. Takođe, postoje i kontekstne komande koje su dostupne samo onda kada korisnik radi sa nekim posebnim objektom (tabelom, fotografijom i slično). Ovo je još jedan dokaz da korisnički interfejs mora biti pažljivo osmišljen i dizajniran tako da prati korisnikov način razmišljanja.

Zato bi strukturni editor trebao da sledi intuitivni pristup [52]. Trebao bi da zadrži sve osobine tradicionalnih strukturnih editora koje su korisniku od pomoći. To bi se moglo postići posredstvom fleksibilnih sintaksno-orijentisanih operacija za editovanje programskog teksta. Korisnički interfejs bi trebao da bude jednoobrazan da bi omogućio isti tretman svih jezičkih struktura (onda bi se izrazi mogli editovati jednako jednostavno kao i iskazi).

Strukturni editor, odnosno njegov korisnički interfejs, ne bi trebao da ima posebno strukturni i posebno tekstualni režim editovanja. Tekstualno editovanje bi se moglo integrisati sa strukturnim editovanjem, jer je moguće koristiti ga samo za unos leksičkih struktura (imena promenljivih i literala).

Posledice upotrebe ovakvog korisničkog interfejsa editora su:

1. nije potrebno detaljno poznavanje sintakse programskog jezika
2. eliminisanje sintaksnih grešaka i grešaka statičke semantike
3. značajno povećanje produktivnosti programera.

Prve dve posledice su važnije programerima početnicima i neprofesionalnim programerima, a treća je važna i profesionalnim programerima.

3.3.1 Izbor strukturnih operacija

Operacije koje strukturni editor može ponuditi korisniku, mogu se podeliti u dve kategorije: osnovne i dodatne (izvedene, napredne). Osnovne strukturne operacije se koriste za sintezu programa. One obuhvataju sledeće operacije:

navigacija pozicioniranje (kretanje kroz strukturu) i selektovanje strukture, pomoću miša ili tastature

ubacivanje **(*pre/posle*)** dodavanje nove strukture, pre ili posle selektovane strukture

modifikacija izmena selektovane strukture

brisanje uklanjanje selektovane strukture

Dodatne strukturne operacije su one koje proširuju mogućnosti strukturnog editora. One su specifične za svaku implementaciju strukturnog editora. U ove operacije spadaju:

automatska indentacija

sintaksni *highlighting*

rukovanje komentarima (retko se nalazi u strukturnim editorima)

proširenje selekcije na primer, 'uključi u selekciju i narednu/prethodnu strukturu'

semantička navigacija na primer, 'označi mesto deklaracije imena'

semantički upit na primer, 'označi sva mesta upotrebe imena'

semantička promena imena izmena deklaracije imena se istovremeno odslikava i na svim mestima njegove upotrebe

višestruka selekcija mogućnost istovremenog postojanja više selektovanih struktura

cut/paste

copy/paste

strukturna transformacija (*refactoring*).

Strukturni editor bi trebao da podrži sve tradicionalne strukturne operacije. Ove operacije obezbeđuju disciplinovan režim sinteze proramskog teksta i time garantuju kontinualni strukturni integritet programa. Time je obezbeđena nemogućnost pojave sintaksnih grešaka u toku editovanja programskog teksta. Istovremeno, ovim operacijama

postiže se i usmeravanje, odnosno vođenje korisnika, tokom procesa editovanja.

Osim uobičajenih operacija navigacije, koje nude pomeranje kursora samo sa jedne strukture na drugu, od velike je važnosti ponuditi korisniku i dodatne navigacione operacije kao što su:

- pomeranje kursora sa znaka na znak unutar jedne strukture
- pomeranje kursora sa jedne linije teksta na drugu.

Ovim operacijama se postiže intuitivnost u kretanju kroz strukturu programa. Takođe, ovakva navigacija kursora podseća na navigaciju u tradicionalnim tekstualnim editorima, što je velika prednost u interfejsu ka korisnicima.

Strukturni editor koji želi da ponudi korisniku punu udobnost editovanja, trebao bi u skupu svojih operacija da sadrži i operaciju strukturne transformacije. Operacija strukturne transformacije (*structural transformation*) zamenjuje jedan fragment programa drugim. Odnosno, strukturna transformacija predstavlja vezu između dva templejta, izvornog i ciljnog. Izvorni opisuje na koji fragment programa se transformacija može primeniti, a ciljni opisuje kako izvorni templejt izgleda posle transformacije. Editorska komanda strukturne transformacije nudi meni koji sadrži sve strukturne transformacije koje su moguće za trenutno označenu strukturu. Skup strukturnih transformacija ne bi trebao biti ugrađen u editor (*hard-coded*), da bi ga korisnik mogao menjati.

3.4 Izbor interne reprezentacije

Za punu realizaciju željenog sistema, interna reprezentacija programa treba da bude kombinacija strukture drveta i strukture grafa. Tačnije, najbolja reprezentacija bi bila usmereno drvo (*ordered tree*). Usmerenost je neophodna za realizaciju redosleda susednih čvorova. Ovaj redosled odslikava sintaksnu strukturu jezika.

Čvorovi drveta bi mogli da se implementiraju kao strukture (alocirane na *heap*-u), sa pokazivačem na roditelja, pokazivačem na svog prvog potomka i pokazivačima na svog prethonika i svog sledbenika (*leftmost-child right-sibling representation*).

Autor bi hteo na ovom mestu da naglasi činjenicu, da je cilj teze bila opštost i što brža izrada prototipa, radi njegovog testiranja i daljeg razvoja, a ne efikasnost implementacije. To, naravno, ne znači da je implementacija neefikasna. Efikasnost implementacije direktno utiče na brzinu odziva editora, koja predstavlja vrlo važan faktor za ocenu njegovog kvaliteta.

3.5 Izbor formalnog opisa specifikacije

Da bi strukturni editor podržao editovanje u nekom programskom jeziku, neophodno je da sadrži potpunu definiciju programskog jezika. Poželjno je da ova definicija bude jednostavna za čitanje i pisanje [17].

Specifikacija programskog jezika bi trebala da ima oblik tekstualne datoteke. Sadržaj formalnog opisa jezika bi trebao da sadrži:

- leksičku strukturu jezika (definiciju ključnih reči i delimitera)
- sintaksnu strukturu jezika (pravila koja opisuju izgled programa)
- statičku semantiku (definiciju pravila opsega važenja, provere tipova, ...)
- format prikaza za jezičke konstrukcije (ovaj deo uključuje *pretty-printing* informacije kao što su znak za novi red i indentacija podstruktura)
- podatke o vezi između strukture jezika i načina njenog vizuelnog prikazivanja u okviru okruženja (koja jezička struktura opisuje jednu kompilacionu jedinicu, putanje do biblioteka, ...)

Opis formalne specifikacije jezika bi trebao da obezbedi da:

- specifikacija bude relativno kratka
- specifikacija bude čitljiva
- specifikacija integriše definicije sintakse i semantike
- se sve nekonzistentnosti u specifikaciji detektuju u vreme parsiranja specifikacije

- ne opterećuje korisnika detaljima implementacije (organizacija tabele simbola i sl.)

Na osnovu pregleda postojećih formalizama za opis sintakse i semantike (programskih) jezika, odlučeno je da se ne koristi ni jedan od postojećih, već da se definiše novi formalizam, koji odgovara konkretnim potrebama izrade prototipa okruženja. Postojeći formalizmi su uglavnom preopširni i uopšteni, i zahtevaju od korisnika koji piše specifikaciju puno vremena za njihovo savladavanje. Novi formalizam bi trebao da bude intuitivan, i da (za početak) pokrije potrebe proceduralnih programskih jezika.

3.6 Izbor arhitekture okruženja

Arhitektura sistema odslikava strukturu postavljenih ciljeva, pa sistem treba da sadrži nekoliko modula.

Ponašanje editora u generičkom okruženju zavisi od specifikacije programskog jezika. Zato, mora postojati modul koji prima specifikaciju programskog jezika i parsira njen sadržaj. Specifikacioni parser bi trebao dobijene podatke negde trajno da pohrani. U tu svrhu, pogodno je koristiti strukturu tabele. Specifikacione tabele bi trebale da sadrže opis leksičke i sintaksne strukture jezika, statičku semantiku, pravila prikaza struktura i dodatne informacije iz specifikacije. Znači, potreban je i modul koji rukuje specifikacionim tabelama.

Podatke iz ovih tabela koristi poseban modul namenjen rukovanju internom reprezentacijom programskog teksta. Svaki put kada treba izmeniti strukturu editovanog teksta, ovaj modul mora konsultovati opis sintakse i semantike zapisane u specifikacionim tabelama, ali i tabelu simbola, u kojoj se nalaze sve informacije o svim simbolima u editovanom programu. Ovaj modul implicitno mora sadržati nekoliko alata: interpreter sintakse i statičke semantike programskog jezika, skener, *pretty-printer* i *import/export* alate.

Na kraju, usluge modula koji modifikuje internu reprezentaciju koristi zaseban modul interfejsa, koji je u direktnoj komunikaciji sa korisnikom. Ovaj modul treba da ima dvostruku komunikaciju sa korisnikom. S jedne strane, treba da preuzima editorske komande od korisnika, a s druge strane korisniku treba da prikazuje rezultate

obrade tih zahteva. U prvom slučaju, modul interfejsa treba da preuzme zahtev od korisnika i da traži od modula koji rukuje internom reprezentacijom da izvrši editorsku operaciju. Posle završene akcije, nazad se šalju podaci koji predstavljaju rezultat izvršene akcije. Ovi podaci mogu predstavljati niz znakova koje treba prikazati, ili informaciju o interakciji u koju treba ući sa korisnikom (na primer: otvaranje dijaloga, ili prijavljivanje grešaka).

3.7 Razlike u odnosu na postojeće sisteme

Predloženi pristup se na sledeći način razlikuje od pristupa u proučenim sistemima slične namene:

- generičko okruženje sadrži čisto strukturni editor, dok proučena generička okruženja koriste hibridne editore
- strukturni editor nudi korisnički interfejs koji sledi intuitivni pristup editovanju programskog teksta, koji najviše liči (podseća) na korisnički interfejs tradicionalnih tekstualnih editora. Ovakav interfejs je naročito zanimljiv programerima početnicima i neprofesionalnim programerima
- formalizam za opis specifikacije jezika je nov (moglo bi se reći da se za pisanje specifikacije koristi kombinacija *BNF*, *DTD* i *XML* načina formalnog opisa)
- arhitektura sistema sadrži modul koji vrši istovremeno interpretiranje svih aspekata (osobina) gramatike: i leksičke i sintaksne i semantičke, uz poštovanje pravila o formatima prikaza
- interna reprezentacija je usmereno drvo

Poglavlje 4

Opis rešenja

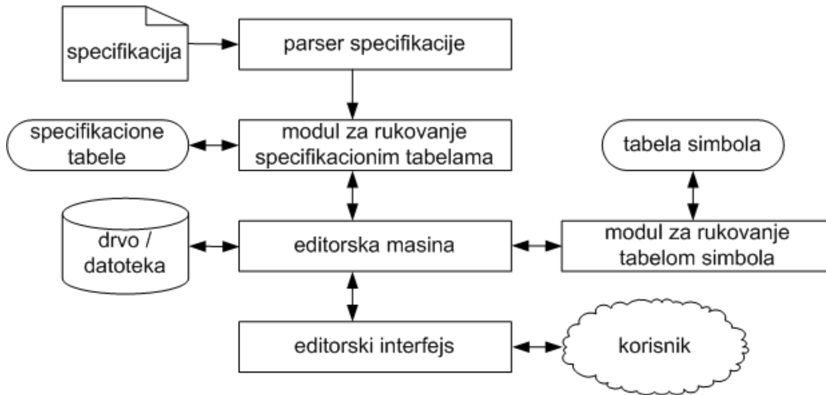
Prototip generičkog okruženja sa strukturnim editorom je nazvan *Univerzalni Strukturni Editor* (*USE*). Njegova implementacija je rezultat višegodišnjeg istraživanja. Jedan deo prototipa zasniva se na implementaciji programskog okruženja za programski jezik *Structure Text* (*Strukturni sintaksno-vođeni editor za ST* [71]).

Prototip je napravljen i testiran za programski jezik *C*. Specifikacija ovog jezika (sintaksa i semantika), je preuzeta iz knjige *The C Programming Language, second edition, Brian W. Kernighan, Dennis M. Ritchie* [39]. Ona sadrži "*American Standard for Information Systems - Programming Language C, X3.159-1989*" standard, iz 1988. godine.

Za implementaciju sistema korišćen je programski jezik *C#*, razvojno okruženje *Microsoft Visual Studio 2005* i *Microsoft .NET Framework*. Prototip se može izvršavati na platformama koje poseduju *Microsoft .NET Framework*. Uz pomoć postojećih alata za konverziju programa iz *C#* u *Java* programski jezik, i dodatnu konverziju grafičkog interfejsa u skladu sa *Java* tehnologijom, ovaj prototip relativno jednostavno postaje izvršiv na drugim platformama.

4.1 Arhitektura sistema

Model sistema se sastoji od pet modula (slika 4.1).



Slika 4.1: Model sistema

Specifikacija programskog jezika sadrži formalni opis programskog jezika. Nju preuzima specifikacioni parser, parsira je i prevodi u interni format. Ove informacije se smeštaju u specifikacione tabele posredstvom modula za rukovanje specifikacionim tabelama.

Editorska mašina obezbeđuje interpretiranje pravila programskog jezika i prevodi ih u informacije koje su potrebne editorskom interfejsu. Ona konsultuje specifikacione tabele i tabelu simbola. Editorska mašina pri tome koristi modul za rukrukovanje tabelom simbola i modul za rukovanje specifikacionim tabelama.

Editorski interfejs predstavlja peti modul koji ima ulogu posrednika između editorske mašine i korisnika. On prima zahtev od korisnika i prosleđuje ga editorskoj mašini, a zatim preuzima rezultat obrade zahteva i prikazuje ga korisniku.

Svaki od modula implementiran je u posebnom *C# namespace* modulu.

4.2 Specifikacija programskog jezika

Formalna specifikacija programskog jezika ima oblik tekstualne datoteke. Moglo bi se reći da se za pisanje specifikacije koristi kombinacija *BNF*, *DTD* i *XML* načina formalnog opisa.

Na početku ove datoteke se definiše ime programskog jezika koji se opisuje u specifikaciji, kao na primer:

```
%language name="AnsiC"
```

U početnom delu specifikacije se opisuje leksička struktura jezika, jer se ona koristi kasnije, prilikom opisa sintaksne strukture. Opis leksičke strukture se sastoji od definicija simbola, ključnih reči, delimitera, selektora (*member selector*) i definicije komentara.

Nakon toga se opisuju regularni izrazi, a zatim osnovni tipovi podataka koje podržava dati programski jezik.

U nastavku se opisuje sintaksa programskog jezika. Sintaksna struktura jezika se opisuje sintaksnim pravilima. Uz svako sintaksno pravilo moguće je vezati opis statičke semantike tog pravila ili format prikaza te strukture na ekranu. Definicije statičke semantike opisuju opseg važenja identifikatora, pravila prenošenja tipova sa strukture na strukturu, mesta uvođenja novih imena, načine korišćenja imena na određenim mestima i slične informacije. Format prikaza sintaksnog elementa uključuje *pretty-printing* informacije kao što su znak za novi red i indentacija podstruktura.

Problemi koji su se javili prilikom definisanja strukture specifikacije su bili vrlo izazovni i vremenski zahtevni. Razlog tome je potreba za jakom integracijom leksičke, sintaksne i semantičke strukture jezika. Editorska mašina treba da interpretira pravila, koja moraju da uključuju i opis sintakse, i opis statičke semantike, i opis leksičkog sadržaja strukture i format njenog prikaza.

Specifikacija programskog jezika mora da sadrži opis gramatike koja je tipa *LL* [3]. Ova potreba proistekla je iz načina rada editorskog interfejsa. Različita sintaksna pravila, ponekad, započinju istim ključnim rečima. Kako se ove ključne reči koriste u interakciji sa korisnikom, kada korisnik odabere jednu ključnu reč, javlja se dvosmislenost u odabiru sintaksnog pravila. Zato, svaki sintaksni simbol mora imati jedinstven početni skup simbola (*FIRST*). Iz istog razloga, za pisanje gramatike se ne bi smela koristiti tipična leva rekurzija. Međutim, u specifikaciji je predviđena posebna oznaka (#) kojom se označava da određeno pravilo (u toku interpretiranja) treba da se tretira kao da je opisano levom rekurzijom. Korišćenje desne rekurzije nema posledica na interpretaciju sintaksnih pravila, pa se može koristiti na uobičajen način.

4.2.1 Opis leksičke strukture jezika

Opis leksičke strukture programskog jezika sadrži definiciju simbola. Definicija simbola započinje ključnom rečju *%symbol*. Svaki simbol se obavezno opisuje imenom (*name*) i stringom (*string*), a opciono može sadržati atribute (*attr*). Atributi imaju oblik stringova sa unapred definisanim značenjem. Na ovaj način mogu se definisati: ključne reči, separatori ili selektora.

U nastavku slede primeri definicije simbola: ključna reč *if*, separator zapeta (zarez), selektor tačka koji sme da se primenjuje na objekte tipa strukture ili unije i simbol tačka-zapeta (koji ne sadrži atribute).

```
%symbol name="IF"
        string="if"
        attr="keyword"

%symbol name="COMMA"
        string=","
        attr="separator"

%symbol name="DOT"
        string="."
        attr="member_selector(Structure, Union)"

%symbol name="SEMICOLON"
        string=";"
```

Opis komentara započinje navođenjem ključne reči *%comment*. Dalje se mogu definisati string kojim se označava početak komentara u programskom tekstu (*begin*), opcioni string kojim se označava kraj komentara u programskom tekstu (*end*) i opcioni podatak (*nested*) o tome da li komentari ovakve vrste mogu biti ugnježđeni ili ne. Ukoliko se ne navede vrednost za *end*, podrazumeva se znak za novi red, a ukoliko se ne navede vrednost atributa *nested*, podrazumeva se vrednost *false*. U nastavku je naveden primer opisa (višelinijskog) komentara koji ne sme biti ugnježđen:

```
%comment begin="/*" end="*/" nested="false"
```

4.2.2 Opis osnovnih tipova podataka jezika

U svrhu definisanja literala za određene tipove podataka, potrebno je definisati neke regularne izraze. Definicija regularnog izraza u specifikaciji započinje ključnom rečju *%regexp*, a dalje se navodi ime regularnog izraza (*name*) i njegova definicija (*exp*). Za definiciju nekog regularnog izraza može se koristiti ime prethodno definisanog regularnog izraza, tako što se navede u vitičastim zagradama. U nastavku sledi primer (delimične) definicije literala za tip podatka *int*:

```
%regexp name="digit"
        exp="[0-9]"
%regexp name="decimal_int"
        exp="{digit}+"
...
%regexp name="INT_LITERAL"
        exp="{decimal_int}|{octal_int}|{hexa_int}"
```

Definicija osnovnog tipa podatka započinje ključnom rečju *%data_type*. Za svaki tip, obavezno se mora definisati ime (*name*), njegov string (*string*) i regularni izraz koji opisuje literal (*literal*), a opciono se može definisati i skup drugih tipova podataka u koje se ovaj tip može konvertovati (*conversion_to*). U nastavku sledi primer opisa celobrojnog tipa *int*.

```
%data_type name="INT"
           string="int"
           literal="int_literal"
           conversion_to="FLOAT DOUBLE"
```

Nakon definicije svih tipova podataka, mogu se definisati i grupe tipova. Njihova svrha je lakši, pregledniji i kraći opis operatora programskog jezika. Opis jedne grupe započinje ključnom rečju *%data_type_group* i obavezno sadrži ime grupe (*name*) i članove grupe (*members*). Članovi se navode razdvojeni praznim mestom, a član (osim osnovnog tipa) može biti i neka grupa tipova. U nastavku slede primeri opisa grupe tipova podataka: opis grupe *INTEGRAL_TYPES* koja sadrži tipove *CHAR* i *INT*, potom opis grupe *FLOATING_TYPES* koja sadrži tipove *FLOAT* i *DOUBLE*, a onda opis grupe *ARITHMETIC_TYPES* koja sadrži tipove iz dve prethodno definisane grupe (a to su *CHAR*, *INT*, *FLOAT* i *DOUBLE*).

```
%data_type_group name="INTEGRAL_TYPES"
                 members="CHAR INT"
%data_type_group name="FLOATING_TYPES"
                 members="FLOAT DOUBLE"
%data_type_group name="ARITHMETIC_TYPES"
                 members="INTEGRAL_TYPES FLOATING_TYPES"
```

4.2.3 Opis operacija i operatora jezika

Operacija se definiše za određeni tip podatka. Razlikuju se definicije unarnih i binarnih operacija jezika. Definicija unarne operacije započinje ključnom rečju *%unary_operation*. Za unarnu operaciju, obavezno se mora definisati tip podatka na koji se primenjuje. U nastavku se opisuju n-torke koje se navode u malim zagradama. U njima se definišu string jednog ili više operatora razdvojenih praznim mestom (*operator*), tip rezultata operacije (*result*) i opciono, da li rezultat predstavlja *lvalue* referencu ili ne. Vrednost za *result* može biti ime nekog tipa podatka (ili grupe) ili jedan od predefinisaih stringova (*promoted_type(operands)*, *promoted_type(operand1)*, *type(operand1)*, *LOGICAL*). Podrazumevana vrednost za *lvalue* je *false*. U nastavku sledi primer opisa unarne operacije koja se primenjuje na tip *ARITHMETIC_TYPES*:

```
%unary_operation operand="ARITHMETIC_TYPES"
                 ( operator="+ -"
                   result="promoted_type(operands)" )
                 ( operator="!"
                   result="INT" )
```

Definicija binarne operacije započinje ključnom rečju *%binary_operation*. Ostatak definicije je isti kao za unarnu operaciju, s tom razlikom da n-torka mora sadržati još i definiciju tipa podatka za drugi operand (*operand*). U nastavku sledi primer opisa binarne operacije koja se primenjuje na tip *ARITHMETIC_TYPES*:

```
%binary_operation operand="ARITHMETIC_TYPES"
                  ( operator="+ - * /"
                    operand="ARITHMETIC_TYPES"
                    result="promoted_type(operands)" )
```

```
                         ( operator="< > <= >= == != && ||"
                           operand="ARITHMETIC_TYPES"
                           result="LOGICAL" )
```

Specifikacioni parser automatski kompletira podatke o binarnim operacijama koji važe kada tipovi operanada zamene mesta.

Definicija operacije dodele započinje ključnom rečju *%assignment_operation*. Za operaciju dodele treba definisati tip leve strane dodele (*left*) i string jednog ili više operatora razdvojenih praznim mestom (*operator*). Podrazumeva se da leva strana iskaza dodele mora sadržati *lvalue* referencu. U nastavku sledi primer opisa operacije dodele (predefinisana konstanta *ANY_TYPE* označava da tip leve strane iskaza dodele može biti bilo koji tip):

```
%assignment_operation left="ANY_TYPE"
                      operator="= *="
```

4.2.4 Opis imena

Identifikatori se mogu razvrstati u nekoliko kategorija (*namespace*), koje se međusobno ne mešaju. To znači da isti identifikator može biti korišćen u različite svrhe, čak i u istom opsegu važenja (*scope*), ako je njegova upotreba unutar različitih *namespace* kategorija.

Nova kategorija imena se definiše tako što se iza ključne reči *%namespace* navede ime kategorije. U nastavku slede tri primera opisa kategorije imena:

```
%namespace OBJECT
%namespace FUNCTION
%namespace FIELD
```

4.2.4.1 *lvalue* referenca

Specifikacija nudi ključnu reč *%lvalue* kojom korisnik može definisati uslove pod kojima neki izraz predstavlja *lvalue* referencu [39]. Objekat je imenovani region memorije, a *lvalue* je izraz koji referencira objekat. *lvalue* reference su bitne za definisanje operatora. Neki operatori prihvataju *lvalue* operande, a neki kao rezultat daju *lvalue* izraze. U nastavku sledi primer definicije *lvalue* reference (programski jezik *C*):

```
%lvalue namespace="OBJECT"
        type="ARITHMETIC_TYPES | Structure
             | Union | Pointer"
```

4.2.5 Opis sintaksne strukture jezika

Sintaksna struktura jezika se opisuje sintaksnim pravilima. Svako sintaksno pravilo ima oblik:

```
ime_simbola
    :   telo
    ;
```

Znak ":" razdvaja pravilo na levu i desnu stranu. Na levoj strani se nalazi ime simbola, a na desnoj njegova definicija. Desna strana može sadržati sekvencu simbola kojom se ime sa leve strane može zameniti. Sa stanovišta interne reprezentacije, desna strana opisuje poddrvo čvora kojim je predstavljen simbol sa leve strane. Kraj pravila se označava znakom ";".

Ime simbola može biti proizvoljne dužine, sastavljeno od slova, brojeva i podcrte "_". Ista imena napisana malim ili velikim slovima se razlikuju. Može postojati samo jedno pravilo u kojem se sa leve strane pojavljuje dato ime.

U telu pravila se mogu koristiti imena leksičkih i sintaksnih simbola. Od specijalnih operatora, sa desne strane pravila, mogu se koristiti operator za opis alternative (znak "|") i kvantifikatori.

Kvantifikator se (u prvoj verziji prototipa) može primeniti samo na pojedinačni simbol. Postoji nekoliko kvantifikatora:

+ izraz se može pojaviti jednom ili više puta

***** izraz se može pojaviti nijednom ili više puta

? izraz se može pojaviti nijednom ili jednom.

Već je naglašeno da se ne sme koristiti leva rekurzija prilikom definisanja sintaksnih pravila. Međutim, u praksi se javlja realna potreba za takvom reprezentacijom struktura. U tu svrhu, specifikacija nudi kvantifikator "#", koji opisuje ponavljanje nijednom, ili više puta, izraza na koji se primenjuje, ali uz specijalnu interpretaciju datog pravila. Editorska mašina dato pravilo (koje je označeno ovim

kvantifikatorom) obrađuje kao da je definisano pomoću leve rekurzije. To znači da i poddrvo, koje se kreira na osnovu ovog pravila, odslikava levu rekurziju. U nastavku sledi primer upotrebe ovog kvantifikatora, u pravilu koje opisuje binarni izraz:

```
binary_expression
   :   expression (BINARY_OPERATOR expression)#
   ;
```

Editorska mašina će svaki deo poddrveta, koji se sastoji od celokupne desne strane pravila (dve pojave pojma *expression*), uvek graditi kao poddrvo koje pripada simbolu (čvoru) *binary_expression*, bez obzira koliko ukupnih pojava pojma *expression* postoji.

4.2.5.1 Akcije

Svakom pravilu se može dodeliti akcija, koja će biti izvršena u toku editovanja tog dela poddrveta. Sve akcije zajedno se navode posle tela pravila, u vitičastim zagradama.

```
ime_simbola
    :   telo
            {  opis_akcija  }
    ;
```

Akcija sadrži opis statičke semantike (uključujući proveru tipova) i opis formata prikaza. Za potrebe akcija, specifikacija nudi specijalne operatore. Svaka akcija se završava znakom `";"`. Akcija se izvršava svaki put kada se edituje dati deo programskog koda, odnosno kad se modifikuje odgovarajući deo poddrveta.

4.2.6 Operatori

Operatori se koriste za opis statičke semantike jezika i opis formata prikaza strukture na ekranu. Njihova upotreba je dozvoljena jedino unutar opisa akcija.

4.2.6.1 Opis statičke semantike jezika

Za opis pojedinačnih osobina statičke semantike, predviđeni su zasebni operatori.

Operator *%start_symbol* se navodi u akciji onog simbola koji predstavlja startni simbol gramatike (i u isto vreme i koren drveta). Ukoliko se ovaj operator ne navede ni u jednom pravilu, za startni simbol gramatike se uzima simbol čije pravilo je prvo navedeno.

Operator *%compilation_unit* se navodi u akciji onog simbola koji predstavlja celinu koja odgovara jednoj datoteci. Ukoliko se ovaj operator ne navede ni u jednom pravilu, uzima se simbol čije pravilo je prvo navedeno.

Operator *%new_scope* uvodi novi opseg važenja (*scope*). Za svaki opseg važenja imena, postoji po jedna heš (*hash*) tabela preko koje se imena ubacuju u tabelu simbola, ili se tabela simbola pretražuje.

Operator *%new_type(...)* definiše novi tip podatka. Time je spremna za upotrebu (inicijalizovana) implicitna promenljiva *type*. Promenljiva *type* je validna u okviru podstabla onog čvora koji je uveo novi tip. Ova promenljiva predstavlja strukturu podataka koja opisuje pojedinačne tipove podataka. Novi tip se smešta u tabelu tipova *Types*. Moguće je napraviti sledeće kompozitne tipove podatka [67]: enumeraciju, strukturu, uniju, pokazivač ili niz (u sadašnjoj verziji prototipa nije podržano kreiranje korisničkog tipa). Vrednosti (string-konstante) koje treba proslediti operatoru, da bi napravio odgovarajući tip podatka (respektivno) su: *Enumeration*, *Structure*, *Union*, *Pointer*, *Array*. Za tip enumeracije, dodatno se definišu pripadajuće konstante; za strukturu se dodatno definišu pripadajuće polja; za uniju takođe; za pokazivač se dodatno definiše tip podatka na koji pokazuje i za niz se dodatno definišu još i tip podatka elemenata niza i, eventualno, veličina niza. Svaki tip podatka može imati i svoje atribute. U nastavku slede primeri zadavanja tipova podatka (obično akcija koja deklariše novi tip i preostale akcije ne pripadaju istom pravilu):

```
%new_type(Enumeration);
type.constant(IDENTIFIER);

%new_type(Structure);
type.field(declarator_id);
type.attr(COMPLETE_TYPE);
```

```
%new_type(Pointer);
type.base_type(symbol.type);

%new_type(Array);
type.base_type(symbol.type);
type.size(constant);
```

Operator *%new_symbol* zauzuma prazan element u tabeli simbola. Time je spremna za upotrebu (inicijalizovana) implicitna promenljiva *symbol*. Promenljiva *symbol* je validna u okviru podstabla čvora koji je uveo novi simbol. Ova promenljiva predstavlja objekat koji sadrži atribute koji postoje u tabeli simbola. To su ime, tip, atributi simbola i namespace kojem pripada simbol. Svaki od ovih atributa se, objektu *symbol* može dodeliti zadavanjem akcije. U akciji se navodi ime objekta (*symbol*), zatim tačka, pa onda ime atributa kojem se dodeljuje nova vrednost, a zatim se u malim zagradama navodi i nova vrednost atributa. U nastavku slede primeri zadavanja imena, atributa i *namespace* kategorije, objektu *symbol*:

```
symbol.name(IDENTIFIER);
symbol.attr(GLOBAL);
symbol.namespace(FUNCTION);
```

Zadavanje tipa se može napisati na nekoliko načina, u zavisnosti od toga odakle se preuzima nova vrednost. U nastavku su dati primeri zadavanja osnovnog tipa *INT*, preuzimanja tipa podatka od simbola (koji pripada istom pravilu) *type_specifier*, i zadavanja tipa podatka preko implicitnog objekta *type*.

```
symbol.type(INT);
symbol.type(type_specifier);
symbol.type(type);
```

Operator *%lookup_symbol(...)* omogućuje pretraživanje tabele simbola po zadanoj kategoriji simbola. Istovremeno, ovaj operator opisuje i semantički sadržaj simbola. Na primer, sledeća akcija, kao sadržaj datog simbola, dozvoljava pojavu imena koja pripadaju kategoriji *OBJECT*.

```
%lookup_symbol(IDENTIFIER, OBJECT);
```

Operator *%get_signature(...)* omogućuje opis potpisa funkcije, odnosno opis broja i tipova njenih parametara. Ove informacije su editoru potrebne kada želi automatski da pripremi argumente prilikom poziva funkcije. Nakon imena operatora, u malima zagradama, navode se parovi korespodentnih (argument - parametar) simbola. Ukoliko jedan od simbola u paru može biti prazan, korespondentni simbol treba navesti u malim zagradama uz drugi simbol. U nastavku sledi primer upotrebe ovog operatora:

```
%get_signature(
   arguments, parameters;
   argument_expression_list, parameter_list(void);
   expression, parameter_declaration);
```

Ovaj operator se može koristiti i za automatsko popunjavanje parametara u definiciji funkcije na osnovu njene prethodne deklaracije. Tada se u zagradama, nakon operatora, navodi jedino ime simbola čije podstablo se prenosi iz deklaracije u definiciju. Na primer:

```
%get_signature(parameters);
```

Bilo koji simbol može uvesti novi kontekst, koji je onda vidljiv jedino u poddrvetu koji pripada tom simbolu (čvoru). Kontekst se može koristiti za opis statičke semantike. Na primer, neka sintaksna pravila postoje (mogu da se pojave) jedino u određenom kontekstu. Recimo, u *C* programskom jeziku, *break_statement* sme da se pojavi jedino unutar nekog iteracionog iskaza (*while*, *do-while* ili *for*).

Operator kojim se definiše novi kontekst je *%new_context(...)*, a u zagradama se zadaje ime konteksta. Za prethodni primer, iteracioni iskaz treba da sadrži akciju:

```
%new_context(ITERATION);
```

Korišćenje konteksta se opisuje operatorom *@context(...)*. U zagradama se zadaje ime konteksta u kom dato pravilo može da postoji. Za prethodni primer, *break_statement* treba da sadrži akciju:

```
@context(ITERATION);
```

Drugi način korišćenja konteksta je da se u zagradama navede egzistencijalni uslov pod kojim pravilo postoji. Na primer, u *C*

programskom jeziku, deklaracija može istovremeno biti i definicija promenljive, ukoliko postoji deo sa incijalizacijom. Sintaksni opis inicijalizatora se radikalno razlikuje ukoliko je tip promenljive neki od osnovnih tipova ili pokazivač, u odnosu na situaciju kada je tip promenljive struktura, unija ili niz. U prvom slučaju, inicijalizator je predstavljen jednostavnim izrazom, a u drugom slučaju je predstavljen listom inicijalizatora smeštenih između vitičastih zagrada. U ovakvoj sitaciji, sintaksno pravilo koje opisuje prvi slučaj, treba da sadrži akciju:

```
@context(symbol.type(ARITHMETIC_TYPES | Pointer));
```

a sintaksno pravilo koje opisuje drugi slučaj, treba da sadrži akciju:

```
@context(symbol.type(Structure | Union | Array));
```

4.2.6.2 Opis provere tipova

Za opis provere tipova ne postoji poseban operator. U tu svrhu se koristi ime simbola (čiji tip treba promeniti), ili konstanta *root* (ukoliko se menja tip simbola koji se nalazi sa leve strane pravila). Sintaksa ima oblik:

```
X.type(Y);
root.type(A | B);
```

U prvom slučaju, simbol X preuzima tip simbola Y. U drugom slučaju, simbol sa leve strane pravila preuzima uniju tipova simbola A i B. Ovakvom akcijom se opisuje skup validnih tipova podataka koje dati simbol može da sadrži. Kada se konkretizuje sadržaj datog čvora, on će, naravno, zadržati samo jednu vrednost iz tog skupa (onu koja odgovara njegovom sadržaju). U zagradama se može naći i ime atributa koji opisuje neki tip podatka.

4.2.6.3 Opis formata prikaza

Formatiranje prikaza struktura na ekranu obuhvata definisanje boje različitih vrsta imena (ključnih reči, promenljivih, funkcija, ...), zatim indentaciju i podatak da li struktura svoj prikaz započinje u novoj

liniji. U prvoj verziji prototipa nije podržano definisanje boja od strane korisnika, koristi se podrazumevana tipografija.

Operatorom *%new_line* se označava da prikaz strukture počinje u novoj liniji.

Operatorom *%indent* se označava da se struktura prikazuje uvučeno u odnosu na roditeljsku strukturu (onu koja je sadrži).

4.3 Specifikacioni parser

Pisanje parsera i skenera za iole veću gramatiku je zahtevan posao. Zato, još od 1970-tih godina, postoje generatori ovih alata. Najpoznatiji i najčešće korišćeni generatori su *Lex* [45] i *Yacc* [36]. *Lex* je generator leksičkih analizatora (skenera), a *Yacc* je generator parsera. Oba alata generišu *C* kod, i mogu lako da sarađuju. Skener se generiše na osnovu specifikacije koja sadrži opis simbola (u formi regularnih izraza) i opis akcija (koje se izvršavaju kada se prepozna dati simbol). Parser se generiše na osnovu specifikacije koja sadrži opis sintaksnih pravila jezika, i opis akcija (koje se izvršavaju kada se prepozna dato pravilo). Skener (izgenerisan upotrebom *Lex*-a) prepoznaje simbole jezika, odnosno, njima odgovarajuće tokene. Svaki put kada je parser (izgenerisan upotrebom *Yacc*-a) spreman da nastavi svoju aktivnost, on preuzima novi simbol od skenera pozivom odgovarajuće funkcije. Skener prosleđuje informacije o (sledećem) prepoznatom simbolu.

Po uzoru na *Lex* i *Yacc*, napravljeni su alati *JLex* [10] i *Parser Constructor CUP* [33], koji generišu *Java* kod. Oba alata su dalje prerađena za *.NET* platformu, tako što su napisane verzije ovih programa, na *C#* programskom jeziku. Tako su nastali *C# Lex* i *C# CUP* [34].

Modul specifikacionog parsera je implementiran upotrebom alata *C# Lex* i *C# CUP*. *C# Lex* specifikacija je vrlo slična *Lex* specifikaciji, i sastoji se od opisa simbola u formi regularnih izraza, i od akcija koje piše korisnik. U specifikaciji su opisani simboli koji se koriste u pisanju specifikacije prototipa. *C# Lex* iz polazne specifikacije (*scanner.lex*) izgeneriše datoteku (*scanner.lex.cs*), koja predstavlja skener za specifikaciju prototipa. *C# CUP* specifikacija je vrlo slična *Yacc* specifikaciji, i sastoji se od sintaksnih pravila jezika, i od akcija koje piše korisnik. U toj specifikaciji je opisana sintaksa specifikacije prototipa, odnosno način na koji se pišu pravila i akcije. *C# CUP* iz

polazne specifikacije (*parser.cup*) izgeneriše datoteku (*parser.cs*), koja predstavlja parser za specifikaciju prototipa. Skener se, implicitno, poziva iz parsera.

Obe izgenerisane datoteke (i skener i parser) su dobijene zasebnim postupkom, a zatim su inkorporirane u prototip.

4.4 Specifikacione tabele

Specifikacione tabele sadrže internu formu specifikacije programskog jezika. Postoji više tabela organizovanih po vrsti informacija koje sadrže. Sve tabele koristi editorska mašina tokom editovanja programskog teksta, a u procesu interpretiranja pravila gramatike.

Tabela *Grammar* sadrži sve podatke o gramatici programskog jezika. Ovi podaci uključuju: ime jezika, početni simbol gramatike, opis komentara, *namespace* podatke, informacije o radu sa datotekama, opis *lvalue* reference, (referencu na) tabelu simbola jezika, (referencu na) tabelu tipova i (referencu na) tabelu ključnih reči.

Tabela *Types* sadrži sve podatke o tipovima programskog jezika. Ovi podaci su podeljeni u više kategorija: osnovni tipovi, kompozitni tipovi i grupe tipova podataka. Osnovni tip podatka je opisan imenom, stringom, regularnim izrazom koji gradi literal, nizom mogućih konverzija u druge tipove podataka i nizom operacija koje se mogu primeniti na dati tip podatka. Operacija može biti unarna ili binarna. Unarna operacija je opisana imenima operatora, tipom rezultata i informacijom o tome da li rezultat predstavlja *lvalue* referencu. Binarna operacija je, osim ovim podacima, opisana i tipom drugog operanda koji učestvuje u operaciji.

Tabela *Symbols* sadrži sve simbole gramatike. Svaki simbol sadrži opis imena, sadržaja simbola (koji može biti leksičkog ili sintaksnog tipa), eventualne simbole (stringove) kojima započinje ili završava prikaz tog simbola, informacije o tome da li simbol treba prikazati u novoj liniji i/ili uvučeno u odnosu na prethodni red (*indent*) i informacije o statičkoj semantici. Opis sintaksnog sadržaja čvora predstavljen je hijerarhijskom strukturom elemenata, uz prisustvo opisa kontekstnih informacija. Opis statičke semantike se sastoji od informacija da li dati simbol uvodi novi tip podatka i podatke o tom tipu, da li uvodi novi simbol u tabelu simbola i podatke o tom simbolu, da li uvodi novi opseg važenja, da li definiše kompilacionu jedinicu, da

li uvodi nove kontekstne informacije i koje, kao i informacije o skupu tipova podataka koji su dozvoljeni za taj simbol.

Niz ključnih reči *Keywords* služi za proveru da li se nova imena, koja korisnik uvodi u program, podudaraju sa ključnim rečima.

4.5 Tabela simbola

Tabela simbola se koristi za smeštanje informacija o imenima u programu. Na najnižem nivou, tabela simbola je rečnik: ona preslikava ime u informacije koje postoje o njemu. Svaki element tabele simbola odgovara jednoj deklaraciji imena.

Osnovne operacije su ubacivanje novog simbola (*insert*) i efikasno pretraživanje postojećih simbola (*lookup*). Novi simbol će biti ubačen, samo ako se njegov string razlikuje od ključnih reči, i ako je jedinstven u svom opsegu važenja. Simbol se u tabelu simbola ubacuje preko heš tabele. Jedna heš tabela opisuje jedan opseg važenja i 'vidi' samo onaj deo tabele simbola koji sadrži imena iz tog opsega. Time je rešen i problem ugnježdenih opsega važenja, u kojima unutrašnja deklaracija sakriva spoljašnje deklaracije istog imena. Pretraživanje tabele simbola se radi na osnovu nekog kriterijuma (kategorija imena ili tip podatka). Proces pretraživanja kreće od trenutnog opsega važenja, pa se proširuje na opsege koji sadrže trenutno pretraživani opseg.

Tabela simbola je implementirana kao dinamički niz, čime joj je obezbeđen dinamički rast tabele. Informacije koje se smeštaju u tabelu simbola uz svako ime uključuju string imena, kategoriju (*namespace*) simbola, tip simbola i atribute simbola.

4.6 Editorska mašina i editorski interfejs

Osnovna uloga editorske mašine jeste da rukuje internom reprezentacijom programskog teksta. Interna reprezentacija je predstavljena drvetom čiji su čvorovi implementirani kao strukture. Svaki čvor poseduje pokazivač na roditelja, pokazivač na svog prvog potomka i pokazivače na svog prethonika i svog sledbenika. Za potrebe prikazivanja strukture na ekran, čvor sadrži koordinate gornjeg levog karaktera strukture i donjeg desnog prikazanog karaktera, koji pripada toj strukturi. Svaki čvor sadrži referencu na heš tabelu koja definiše

opseg važenja (*scope*) kojem on pripada. Mogući tip podatka koji može opisivati sadržaj čvora je predstavljen nizom, kao i skup konteksta koji opisuju taj deo poddrveta. Svaki čvor sadrži i referencu na deskriptor simbola koji predstavlja. Deskriptor sadrži gramatički opis simbola.

Za obilazak drveta se koristi *preorder traversal* metoda, koja podrazumeva primenu rekurzije. U nastavku sledi meta-kod ove metode:

```
Preorder(n)
  if n is not NULL
    Inspect n;
    Preorder(n->right);
```

Editorski interfejs je deo sistema koji je u direktnoj komunikaciji sa korisnikom, s jedne strane, i editorskom mašinom, s druge strane. To ga čini posrednikom između interne reprezentacije i korisnika, pa je njegova uloga dvostruka.

S jedne strane, on je zadužen da prihvata komande od korisnika. Korisnik može zadati komandu preko menija (pomoću miša) ili preko definisanih skraćenica (pomoću tastature). Korisnička komanda predstavlja zahtev za izvršenjem neke editorske operacije, kao na primer: operacije ubacivanja nove strukture, modifikacije selektovane strukture ili zahtev za pronalaženje svih mesta na kojima se pojavljuje dato ime. Editorski interfejs, ovaj zahtev za izvršenjem operacije, prosleđuje editorskoj mašini, u vidu zahteva za modifikacijom interne strukture.

Kada primi zahtev, editorska mašina će pokušati da ga izvrši. Ukoliko ne uspe, ona će pripremiti sadržaj poruke o nemogućnosti izvršenja zahteva, i proslediti je editorskom interfejsu, koji dalje tu poruku treba da prikaže korisniku (putem *status bar*-a, dijaloga, ...). Ukoliko editorska mašina uspešno izvrši zahtev, njen odgovor editorskom interfejsu će biti pripremljen niz znakova koji odslikava izmenu u internoj strukturi programskog teksta, koju editorski interfejs treba da prikaže na ekran (*pretty-print*). Može se desiti da editorska mašina izvrši deo operacije i da joj za nastavak treba još informacija od korsnika. U tom slučaju, ona priprema podatke koji su potrebni editorskom interfejsu za otvaranje dijaloga, putem kojeg će se od korsnika preuzeti dodatne informacije. Kada od editorskog interfejsa dobije dalja uputstva, editorska mašina završava izvršenje započetog zahteva.

S druge strane, editorski interfejs prihvata rezultat obrade zahteva od editorske mašine, i prikazuje ih korisniku. Rezultati se mogu prikazati u raznim oblicima (u zavisnosti od vrste informacije koja se želi prikazati), kao na primer: prikaz izmenjenog programskog teksta (*pretty-printing*), prikaz poruke, prikaz dijaloga za unos ili modifikaciju leksičkog sadržaja ili prikaz dijaloga za izbor jedne od ponuđenih alternativa.

4.6.1 Funkcionalne osobine editorske mašine

U nastavku su pobrojane najvažnije osobine editorske mašine, s ciljem sagledavanja funkcionalnosti:

- rukuje internom reprezentacijom programskog teksta (aspekt parsera)
- rukuje tabelom simbola editovanog programskog teksta (aspekt parsera)
- rukuje prikazom programskog teksta na ekran (*pretty-printer*)
- rukuje unosom i modifikacijom leksičkog sadržaja strukture (skener)
- rukuje obradom statičke semantike u toku modifikacije interne strukture (uključuje *type-checker*)
- priprema poruke o statusu izvršenja zahteva, koju editorski interfejs treba da prikaže korisniku
- obezbeđuje rad sa datotekama
  - internu strukturu pohranjuje u datoteku
  - vrši konverziju programskog koda u tekstualni format (*export* operacija)

Iz pregleda se može zaključiti da sistem pruža ekvivalentnu funkcionalnost (*pretty-printer*, skener, *type-checker*, neki aspekti tipičnog parsera), iako ne sadrži posebne alate koji su tipični za jezički-zasnovane sisteme 2.7

4.6.2 Kontekstno editovanje

Kontekstno editovanje podrazumeva da se korisniku u svakom koraku editovanja, na jednostavan i prihvatljiv način, nude one informacije koje su mu u tom trenutku potrebne. Da bi se realizovalo kontekstno editovanje, potrebne su informacije i o sintaksi i o statičkoj semantici jezika.

4.6.2.1 Strukture

Primer strukture definicije promenljive (za programski jezik *C*) je prikazan na slici 4.2. Podvučene podstrukture su opcione, a nepodvučene su obavezne. Strukture se dele na terminalne, koje ne mogu da sadrže podstrukture, i neterminalne, koje mogu da sadrže podstrukture. U primeru sa slike 4.2 sve podstrukture su terminalne. Primer neterminalne strukture bi bilo telo funkcije, koje se sastoji od više podstruktura od kojih svaka opisuje neki iskaz (naredbu).

```
type name "=" value ";"
```

Slika 4.2: Izgled strukture definicije promenljive

Struktura je **prazna**, kada su sve njene podstrukture prazne (kada sadrže upitnike). Na slici 4.3 je prikazana prazna struktura definicije promenljive (koja sadrži samo obavezne podstrukture).

```
? ?;
```

Slika 4.3: Prazna struktura definicije promenljive

Popunjavanje strukture se sastoji od zamene upitnika odgovarajućim sadržajima. U slučaju prve podstrukture (*type*) sa slike 4.3, umesto upitnika se može pojaviti ime nekog prethodno definisanog tipa, a u slučaju druge podstrukture (*name*) umesto upitnika može da dođe niz znakova, koji jednoznačno određuje promenljivu.

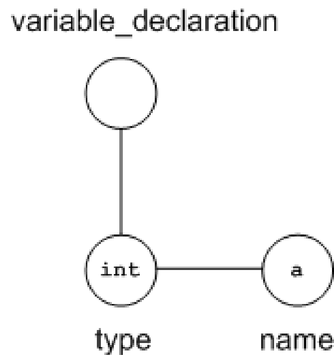
Struktura je **puna** kada se popune sve njene podstrukture. Na slici 4.4 je prikazana puna struktura definicije promenljive (koja sadrži samo obavezne podstrukture).

```
int a;
```

Slika 4.4: Puna struktura definicije promenljive

Struktura je **poluppopunjena** kada je bar jedna od njenih podstruktura prazna (sadrži upitnik).

Implementacija prethodne strukture (sa obaveznim podstrukturama) ima oblik usmerenog poddrveta, koje je prikazano na slici 4.5. String ";" pripada čvoru *variable_declaration*, i prikazuje se posle ispisa njegovog poddrveta.



Slika 4.5: Interna reprezentacija strukture definicije promenljive

4.6.2.2 Dijalozi

Popunjavanje strukture podrazumeva:

1. zadavanje
   (a) novih (ili izmenu postojećih) imena, ili
   (b) novih (ili izmenu postojećih) literala

2. selekciju

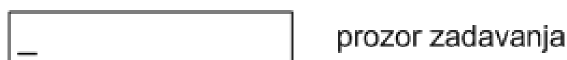
    (a) jednog od prethodno zadanih imena, ili

    (b) jedne od mogućih sintaksnih struktura.

Na primer, zadavanje novog imena je vezano za podstrukturu *name*, zadavanje novog literala je vezano za podstrukturu *value*, a selekcija jednog od prethodno zadanih imena je vezana za podstrukturu *type*.

Prilikom zadavanja (ili izmene) imena ili literala, neophodno je sprečiti (filtrirati) nedozvoljene znakove. Time se sprečava pojava leksičkih grešaka. Zato se dijalog, posredstvom koga se zadaju imena i literali, naziva **leksički dijalog**. Iako je prevashodna namena leksičkog dijaloga da filtrira leksičke greške, on filtrira i neke semantičke greške. Tako, on dozvoljava zadavanje samo jedinstvenih imena, odnosno zadavanje samo literala čiji tip je prihvatljiv (na primer, ne prihvata *string* literal kao početnu vrednost *int* promenljive).

Leksički dijalog se sastoji od prozora zadavanja, u kome se zadaju znakovi imena ili literala (slika 4.6).



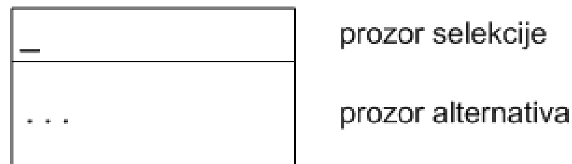
Slika 4.6: Leksički dijalog

Znak "_" pokazuje poziciju kursora u leksičkom dijalogu, odnosno mesto na kome se zadaje novi znak. Znak, određen pozicijom kursora, se može izbrisati. Kursor se može pomerati na neki od znakova koji mu prethode ili slede. Moguće je odustati od zadavanja (*escape*) ili završiti zadavanje (*enter*). U oba slučaja se izlazi iz leksičkog dijaloga. To se u drugom slučaju ne desi jedino ako zadano ime ili zadani literal nisu prihvatljivi prema leksičkim i semantičkim pravilima.

Smisao selekcije imena, odnosno selekcije sintaksne strukture je da se dozvoli izbor isključivo prihvatljivog imena, odnosno sintaksne strukture. Na taj način se sprečavaju (filtriraju) sintaksne greške. Zato se dijalog, posredstvom koga se selektuju imena ili sintaksne strukture, naziva **sintaksni dijalog**. Iako je prevashodna namena sintaksnog dijaloga da filtrira sintaksne greške, on filtrira i neke semantičke greške. Na primer, on nudi samo imena definisanih tipova prilikom popunjavanja podstrukture *type*, ili samo imena definisanih

promenljivih, određenog tipa, prilikom popunjavanja desne strane iskaza pridruživanja, ili samo oznake dozvoljenih iskaza prilikom popunjavanja strukture koja opisuje telo funkcije.

Sintaksni dijalog se sastoji od prozora selekcije i prozora alternativa (slika 4.7).



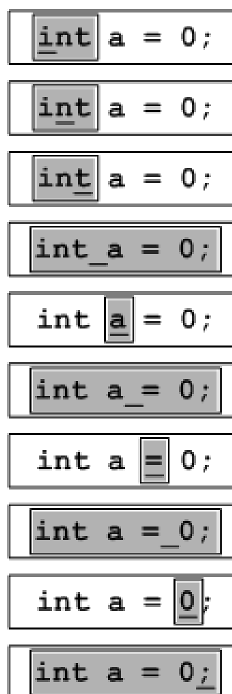
Slika 4.7: Sintaksni dijalog

Prozor alternativa sadrži skup alternativa, od kojih se jedna selektuje. Selekcija se obavlja ili pomoću miša ili navođenjem jedinstvenog niza početnih znakova selektovane alternative u prozoru selekcije. Preostali znakovi selektovane alternative se automatski popunjavaju u prozoru selekcije. Kada postoji samo jedna alternativa, njeni znakovi se automatski pojavljuju u prozoru selekcije. U prozoru selekcije se prihvataju samo znakovi koji odgovaraju nekoj od alternativa. Znak, određen pozicijom kursora, se može izbrisati. Moguće je odustati od selekcije (*escape*) ili završiti selekciju (*enter*). U oba slučaja se izlazi iz sintaksnog dijaloga.

4.6.2.3 Kretanje po programskom tekstu i njegovo markiranje

Tradicionalan i prirodan način kretanja po programskom tekstu zahteva da se kursor može direktno pozicionirati (mišem ili dirkama tastature) na bilo koji znak programskog teksta. Ovaj zahtev *USE* editor može ispoštovati, bez obzira na to što je programski tekst komponovan od struktura. Činjenica da je programski tekst organizovan po strukturama ne ometa ostvarenje prethodnog zahteva, i stvara neophodne preduslove za automatsko markiranje programskog teksta. Tako se podstruktura markira dok god je kursor pozicioniran na neki od njenih znakova, a struktura se markira dok god je kursor pozicioniran na neki od njenih znakova, koji istovremeno ne pripadaju njenim podstrukturama. Na slici 4.8 su prikazani primeri automatskog markiranja za strukturu definicije promenljive. Slika

prikazuje stanja ekrana, nakon sukcesivnih pozicioniranja kursora na sledeći znak (upotrebom dirke →). Markirani tekst je uokviren zasivljenim pravougaonikom.



Slika 4.8: Primeri automatskog markiranja

4.6.2.4 Rukovanje strukturama i dijalozima

Rukovanje strukturama obuhvata njihovo ubacivanje u programski tekst i brisanje iz programskog teksta. To omogućuju operacija ubacivanja (dirka *insert*) i operacija brisanja (dirka *delete*). Rukovanje dijalozima se sastoji od ulaska u leksički i sintaksni dijalog. To omogućuje operacija modifikacije (dirka *enter*). Sve tri operacije se odnose na markirani deo programskog teksta, a rezultat njihove primene zavisi od toga šta je markirano i menja se od slučaja do slučaja (kontekstno je zavisan).

Operacija **ubacivanja** omogućuje ubacivanje nove strukture iza markirane strukture (ako je ubacivanje uopšte moguće, inače ova operacija nema efekta). U okviru obavljanja operacije ubacivanja, automatski se ulazi u sintaksni dijalog radi selektovanja alternative koja se ubacuje. Kada postoji samo jedna alternativa, ona se ubacuje automatski.

Operacija **brisanja** omogućuje brisanje markirane strukture (ako je brisanje moguće, inače ova operacija nema efekta). Obavljanje operacije brisanja dovodi ili do izbacivanja kompletne markirane strukture, ili do zamene njenog sadržaja upitnikom (to zavisi od konteksta). Ukoliko je struktura opciona, ona može da se ukloni iz drveta, u suporotnom ne može. Ako je struktura obavezna i terminalna onda, primenom operacije brisanja, dolazi do zamene njenog sadržaja upitnikom. Ukoliko je struktura obavezna i neterminalna onda operacija brisanja nema efekta.

Operacija **modifikacije** omogućuje aktiviranje leksičkog ili sintaksnog dijaloga radi popunjavanja prazne strukture, odnosno radi izmene sadržaja pune markirane strukture (ova operacija nema efekta, ako je markirana neterminalna struktura).

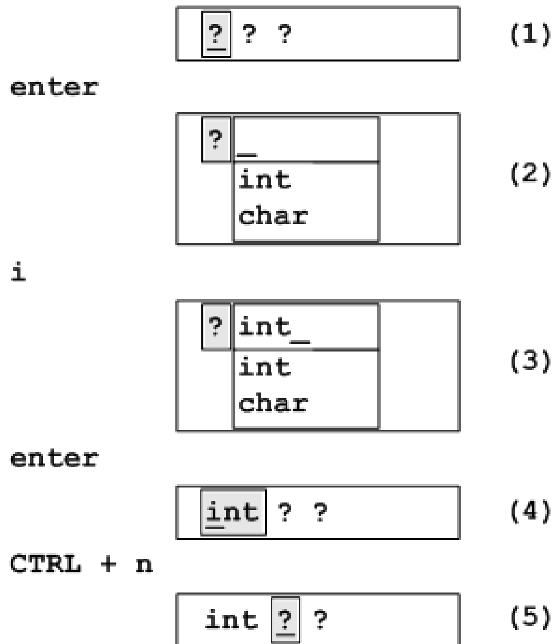
Primena prethodnih operacija se ilustruje na primeru zadavanja segmenta programa sa slike 4.9.

```
int a;
```

Slika 4.9: Primer programskog teksta

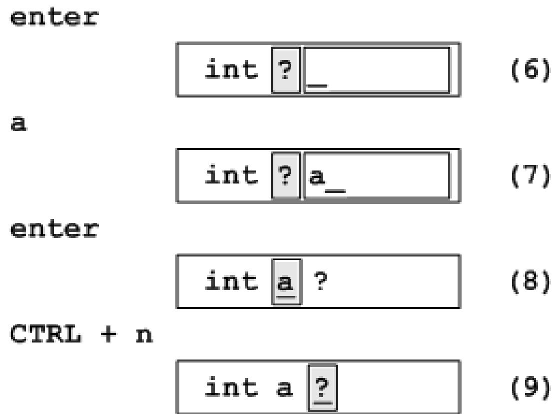
Na slikama 4.10, 4.11, 4.12 i 4.13 je prikazan postupak kontekstnog editovanja, u toku koga se komponuje programski tekst sa slike 4.9. Slika sadrži sekvence stanja ekrana. Sa leve strane sekvenci se nalaze oznake dirki sa različitim ulogama. Polazna pretpostavka je da je editovana datoteka prazna.

Operacija modifikacije (dirka *enter*) dovodi do otvaranja sintaksnog dijaloga (stanje ekrana (2)), koji nudi izbor jednog od ponuđenih tipova (*int* i *char*). Unosom znaka *i*, automatski se popunjava string u prozoru selekcije (3). Dirkom *enter* potvrđuje se odabir selektovanog tipa (4). Operacija *next* (dirka *CTRL + n*) omogućuje prelazak na sledeću strukturu (5).



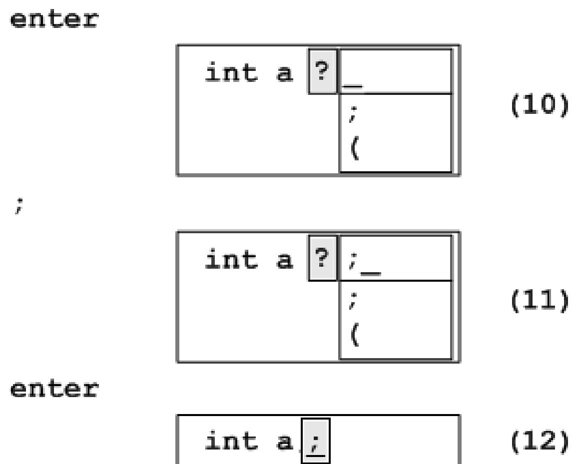
Slika 4.10: Primer kontekstnog editovanja (1)

Ulaskom u selektovanu podstrukturu imena (6), otvara se leksički dijalog, koji očekuje unos znakova imena (7). Dirka *enter* označava kraj unosa (8). Operacija *next* (dirka *CTRL + n*) omogućuje prelazak na sledeću strukturu (9).



Slika 4.11: Primer kontekstnog editovanja (2)

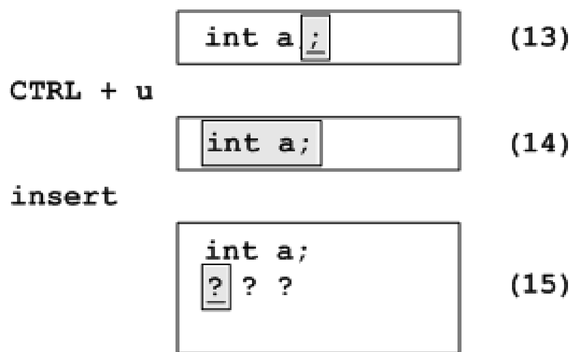
Ulaskom u sledeću selektovanu podstrukturu, otvara se sintaksni dijalog (10), koji nudi izbor delimitera ";" (kojim se završava deklaracija promenljive) ili otvorene male zagrade (kojim započinje definicija funkcije).



Slika 4.12: Primer kontekstnog editovanja (3)

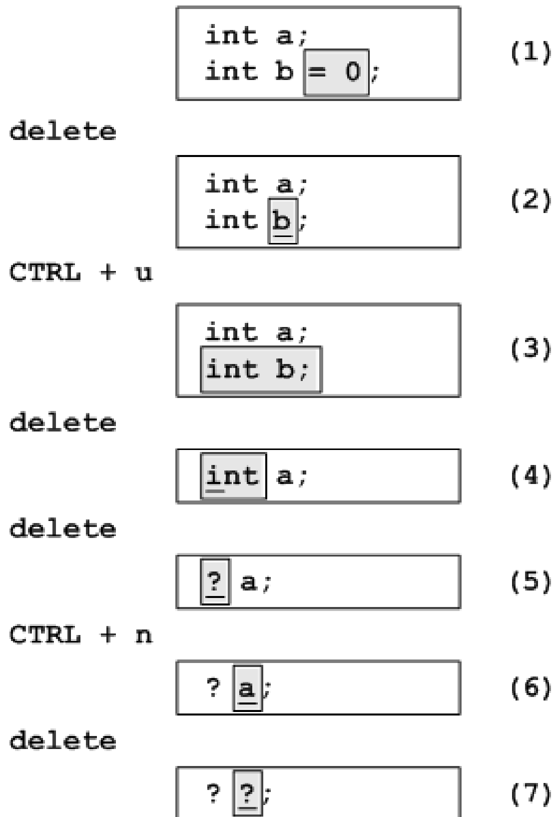
Unosom znaka ";" (11) i potvrdom izbora (dirka *enter*) (12) završava se editovanje strukture definicije promenljive.

Operacija *up* (dirka *CTRL* + u) omogućuje premeštanje selekcije sa znaka ";" (13) na strukturu definicije promenljive (14). Dirka *insert* poziva istoimenu operaciju, koja, u ovom slučaju, ne otvara sintaksni dijalog jer postoji samo jedna alternativa. Zato se automatski ubacuje nepopunjena struktura deklaracije (sa svojim obaveznim podstrukturama) (15).



Slika 4.13: Primer kontekstnog editovanja (4)

Na slici 4.14 su prikazani primeri kontekstnog brisanja (dirka *delete*). Početno stanje (1) prikazuje dve deklaracije promenljive, od kojih druga struktura sadrži i neobaveznu podstrukturu inicijalne vrednosti. Operacija brisanja ove podstrukture dovodi do njenog uklanjanja iz strukture (2). Operacija *up* (dirka *CTRL* + *u*) dovodi do proširenja selekcije sa strukture *name* na strukturu deklaracije (3). Operacija brisanja strukture definicije promenljive (*b*) dovodi do uklanjanja cele (neobavezne) strukture (4). Kada se operacija brisanja primeni na terminalne obavezne strukture, kao što su *type* i *name*, dolazi do uklanjanja njihovog sadržaja (5 i 6), odnosno do njihove zamene upitnikom (7).



Slika 4.14: Primeri operacije brisanja

Ukoliko sintaksni dijalog služi za odabir imena, prozor alternativa, osim imena zadanog osnovnog tipa, sadrži i imena složenih tipova (kao što su *Structure*, *Union*, *Array* ili funkcija). Iza ovih imena u dijalogu se pojavljuju znaci koji upućuju na tip imena, pa tako se (za programski jezik *C*) iza strukture i unije pojavljuje tačka (kao selektor), iza niza `[` i iza funkcije `(`. Kada korisnik bude odabrao ovakvo ime, recimo ime strukture, editor će automatski dodati tačku i nepopunjenu strukturu imena sa upitnikom. Tip kompletne strukture zadržava zadani tip podatka. Ovim postupkom se omogućava korisniku jednostavan pristup poljima struktura i unija i elementima niza. Međutim, editor

za sada ne proverava da li postoji polje traženog tipa unutar strukture, tako da ovaj postupak može korisnika dovesti do nemogućnosti daljeg editovanja datog izraza.

4.6.2.5 Prednosti kontekstnog editovanja

Kontekstno editovanje je veoma značajna osobina *USE* strukturnog editora, a prednosti ovakvog editovanja su vrlo važne:

- Uz pomoć kontekstnog editovanja dovoljno je uneti ime samo jednom, a kasnije ga birati iz spiska alternativnih imena.
- Mnogo je brže odabrati ime iz spiska alternativnih imena, nego unositi ga ponovo preko tastature.
- Teško je pamtiti i voditi računa o svim imenima u iole većem programu, ili kada se koriste moduli koje je pisao neko drugi. Kontekstno editovanje čini nepotrebnim pamćenje imena.

4.7 Neimplementirana funkcionalnost

4.7.1 Neimplementirane osobine jezika

Prototip ne sadrži implementaciju kompletnog *C* programskog jezika. Osnovni razlog tome je skraćenje vremena izrade prototipa. Za kasniju implementaciju su odabrane one osobine jezika koje se mogu zameniti drugim, ili one osobine koje nisu od suštinskog značaja za prototip, a njihova implementacija je vrlo slična implementaciji neke postojeće osobine. Neke od osobina su samo implementirane u manjem obimu. Na primer, nije podržan *switch* iskaz, pošto se on može zameniti sa više *if* iskaza. Funkcionalnost editora je na visokom nivou, korisnik nije uskraćen, a vreme izrade prototipa je skraćeno.

U nastavku sledi spisak osobina *C* programskog jezika koje nisu implemetirane, ili su delimično implementirane, u aktuelnoj (prvoj) verziji prototipa. Spisak je podeljen u nekoliko grupa. Istoj grupi pripadaju one osobine koje su slične s aspekta implementacije.

1. osobine koje se mogu zameniti nekim drugim osobinama

   (a) infiksni i postfiksni inkrement i dekrement operatori se mogu zameniti operatorom sabiranja, odnosno oduzimanja, i primenom zagrada
   (b) ternarni operator se može zameniti *if* strukturom
   (c) od svih operatora pridruživanja, implementiran je samo operator =. Svi ostali (*=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=) se implementiraju na isti način, a mogu biti zamenjeni kombinacijom operatora = i željenog operatora
   (d) *switch* naredba (sa *case* i *default* naredbama) se može zameniti *if* strukturama

2. osobine koje nisu od suštinskog značaja za prototip
   (a) pretprocesorske direktive
   (b) (*storage class spedifier*) *auto* i *register*, a implementacija je analogna ostalim *storage class specifier* (*extern* i *static*)
   (c) kvalifikatori tipa (*type qualifier*) *const* i *volatile*, a njihova implementacija je slična implementaciji *storage class specifier*
   (d) stari stil pisanja definicije funkcije, uključujući i neimenovane parametre
   (e) pisanje deklaratora u zagradama i korišćenje neimenovanih tipova kod struktura i unija

3. osobine koje su implementirane u smanjenom obimu
   (a) uopšteni iskaz dodele, koji može predstavljati više uzastopnih iskaza dodele, je pojednostavljen na iskaz sa jednim operatorom dodele
   (b) podržan je samo jedan stepen indirekcije (ne može se deklarisati pokazivač na pokazivač)
   (c) jedna deklaracija može deklarisati samo jedno ime (umesto više imena)
   (d) veličina niza se opisuje konstantom, a ne konstantnim izrazom
   (e) definicija funkcije na kraju definicije parametara ne može sadržati simbol "..."

(f) ne može postojati više deklaracija istog imena

(g) literal za tip podatka *char* može biti samo znak pod apostrofima, a ne i oktalni broj ili *escape* sekvenca pod apostrofima

(h) u deklaraciji polja strukture ili unije nije moguće definisati *bit-field* (polje koje se specificirano brojem bita, pri čemu se njegova duzina postavlja u deklaraciji pomoću ":")

(i) uopšteni izraz, koji može predstavljati više iskaza dodele odvojenih zapetom, je pojednostavljen na jedan izraz

(j) vrednost argumenta prilikom poziva funkcije je pojednostavljena (sa iskaza dodele) na izraz

4. ostale osobine

   (a) upotreba ključne reči *typedef*
   (b) tip podatka *string*
   (c) pokazivači na funkcije
   (d) nije podržana *goto* naredba i labele, jer se (objektivno) ne koriste često, a njihova implementacija se ne razlikuje od implementacije ostalih struktura
   (e) opis tipa podatka ne sadrži informaciju o broju bita (opsegu) vrednosti, što znači da editor ne proverava opseg konstanti. Ova informacija se razlikuje za različite kompajlere.

4.7.2 Neimplementirane operacije

U prvoj verziji prototipa nisu implementirane sledeće editorske operacije:

1. *undo*
2. *cut/copy/paste*
3. strukturna transformacija (*refactoring*)

Poglavlje 5

Zaključak

5.1 Procena ostvarenih ciljeva

Oživeti ideju strukturnog editovanja Realizacijom prototipa usmerena je pažnja na strukturni način editovanja. Time je postavljena osnova za dalje razvijanje modela strukturnih editora i njihovih okruženja. Iako je prototip predstavljen vrlo maloj grupi ljudi, prednosti strukturnih operacija su odmah primećene.

Intuitivni korisnički interfejs Grupa ljudi koja je koristila prototip je lako prihvatila intuitivni korisnički interfejs. Cilj intuitivnosti je postavljen da bi se izbegle mane strukturnih editora koje su se ogledale u načinu interakcije sa korisnikom. Intuitivnosti korisničkog interfejsa najviše su doprinele sličnost sa tradicionalnim editorima i tehnika kontekstnog editovanja. Kontekstno editovanje je vrlo efikasna tehnika i jednostavna za korišćenje. Mehanizam se može implementirati ukoliko su semantičke informacije uvek ažurne i dostupne u toku editovanja. Kontekstno editovanje nije korisno samo početnicima koji imaju problema sa sintaksnom, već i iskusnim programerima čiji je osnovni problem voditi računa o velikim programima i unutrašnjim zavisnostima.

Generičko okruženje Cilj uopštavanja je potpuno postignut za klasu proceduralnih programskih jezika (*C*, *Pascal*, *Modula*, ...). Editor

je generički, njegova funkcionalnost i ponašanje ne zavise od jezika koji se koristi. Ovo je postignuto parametrizacijom delova (alata) koji zavise od jezika, a to su parser, skener i *pretty-printer*.

Kvalitetna vizualizacija Prototip je realizovan savremenim alatima, u tom smislu je ispratio savremene grafičke trendove. U realizaciji su iskorišćeni svi vizuelni elementi koji se koriste u interakciji čovek-računar (dijalozi, meniji, prozori i različiti fontovi). Upotrebljivost proizvoda, u smislu brzine i memorijskih zahteva, je u skladu sa savremenim očekivanjima od softverskih alata.

Proširiva arhitektura sistema Proširivost sistema se ogleda u: (a) mogućnosti okruženja da (u perspektivi) podrži druge klase programskih jezika (na primer, objektno orijentisane), (b) mogućnosti proširenja editora novim operacijama, (c) mogućnosti proširenja korisničkog interfejsa dodavanjem novih komandi i (d) mogućnosti lakog konfigurisanja izgleda okruženja.

***Compile-time* analize** Prototip strukturnog editora podržava sintaksu i statičku semantiku programskog jezika, tj. *compile-time* analize, a ne sadrži *run-time* analize.

5.2 Prednosti realizovanog strukturnog editora

Poznavanje sintakse. Predstavljeni strukturni editor vodi korisnika, tako da korisnik ne mora detaljno da poznaje sintaksu programskog jezika. Ovo je veoma bitno za programere početnike i neprofesionalne programere.

Smanjen broj grešaka. Potputno je sprečena pojava sintaksnih grešaka, i drastično je smanjena pojava semantičkih grešaka (statička semantika). Praktično je nemoguće napraviti sintaksno neispravan program. Broj mogućih semantičkih grešaka je smanjen jer se imena dodeljuju samo jednom, a dalje se biraju u zavisnosti od konteksta. Na mestu gde se ime koristi, korisnik bira iz liste alternativnih imena koja su validna u datom kontekstu.

Intuitivni korisnički interfejs. Kursor se može linearno pomerati sa znaka na znak unutar jedne strukture i može se pomerati sa jedne linije teksta na drugu, a ne samo sa jedne strukture na drugu. Ova osobina korisničkog interfejsa nije uobičajena za strukturne editore, a prisutna je u tradicionalnim tekstualnim editorima.

Čitljivost i preglednost. Automatska indentacija i *syntax highlihting* povećavaju preglednost programa. *USE* strukturni editor vodi računa o komentarima i belinama, što nije česta pojava u strukturnim editorima. Na ovaj način, editor dodatno povećava čitljivost programskog teksta.

Vizuelni dizajn. Bitan za čitljivost i shvatanje programa. Tekstualni prikaz (generisan *pretty-printer* alatom) je obogaćen dodatnim informacijama koje koriste tipografske stilove (font, boje) ili različite prikaze dokumenta (na primer sa ili bez komentara).

Konfigurabilnost. Sistem pruža mogućnost podešavanja i izmene izgleda i načina rada okruženja, čime se prilagođava individualnim potrebama korisnika. Vizuelni stil se lako podešava za različite programske jezike i zadatke.

Jezička nezavisnost. Korisnik može istovremeno da rukuje dokumentima koji su pisani na različitim programskim jezicima. Ova osobina je pogodna za programere početnike. Dovoljno je da nauče samo ovo okruženje, a u njemu mogu da edituju programe u bilo kom programskom jeziku.

Povećana produktivnost. Zbog svih prethodno pobrojanih osobina, produktivnost programera postaje veća.

5.3 Odnos *USE* editora i kompajlera

Leksičko i sintaksno filtriranje u *USE* editoru imaju istu funkciju kao i leksička i sintaksna analiza u kompajleru. Nema suštinske razlike između podataka koji su rezultat aktivnosti *USE* editora i podataka koji su rezultat leksičke i sintaksne analize kompajlera. *USE* editor bi zato mogao da posluži kao *front-end* kompajlera i zameni njegov leksički i sintaksni analizator. Ova mogućnost nije od značaja za postojeće

kompajlere, nego, eventualno, za buduće kompajlere, koji će se tek projektovati.

5.4 Budući rad

Moglo bi se reći da je malo dobrih interaktivnih strukturnih editora. To je zato što postoji želja da se korisniku ponudi udobnost, pa se u skladu s tim donose odluke u vezi dizajna koje vode vrlo kompleksnim sistemima. Posledica toga je da je korisniku vrlo teško da formira mentalni model, koji je dovoljno jednostavan, da bi bilo moguće lako i jednostavno razumeti posledicu interakcije. Ovim prototipom je dat temelj i osnova za dalje razvijanje modela strukturnih editora, koji bi mogao podržati racionalniji pristup donošenju dizajnerskih odluka. Jedna od najznačajnih osobina ovog modela je drugačije rukovanje selekcijom, koja je doprinela intuitivnijem kretanju kroz strukture programskog teksta.

Budući rad na početnoj verziji prototipa bi mogao da uključi:

- realizaciju neimplementiranih funckionalnosti
  - ubacivanje ispred selektovane strukture
  - *undo*
  - *cut/copy/paste*
  - strukturna transformacija (*refactoring*)
  - *syntax highlighting* koji definiše korisnik
- proširenje formalizma za opis programskog jezika
  - podržati i opis objektno-orijentisanih jezika
- proširenje skupa operacija (editora i okruženja)
  - mogućnost automatskog pozicioniranja kursora na prvu sledeću praznu zonu
  - rukovanje komentarima
  - mogućnost višestruke selekcije
  - semantički upiti

- *run-time* analize
- realizacija istovremenog rada sa više programskih jezika
- konverzija iz tekstualnog formata u interni format drveta (*import* operacija)
- rad sa bibliotekama
- pozivanje (postojećeg) kompajlera.

Bibliografija

[1] International standard iec 1131-3, programmable controllers - part 3: Programming languages, iec. Technical report, 1993.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983. ISBN 0-201-00023-7.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[4] M. Anlauff, P. Kutter, and A. Pierantonio. Formal aspects of and development environments for montages. In M. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Workshops in Computing, Amsterdam, 1997. Springer.

[5] M. Anlauff, P. Kutter, A. Pierantonio, and L. Thiele. Generating an Action Notation Environment from Montages Descriptions. In P. Mosses and D. Watt, editors, *Proceedings of the 2nd International Workshop on Action Semantics (AS'99)*, number NS-99-3 in BRICS Notes Series, pages 1–42. University of Aarhus, Department of Computer Science, March 1999.

[6] Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, Joël Fillon, Didier Parigot, Claude Pasquier, and Claudio Sacerdoti Coen. SmartTools: a development environment generator based on XML technologies. In *XML Technologies and Software Engineering, Toronto, Canada, ICSE'2001, ICSE workshop proceedings.*, 2001.

[7] Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, Didier Parigot, and Claude Pasquier. SmartTools: A generator of interactive environments tools. *CC 2001*, pages 355–360, 2001.

[8] Rolf Bahlke and Gregor Snelting. The PSG system: from formal language definitions to interactive programming environments. *ACM Trans. Program. Lang. Syst.*, 8(4):547–576, 1986.

[9] Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter. The Pan language-based editing system. *ACM Trans. Softw. Eng. Methodol.*, 1(1):95–127, 1992.

[10] Elliot Berk. *JLex: A lexical analyzer generator for Java.* Department of Computer Science, Princeton University, 1997.

[11] E. Bjarnason. Applab - a laboratory for application languages. In *In Bendix et al. (Eds): Proceedings of NWPER'96, Nordic Workshop on Programming Environment Research, Aalborg*, pages pp 99–104, May 1996.

[12] E. Bjarnason and G. Hedin. Tool support for framework-specific language extensions. In *In Bosch and Mitchell (Eds.): Object-Oriented Technology. ECOOP'97 Workshop Reader, pp129-132. LNCS 1357, Springer Verlag*, 1997.

[13] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *SDE 3: Proceedings of the third ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 14–24, New York, NY, USA, 1988. ACM Press.

[14] Susan A. Dart, Robert J. Ellison, Peter H. Feiler, and Nico A. Habermann. Overview of software development environments, 1992.

[15] Valentin David, Akim Demaille, and Olivier Gournet. Attribute grammars for modular disambiguation. In *In the proceedings of IEEE 2nd International Conference on Intelligent Computer Communication and Processing.* Technical University of Cluj-Napoca, Romania, 1 - 2 September 2006.

[16] Michael L. Van de Vanter. The documentary structure of source code. *Information & Software Technology*, 44(13):767–782, 2002.

[17] Alan Dearle, Michael Oudshoorn, and Karen Wyrwas. An integrated approach to the generation of environments from formal specifications. *17th Annual Computer Science Conf. (Australian Computer Science Comm.)*, 16(1):217–228, 19–21 January 1994.

[18] Rick Decker. *Data Structures*. Prentice-Hall, Inc., 1989. ISBN 0-13-198813-1.

[19] Charles Donnelly and Richard Stallman. Bison, the yacc-compatible parser generator.

[20] V Donzeau-Gouge, B. Lang, and B. Mélèse. Practical applications of a syntax directed program manipulation environment. In *ICSE '84: Proceedings of the 7th international conference on Software engineering*, pages 346–354, Piscataway, NJ, USA, 1984. IEEE Press.

[21] Arnon D.S., Attali I., and Franchi-Zannettacci P. A document manipulation system based on natural semantics. *Mathematical and Computer Modelling, publisher Elsevier*, 25(4):37–56, February 1997.

[22] R. Farrow, T. J. Marlowe, and D. M. Yellin. Composable attribute grammars: support for modularity in translator design and implementation. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 223–234, New York, NY, USA, 1992. ACM Press.

[23] David B. Garlan and Philip L. Miller. Gnome: An introductory programming environment based on a family of structure editors. volume 9, pages 65–72, New York, NY, USA, 1984. ACM Press.

[24] Miroslav Hajdukovic and Zorica Suvajdzin. *Uvod u medjunarodni standard IEC 61131-3*. Fakultet tehnickih nauka, 2002., Pomocni udzbenik, 2002.

[25] Miroslav Hajdukovic, Zorica Suvajdzin, and Zarko Zivanov. Da li je znakovno editiranje programa neizbezno? In *XLVI Konferencija za ETRAN, Banja Vrucica - Teslic, 2002., tom III, 35-38*, pages 35–38, June 2002.

[26] Miroslav Hajdukovic, Zorica Suvajdzin, and Zarko Zivanov. Regularni editor. *INFO M, god. 1, sv. 3-4, Beograd, 2002., 46-54*, 3-4:46–54, 2002.

[27] Miroslav Hajdukovic, Zorica Suvajdzin, and Zarko Zivanov. Character oriented program editing - habit or necessity. *Novi Sad Journal of mathematics, Novi Sad*, 33(1):53–65, 2003.

[28] G. Hedin. Context-sensitive editing in Orm. Technical report, Proceedings of the Nordic Workshop on Programming Environment Research, Tampere, Finland., 1991. Also published in Proceedings of the Nordic Workshop on Programming Environment Research, Tampere, Finland. 1992.

[29] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF (reference manual). *SIGPLAN Not.*, 24(11):43–75, 1989. and revised edition, December 1992.

[30] Jan Heering and Paul Klint. Semantics of programming languages: a tool-oriented approach. *SIGPLAN Not.*, 35(3):39–48, 2000.

[31] Pedro Rangel Henriques, Maria João Varanda Pereira, Marjan Mernik, Mitja Lenic, Enis Avdicausevic, and Viljem Zumer. Automatic generation of language-based tools. *Electr. Notes Theor. Comput. Sci*, 65(3), 2002.

[32] Koen De Hondt. A generic, customisable, hybrid structure-oriented editor. In *Proceedings of the 1994 Groningen Student Conference on Computer Science, number CS-N9401 in Computing Science Notes*, pages pages 27–35, 1994.

[33] Scott E. Hudson. *CUP User's Manual.* Graphics Visualization and Usability Center Georgia Institute of Technology, 1995.

[34] Samuel Imriska. *C# CUP & Lex.* Distributed System Group Technical University Vienna, September 2003.

[35] International Standard ISO/IEC 14977. *Information technology - Syntactic metalanguage - Extended BNF*, first edition edition, 1996.

[36] Stephen C. Johnson. Yacc: Yet another compiler-compiler. AT&T Bell Laboratories, New Jersey, 1975.

[37] Tim Jones, , and Jim Welsh. Requirements for a generic, language-based diagram editor. Technical report, 1996.

[38] Ken Kennedy, Kathryn S. McKinley, and Chau-Wen Tseng. Interactive parallel programming using the ParaScope editor. *IEEE Transactions on parallel and Distributed Systems*, 2(3):329–341, July 1991.

[39] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, second edition*. Prentice Hall Software Series, second edition edition, 1988.

[40] Ivan Klajn and Milan Sipka. *Veliki recnik stranih reci i izraza*. Prometej, 2007.

[41] J. W. C. Koorn. GSE: a generic text and structure editor. Technical report, December 1992.

[42] J. W. C. Koorn and H. C. N. Bakker. Building an editor from existing components: An exercise in software re-use. Technical report, July 12 1993.

[43] Philipp W. Kutter and Alfonso Pierantonio. Montages specifications of realistic programming languages. *J. UCS*, 3(5):416–442, 1997.

[44] Barbara Staudt Lerner. Contrasting approaches of two environment generators: The Synthesizer Generator and Pan. Technical report 93-032, Computer Science Department, University of Massachusetts at Amherst, 1993.

[45] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. AT&T Bell Laboratories, New Jersey, 1975.

[46] Nazim H. Madhavji and Nikos Leoutsarakos. A dynamically self-adjusting structured editor. In *SIGSMALL '85: Proceedings of the 1985 ACM SIGSMALL symposium on Small systems*, pages 101–116, New York, NY, USA, 1985. ACM Press.

[47] Boris Magnusson, Mats Bengtsson, Lars-Ove Dahlin, Goran Fries, Anders Gustavsson, Gorel Hedin, Sten Minor, Dan Oscarsson, and Magnus Taube. An overview of the Mjolner/Orm environment:

Incremental language and software development. Technical Report LU-FS-TR:90-57. 12 pages., The Mjolner Group, Department of Computer Science, Lund University, Sweden, 1990. Also published in Proceedings of the 2nd International Conference TOOLS (Technology of Object-Oriented Languages and Systems), 1990.

[48] Boris Magnusson, Sten Minor, Gorel Hedin, Mats Bengtsson, Magnus Taube, Lars-Ove Dahlin, Dan Oscarsson, Goran Fries, Anders Gustavsson, and Par-Anders Aronsson. Mjolner/Orm user's guide (version 1.3). Technical Report LU-CS-IR:91-4, The Mjolner Group, Department of Computer Science, Lund University, Sweden, 1991.

[49] Linda McIver. Evaluating languages and environments for novice programmers. In L. Baldwin & R. Scoble J. Kuljis, editor, *Proceedings of the 14th Annual Workshop of the Psychology of Programming Interest Group*, pages 100–110, Brunel University, London, UK, 18–21 2002.

[50] John McLean. A formal method for the abstract specification of software. *J. ACM*, 31(3):600–627, 1984.

[51] Lambert G. L. T. Meertens, Steven Pemberton, and Guido van Rossum. The ABC structure editor - structure-based editing for the ABC programming environment. In *63*, page 19. Centrum voor Wiskunde en Informatica (CWI), ISSN 0169-118X, December 31 1992. AA (Department of Algorithmics and Architecture).

[52] Sten Minor. Interacting with structure-oriented editors. Technical report, Department of Computer Science, Lund University, Sweden, 1991. Also published in International Journal of Man-Machine Studies, 37, 1992.

[53] Sten Minor and Boris Magnusson. Using Mjolner Orm as a structure-based meta environment. pages 71–106, Orlando, FL, USA, 1996. Academic Press, Inc.

[54] Peter D. Mosses. Theory and practice of action semantics. In *MFCS: Symposium on Mathematical Foundations of Computer Science*, 1996.

[55] Peter D. Mosses. Casl for ASF+SDF users. In M. P. A. Sellink, editor, *ASF+SDF'97, Proc. 2nd Intl. Workshop on the Theory and Practice of Algebraic Specifications*, volume ASFSDF-97 of *Electronic Workshops in Computing*. British Computer Society, 1997.

[56] Peter D. Mosses. Action semantics and ASF+SDF (system demonstration). *Electronic Notes in Theoretical Computer Science*, 65(3):7, 2002.

[57] Katherine Murray. *First look 2007 Microsoft Office System*. Microsoft, 2007.

[58] Lisa Neal and Gerd Szwillus. Report on the CHI'90 workshop on structure editors. *SIGCHI Bull.*, 22(2):49–53, 1990.

[59] office.microsoft.com.

[60] Didier Parigot, Carine Courbis, Pascal Degenne, Alexandre Fau, Claude Pasquier, Joël Fillon, Christophe Held, and Isabelle Attali. Aspect and XML-oriented semantic framework generator: Smarttools. *Electr. Notes Theor. Comput. Sci*, 65(3), 2002.

[61] Vern Paxson. Flex, a fast scanner generator.

[62] Giuseppe Psaila and Stefano Crespi-Reghizzi. Adding semantics to xml. In *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, 1999.

[63] Vincent Quint and Irène Vatton. Making structured documents active. *Electronic Publishing-Origination, Dissemination, and Design*, 7(2):55–74, June 1994.

[64] J. Rekers. On the use of graph grammars for defining the syntax of graphical languages. Technical Report 94-11, Leiden University, March 1994.

[65] Thomas Reps and Tim Teitelbaum. The synthesizer generator. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 42–48, New York, NY, USA, 1984. ACM Press.

[66] D. Schmidt. Programming language semantics. In CRC Handbook of Computer Science, Allen Tucker, ed CRC Press, Boca Raton, FL, in press. Abridged version, ACM Computing Surveys, 1996.

[67] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, San Francisco, isbn 1-55860-442-1 edition, 2000.

[68] John N. Shutt. Survey of grammar models, 1996.

[69] Richard M. Stallman. Emacs the extensible, customizable self-documenting display editor. In *Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation*, pages 147–156, New York, NY, USA, 1981. ACM Press.

[70] Bernard Sufrin and Oege De Moor. Modeless structure editing, august 1999.

[71] Zorica Suvajdzin. Strukturni sintaksno vodjeni editor za programski jezik ST. Master's thesis, Faculty of Technical Sciences, University of Novi Sad, November 2000.

[72] Zorica Suvajdzin and Miroslav Hajdukovic. Towards program composing assistants. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers, PLC'05, pages: 142-147, Las Vegas, USA, 2005*, pages 142–147, June 2005.

[73] Zorica Suvajdzin and Miroslav Hajdukovic. Program composing assistant for novice programmers. In *The ASEE Mid-Atlantic Spring Conference 2006, Brooklyn NY*, April 2006.

[74] Zorica Suvajdzin and Miroslav Hajdukovic. A structure editor for the program composing assistant. *Computer Science and Information Systems, Volume 3, Number 1, June 2006., 65-76, Beograd,*, 3(1):65–76, June 2006.

[75] Zorica Suvajdzin, Miroslav Hajdukovic, Zarko Zivanov, and Stevan Odri. Strukturni sintaksno-vodjeni editor. In *YUINFO 2001, zbornik radova, Kopaonik*, March 2001.

[76] Zorica Suvajdzin, Miroslav Hajdukovic, Zarko Zivanov, and Stevan Odri. Translator sa programskog jezika ST na programski jezik C. In *YUINFO 2001, zbornik radova, Kopaonik*, March 2001.

[77] Tim Teitelbaum. The Cornell Program Synthesizer: A Tutorial Introduction. Technical Report TR79-381, Cornell University, Computer Science Department, Ithaca, NY, 1980.

[78] Tim Teitelbaum and Richard Chapman. Higher-order attribute grammars and editing environments. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 197–208, New York, NY, USA, 1990. ACM Press.

[79] Tim Teitelbaum and Thomas Reps. The Cornell program synthesizer: a syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, 1981.

[80] Mark A. Toleman and Jim Welsh. Systematic evaluation of design choices for software development tools. Technical report, 1998.

[81] Mark Anthony Toleman and To Margaret. *The Design Of The User Interface For Software Development Tools*. PhD thesis, Department of Computer Science, The University of Queensland, September 08 1996.

[82] S. Üsküdarli and T. B. Dinesh. The VAS formalism in VASE. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 140–147, Washington, September 3–6 1996. IEEE Computer Society Press.

[83] S. M. Üsküdarli. Generating visual editors for formally specified languages. In Allen L. Ambler and Takayuki Dan Kimura, editors, *Proceedings of the Symposium on Visual Languages*, pages 278–287, Los Alamitos, CA, USA, October 1994. IEEE Computer Society Press.

[84] M. van den Brand and C. Groza. The algebraic specification of annotated abstract syntax trees. Technical report, March 03 1994.

[85] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The Asf+Sdf Meta-environment: A component-based language development environment. *Lecture Notes in Computer Science*, 2027:365–370, 2001.

[86] Mark van den Brand, Alex Sellink, and Chris Verhoef. Generation of components for software renovation factories from context-free grammars. Technical report, January 19 1997.

[87] M.G.J van den Brand, T. Kuipers, L. Moonen, and P. Olivier. Implementation of a prototype for the new ASF+SDF Meta-Environment. *2nd International Workshop on the Theory and Practice of Algebraic Specifications*.

[88] Arie van Deursen and Peter D. Mosses. Asd: The action semantic description tools. In *AMAST'96, Proc. 5th Intl. Conf. on Algebraic Methodology and Software Technology, Munich*, volume 1101 of *Lecture Notes in Computer Science*, pages 579–582. Springer-Verlag, 1996.

[89] M. L. Van De Vanter. Practical language-based editing for software engineers. *Lecture Notes in Computer Science*, 896, 1995.

[90] Michael L. Van De Vanter. *User Interaction in Language-Based Editing Systems*. PhD thesis, EECS Department, University of California, Berkeley, Apr 1993.

[91] Michael L. Van De Vanter, Robert A. Ballance, and Susan L. Graham. Coherent user interfaces for language-based editing systems. Technical Report UCB/CSD 90/591, Computer Science Division (EECS), University of California, Berckley, California., Berkeley, CA, USA, July 1990.

[92] Michael L. Van De Vanter and Marat Boshernitsan. Displaying and editing source code in software engineering environments. *Second International Symposium on Constructing Software Engineering Tools (CoSET'2000)*, page 10 pages, June 5 2000.

[93] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, Amsterdam, 1997.

[94] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[95] David A. Watt. An extended attribute grammar for Pascal. *SIGPLAN Not.*, 14(2):60–74, 1979.

[96] Jan Willem, Cornelis Koorn, Academisch Proefschrift, and Door Jan Willem Cornelis Koorn. *Generating Uniform User-Interfaces For Interactive Programming Environments*. PhD thesis, June 19 1994.

[97] Terry A. Winograd. Muir: A tool for language design. Technical report, Department of Computer Science, Stanford University, Stanford, CA, USA, 1987.

[98] www.dot.net.com.

[99] www.eclipse.org.

Indeks

ABC structure editor, 21
abstract state machines, 15
action semantics, 15
Agora Structure Editor (ASE), 43
algebraic semantics, 15
Aloe generator programskih okruženja, 49
APPLAB, 50
apstraktno sintaksno drvo, 6, 8
ASF + SDF Meta-environment, 31
attribute grammar, 15, 53
axiomatic semantics, 53

BNF, 51

Centaur, 25
compiler generator, 15
Cornell Program Synthesizer, 19

definicija semantike, 51
definicija sintakse, 50
denotational semantics, 15, 30, 52

formalna specifikacija, 11, 50

Gem-Mex, 50
generatori programskih okruženja, 23
Generic Syntax-directed Editor (GSE), 38
GNOME okruženje, 49

hibridni editor, 16, 58
human-computer interaction, 56

interna reprezentacija, 8, 64
intuitivni korisnički interfejs, 57, 62

jezički-zasnovani sistemi, 1, 13, 55

language definer, 30
language prototyping, 13
Lex, 51

Mentor, 24
Muir, 45
MUPE-2, 50

natural semantics, 52

okruženje, 4
operational semantics, 15, 51
ORM Environment, 40

Pan, 35
ParaScope editor, 50
parser generator, 15
Programming System Generator, PSG, 29
programsko okruženje, 4

razvojno okruženje, 4

scanner generator, 15
selekcija, 10
sintaksni editor, 3, 16, 58
SmartTools, 34
strukturna transformacija, 64
strukturne operacije, 10
strukturne operacije, dodatne, 63
strukturne operacije, osnovne, 62
strukturni editor, 3, 16, 58
strukturni sintaksno-vođeni editor za programski jezik ST, 13, 69
strukturno-orijentisano okruženje, 6
Synthesizer Generator, 28

tekstualni editor, 3, 8, 16, 58

UQ*, 50
usmereno drvo, 64

Yacc, 51

Dodatak A

Rečnik pojmova

Za izradu rečnika pojmova korišćen je *Veliki rečnik stranih reči i izraza* [40].

A

abstract syntax tree apstraktno sintaksno drvo

annotation anotacija, zabeleška, primedba, napomena

aspect-oriented programming aspektno-orijentisano programiranje

attribute evaluation postupak izračunavanja atributa

attribute grammar

B

bootstrap

bottom-up uzlazni (od dole na gore)

breakpoint tačka prekida programa

browse pregledanje, pretraživanje

C

compile-time errors greške nastale u toku kompajliranja

customisability konfigurabilnost, svojstvo sistema koji korisnik može da menja i prilagođava svojim potrebama

customizer onaj koji vrši podešavanje sistema

D

debugger dibager

default podrazumevan

design dizajn, nacrt, oblikovanje proizvoda

designer dizajner, onaj koji se bavi dizajnom

desk calculator kalkulator

direct manipulation direktna manipulacija

display format format prikaza, format ispisa

domain specific language namenski jezik

E

editor designer dizajner editora, projektant editora

event driven vođena događajima iniciranih korisnikom

F

framework

G

general purpose opšte namene

generic generički, uopšten, zajednički

H

hierarchical window system hijerarhijski sitem prozora

highlighting bojenje teksta, isticanje posebnih osobina teksta

human-computer interaction interakcija čovek-računar

hyperlink link

I

identifier bindings

incremental parsing inkrementalno parsiranje

interactive language technology interaktivna jezička tehnologija

interface interfejs

intermediate language međujezik

K

kernel jezgro sistema

L

language-based operation jezički-zasnovana operacija

language-based system jezički-zasnovan sistem

language construct jezička konstrukcija, jezička struktura

language definer dizajner jezika, projektant jezika

language description author autor specifikacije jezika

language-driven operation jezički-vođena operacija

language prototyping brzi razvoj jezika (kroz iterativno razvijanje prototipova okruženja)

layout izgled, prikaz

linguistic support jezička podrška

linguistically-driven typography lingvistička tipografija

M

modelesness bez modalnosti, jednorežimski

N

namespace kategorija imena

P

phrase

placeholder

pretty-printer program koji vrši transformaciju interne strukturne reprezentacije programskog teksta u niz znakova

primitives primitive, osnovne funkcije

program database baza podataka programa

programming-in-the-large zadaci koji obuhvataju upravljanje projektom

programming-in-the-many zadaci koji obuhvataju upravljanje projektom i ljudima

programming-in-the-small zadaci koji obuhvataju editovanje i kompajliranje

proof checker alat za proveru korektnosti

Q

quantifier kvantifikator, simbol koji iskazuje broj ili količinu

R

redesign redizajn, postojeći dizajn u izmenjenom obliku

repositorium repozitorijum, mesto koje služi za odlaganje i čuvanje nečega

resource file konfiguraciona datoteka sa GUI informacijama

retargeted prilagođen, preusmeren

run-time analyzer alat koji vrši analizu u vreme izvršavanja

S

scope opseg važenja

scope rules pravila opsega važenja

selection selekcija, ono što je odabrano, označeno ili obeleženo

semantic functions funkcije koje vrše semantičku analizu

side-effect sporedni efekat, posledica

single-user jednokorsnički, za jednog korisnika

string niz znakova

strongly-typed strogo-tipizirano

structure operations strukturne operacije

suspend suspendovati, odgoditi važenje nečega

syntactical sugar sintaksno ulepšavanje dodatnim sintaksnim elementima

synthesize sintetizovati, sastaviti, spajati, vršiti sintezu

T

tag oznaka

template obrazac, šablon, mustra

text-formatting editor editor za formatiranje teksta

top-down silazni (od gore na dole)

translate translirati, prevoditi

type-checker program koji vrši proveru tipova

U

unparser program koji ima suprotnu namenu od parsera, i vrši transformaciju interne strukturne reprezentacije programskog teksta u niz znakova; još se zove i *pretty-printer*

unstructured operations nestrukturne operacije

update ažuriranje

user-friendly pristupačan

V

version control kontrola verzija

view prikaz, pogled, način prikazivanja

Dodatak B

Specifikacija programskog jezika *C*

```
%language   name="AnsiC"

/* SYMBOLS */
%symbol    name="IF"        string="if"        attr="keyword"
%symbol    name="ELSE"      string="else"      attr="keyword"
%symbol    name="WHILE"     string="while"     attr="keyword"
%symbol    name="DO"        string="do"        attr="keyword"
%symbol    name="FOR"       string="for"       attr="keyword"
%symbol    name="RETURN"    string="return"    attr="keyword"
%symbol    name="BREAK"     string="break"     attr="keyword"
%symbol    name="CONTINUE"  string="continue"  attr="keyword"

%symbol    name="STRUCT"    string="struct"    attr="keyword"
%symbol    name="UNION"     string="union"     attr="keyword"
%symbol    name="CONST"     string="const"     attr="keyword"
%symbol    name="ENUM"      string="enum"      attr="keyword"
%symbol    name="SIZEOF"    string="sizeof"    attr="keyword"

%symbol    name="EXTERN"    string="extern"    attr="keyword"
%symbol    name="STATIC"    string="static"    attr="keyword"
```

```
%symbol    name="PTR"       string="*"
%symbol    name="COMMA"     string=","   attr="separator"
%symbol    name="SEMICOLON" string=";"

%symbol    name="DOT"
           string="."
           attr="member_selector(Structure, Union)"

%symbol    name="AROW"
           string="->"
           attr="member_selector(Pointer(Structure),
                          Pointer(Union))"

%symbol    name="LEFT_CURLEY_BRACE"     string="{"
%symbol    name="RIGHT_CURLEY_BRACE"    string="}"

%symbol    name="LEFT_PARENTHESIS"      string="("
%symbol    name="RIGHT_PARENTHESIS"     string=")"

%symbol    name="LEFT_BRACKET"          string="["
%symbol    name="RIGHT_BRACKET"         string="]"


/* COMMENTS */
/* multi-line comment */
%comment    begin="//*"     end="*//"   nested="false"


/* REGULAR EXPRESSIONS */
%regexp name="letter"       exp="[a-zA-Z]"
%regexp name="digit"        exp="[0-9]"

%regexp name="octal_digit"  exp="[a-fA-F0-9]"
%regexp name="hexa_digit"   exp="[a-fA-F0-9]"
%regexp name="exponent"     exp="[Ee][+-]?[0-9]+"

%regexp name="decimal_int"  exp="{digit}+"
%regexp name="octal_int"    exp="0{octal_digit}+"
%regexp name="hexa_int"     exp="0[xX]{hexa_digit}+"
```

```
%regexp name="INT_LITERAL"  exp="{decimal_int}
                                 |{octal_int}
                                 |{hexa_int}"

%regexp name="float1" exp="{digit}+{exponent}"
%regexp name="float2" exp="{digit}*\.{digit}+({exponent})?"
%regexp name="float3" exp="{digit}+\.{digit}*({exponent})?"
%regexp name="float"  exp="{float1}|{float2}|{float3}"
%regexp name="FLOAT_LITERAL"    exp="{float}[Ff]"
%regexp name="DOUBLE_LITERAL"   exp="{float}"

%regexp name="CHAR_LITERAL"     exp="\'([^\'])+\'"

%regexp name="IDENTIFIER"
        exp="({letter}({letter}|{digit})){1,31}"

%regexp name="NULL" exp="NULL"


/* BASIC TYPES */
%data_type  name="VOID"
            string="void"
            literal="NULL"

%data_type  name="CHAR"
            string="char"
            literal="CHAR_LITERAL"
            conversion_to="INT FLOAT DOUBLE"

%data_type  name="INT"
            string="int"
            literal="INT_LITERAL"
            conversion_to="FLOAT DOUBLE"

%data_type  name ="FLOAT"
            string="float"
            literal="FLOAT_LITERAL"
            conversion_to="DOUBLE"
```

```
%data_type  name="DOUBLE"
            string="double"
            literal="DOUBLE_LITERAL"


/* DATA TYPE GROUPS */
%data_type_group    name="INTEGRAL_TYPES"
                    members="CHAR INT"

%data_type_group    name="FLOATING_TYPES"
                    members="FLOAT DOUBLE"

%data_type_group    name="ARITHMETIC_TYPES"
                    members="INTEGRAL_TYPES FLOATING_TYPES"


/* UNARY OPERATIONS */
%unary_operation    operand="POINTER(INTEGRAL_TYPES)"
                    ( operator="*"
                      result="INTEGRAL_TYPES"
                      result_lvalue="true" )
%unary_operation    operand="POINTER(FLOATING_TYPES)"
                    ( operator="*"
                      result="FLOATING_TYPES"
                      result_lvalue="true" )
%unary_operation    operand="POINTER(STRUCTURE)"
                    ( operator="*"
                      result="STRUCTURE"
                      result_lvalue="true" )
%unary_operation    operand="POINTER(UNION)"
                    ( operator="*"
                      result="UNION"
                      result_lvalue="true" )
%unary_operation    operand="POINTER(POINTER)"
                    ( operator="*"
                      result="POINTER"
                      result_lvalue="true" )
%unary_operation    operand="POINTER(ANY_TYPE)"
                    ( operator="*"
```

```
                    result="ANY_TYPE"
                    result_lvalue=FALSE )

%unary_operation    operand="ARITHMETIC_TYPES"
                  ( operator="+ -"
                    result="promoted_type(operand)" )
                  ( operator="!"
                    result="INT" )

%unary_operation    operand="INTEGRAL_TYPES"
                  ( operator="~"
                    result="promoted_type(operand)" )

%unary_operation    operand="POINTER"
                  ( operator="!"
                    result="INT" )

%unary_operation    operand="ANY_TYPE"
                  ( operator="&"
                    operand_lvalue="true"
                    result="POINTER(operand)" )


/* BINARY OPERATIONS */
%binary_operation   operand="ARITHMETIC_TYPES"
    ( operator="+ - * /"
      operand="ARITHMETIC_TYPES"
      result="promoted_type(operands)" )
    ( operator="< > <= >= == != && ||"
      operand="ARITHMETIC_TYPES"
      result="LOGICAL" )

%binary_operation   operand="INTEGRAL_TYPES"
    ( operator="% & | ^"
      operand="INTEGRAL_TYPES"
      result="promoted_type(operands)" )
    ( operator="<< >>"
      operand="INTEGRAL_TYPES"
      result="promoted_type(first_operand)" )
```

```
%binary_operation   operand="POINTER "
    ( operator="-"
      operand="SAME_TYPE"
      result="INTEGRAL_TYPES")
    ( operator="-"
      operand="INTEGRAL_TYPES"
      result="POINTER")
    ( operator="< > <= >= == != && ||"
      operand="SAME_TYPE"
      result="LOGICAL")
    ( operator="== !="
      operand="NULL"
      result="LOGICAL" )
    ( operator="== !="
      operand="POINTER(void)"
      result="LOGICAL" )

%binary_operation   operand="POINTER(object in a array)"
    ( operator="+"
      operand="INTEGRAL_TYPES"
      result="type(first_operand)" )

%binary_operation   operand="POINTER(void)"
    ( operator="== !="
      operand="POINTER"
      result="LOGICAL" )

%binary_operation   operand="NULL"
    ( operator="== !="
      operand="POINTER"
      result="LOGICAL" )


/* ASSIGNMENT OPERATIONS */
%assignment_operation   left="ANY_TYPE" operator="= *="


/* NAMESPACES */
```

```
%namespace  OBJECT     /* VARIABLE, PARAMETER */
%namespace  FUNCTION
%namespace  ENUM_CONSTANT
%namespace  TAG
%namespace  FIELD

/* LVALUE */
%lvalue     namespace="OBJECT"
            type="ARITHMETIC_TYPES
                 | Structure
                 | Union
                 | Pointer"


/* PRODUCTIONS */

compilation_unit
    :   declaration+
            {   %start_symbol;
                %compilation_unit;
                %new_scope;
            }
    ;

declaration
    :   declaration_begining declaration_continuation
            {   %new_symbol;
                symbol.type(INT);
                symbol.attr(GLOBAL);
                %new_line;
            }
    ;

declaration_begining
    :   declaration_specifiers?, pointer*, declarator_id
    ;

pointer
    :   POINTER_MARK
```

```
            {   %new_type(Pointer);
                type.base_type(symbol.type);
                symbol.type(type);
            }
    ;

declarator_id
    :   IDENTIFIER
            {   symbol.name(IDENTIFIER);    }
    ;

declaration_continuation
    :   var_decl
    |   function_decl
    |   function_def
    ;

var_decl
    :   array_declarator?, init?, SEMICOLON
            {   symbol.namespace(OBJECT);
                init.type(symbol);
            }
    ;

array_declarator
    :   LEFT_BRACKET, constant, RIGHT_BRACKET
            {   constant.type(INTEGRAL_TYPES);
                %new_type(Array);
                type.size(constant);
                type.base_type(symbol.type);
                symbol.type(type);
            }
    ;

init
    :   ASSIGN, initializer
            {   initializer.type(symbol.type); }
    ;
```

```
initializer
    :   asignment_expression
            {   @context(symbol.type(ARITHMETIC_TYPES
                                    | Pointer));   }

    |   LEFT_CURLEY_BRACE, initializer_list,
        RIGHT_CURLEY_BRACE
            {   @context(symbol.type(Structure
                                    | Union
                                    | Array));   }
    ;

initializer_list
    :   (initializer, COMMA)+
    ;

function_decl
    :   parameters, SEMICOLON
            {   symbol.namespace(FUNCTION);
                symbol.attr(DECLARATION);
                %new_scope;
            }
    ;

function_def
    :   parameters, function_body
            {   %new_scope;
                %get_signature(parameters);
            }
    ;

function_body
    :   LEFT_CURLEY_BRACE, compound_body, RIGHT_CURLEY_BRACE
            {   %new_line;  }
    ;

parameters
    :   LEFT_PARENTHESIS, params, RIGHT_PARENTHESIS
    ;
```

```
params
    :   parameter_list
    |   void
    ;

void
    :   VOID
    ;

parameter_list
    :   (parameter_declaration, COMMA)+
    ;

parameter_declaration
    :   type_specifier, pointer*, declarator_id
            {   %new_symbol;
                symbol.namespace(OBJECT);    }
    ;

declaration_specifiers
    :   storage_class_specifier, type_specifier,
        type_qualifier
            {   symbol.type(type_specifier);
                symbol.attr(storage_class_specifier,
                            type_qualifier);
            }
    ;

storage_class_specifier
    :   EXTERN          { symbol.attr(EXTERN); }
    |   STATIC          { symbol.attr(STATIC); }
    ;

type_specifier
    :   VOID            { symbol.type(VOID); }
    |   CHAR            { symbol.type(CHAR); }
    |   INT             { symbol.type(INT); }
    |   FLOAT           { symbol.type(FLOAT); }
```

```
    |   DOUBLE          { symbol.type(DOUBLE); }
    |   enum_specifier
    |   struct_or_union_specifier
    ;

enum_specifier
    :   ENUM, enum_tag, enum_list
            {   %new_symbol;
                %new_type(Enumeration);
                symbol.type(type);
                %new_line;
            }
    ;

enum_tag
    :   IDENTIFIER
            {   symbol.name(ID):
                symbol.attr(ENUM_CONSTANT);
            }
    ;

enum_list
    :   LEFT_CURLEY_BRACE, enumerator_list, RIGHT_CURLEY_BRACE
    ;

enumerator_list
    :   (enumerator, COMMA)+
    ;

enumerator
    :   enumerator_name, enum_const_exp?
    ;

enumerator_name
    :   IDENTIFIER
            {
                %new_symbol;
                symbol.name(IDENTIFIER);
                symbol.type(INT);
```

```
                symbol.namespace(ENUM_CONSTANT);
                type.constant(IDENTIFIER);
            }
    ;

enum_const_exp
    :   ASSIGN, integer_constant
            {   integer_constant.type(INT);  }
    ;

struct_or_union_specifier
    :   STRUCT, IDENTIFIER
           {   lookup symbol(ID, STUCTURE); }

    |   UNION, IDENTIFIER
           {   lookup symbol(ID, UNION);   }

    |   STRUCT, struct_id, struct_list
           {   symbol.namespace(TAG);
               new_type(Structure);
               symbol.type(type);
               type.attr(COMPLETE_TYPE);
           }

    |   UNION, struct_id, struct_list
           {   symbol.namespace(TAG);
               new_type(Union);
               symbol.type(type);
               type.attr(COMPLETE_TYPE);
           }
    ;

struct_id
    :   IDENTIFIER
            {   %new_symbol;
                symbol.name(ID);
            }
    ;
```

```
struct_list
    :   LEFT_CURLEY_BRACE, struct_declaration_list,
        RIGHT_CURLEY_BRACE
            {   %new_scope;    }
    ;

struct_declaration_list
    :   struct_declaration+
    ;

struct_declaration
    :   type_specifier?, struct_declarator_list, SEMICOLON
            {   %new_symbol;
                symbol.namespace(FIELD);
            }
    ;

struct_declarator_list
    :   {struct_declarator, COMMA}+
    ;

struct_declarator
    :   declaration_specifiers?, pointer*, declarator_id,
        array_declarator?
            {   declaration_specifiers.type(COMPLETE_TYPE);
                type.field(declarator_id); }
    ;


    /* STATEMENTS */

statement
    :   compound_statement
    |   expression_statement
    |   selection_statement
    |   iteration_statement
    |   jump_statement
    ;
```

```
compound_statement
    :   LEFT_CURLEY_BRACE, compound_body, RIGHT_CURLEY_BRACE
            {   %new_scope;    }
    ;

compound_body
    :   declaration_list?, statement_list?
            {   %indent;    }
    ;

declaration_list
    :   local_declaration+
    ;

local_declaration
    :   declaration_begining, var_decl
            {   %new_symbol;
                symbol.attr(LOCAL);
                %new_line;
            }
    ;

statement_list
    :   statement+
            {   %new_line;  }
    ;

expression_statement
    :   expression?, SEMICOLON
    ;

selection_statement
    :   if_statement
    ;

if_statement
    :   IF, condition, statement, else_stmt?
    ;
```

```
condition
    :   LEFT_PARENTHESIS, expression, RIGHT_PARENTHESIS
             { expression.type(ARITHMETIC_TYPES
                                | Pointer); }
    ;

else_stmt
    :   ELSE, statement
    ;

iteration_statement
    :   while_statement
    |   do_statement
    |   for_statement
    ;

while_statement
    :   while_condition, statement
            {   %new_context(ITERATION);   }
    ;

do_statement
    :   DO, statement, while_condition, SEMICOLON
            {   %new_context(ITERATION);   }
    ;

while_condition
    :   WHILE, condition
    ;

for_statement
    :   FOR, for_condition, statement
            {   %new_context(ITERATION);   }
    ;

for_condition
    :   LEFT_PARENTHESIS, expression_statement,
        end_condition, expression?, RIGHT_PARENTHESIS
    ;
```

```
end_condition
    :   expression_statement
            {   end_condition.type(ARITHMETIC_TYPES
                                  | Pointer);   }
    ;

jump_statement
    :   CONTINUE, SEMICOLON
            {   @context(ITERATION);   }

    |   BREAK, SEMICOLON
            {   @context(ITERATION);   }

    |   RETURN, expression?, SEMICOLON
            {   expression.type(symbol);    }
    ;


    /* EXPRESSIONS */

primary_expression
    :   var
    |   function_call
    |   constant
    |   parenthesised_exp
    ;

var
    :   IDENTIFIER
            { %lookup_symbol(IDENTIFIER, OBJECT); }
    ;

function_call
    :   function_call_name arguments
            {   %get_signature
                    (arguments, parameters;
                     argument_expression_list,
                        parameter_list(void);
```

```
                    expression, parameter_declaration);
            }
    ;

function_call_name
    :   IDENTIFIER
            {   %lookup_symbol(FUNCTION);    }
    ;

arguments
    :   LEFT_PARENTHESIS, argument_expression_list?,
        RIGHT_PARENTHESIS
    ;

argument_expression_list
    :   (expression, COMMA)+
    ;

parenthesised_exp
    :   LEFT_PARENTHESIS, expression, RIGHT_PARENTHESIS
            { root.type(expression); }
    ;

unary_expression
    :   primary_expression

    |   SIZEOF, primary_expression
            { root.type(INT); }

    |   SIZEOF, parenthesized_type
            { root.type(INT); }

    |   ADDRESS_OP, cast_expression
    |   INDIRECTION_OP, cast_expression
    |   UNARY_PLUS_OP, cast_expression
    |   UNARY_MINUS_OP, cast_expression
    |   COMPLEMENT_OP, cast_expression
    |   LOGICAL_NOT_OP, cast_expression
    ;
```

```
parenthesized_type
    :   LEFT_PARENTHESIS, type_name, RIGHT_PARENTHESIS
            { root.type(type_name); }
    ;

type_name
    :   specifier_qualifier_list, abstract_declarator?
    ;

cast_expression
    :   parenthesized_type, unary_expression
            { root.type(parenthesized_type); }
    ;

expression
    :   unary_expression
    |   cast_expression
    ;

binary_expression
    :   expression (BINARY_OPERATOR expression)#
    ;

assignment_expression
    :   cast_expression
    |   unary_expression ASSIGNMENT rhs
    ;

rhs
    :   unary_expression
    |   binary_expression
    ;

constant
    :   INT_LITERAL
    |   CHAR_LITERAL
    |   FLOAT_LITERAL
    |   DOUBLE_LITERAL
```

```
|   IDENTIFIER
        {   %lookup_symbol(ENUM_CONSTANT);   }
;
```