ŠOLSKI CENTER KRANJ
VIŠJA STROKOVNA ŠOLA
INFORMATIKA

DIPLOMSKO DELO

Kranj, maj 2023 **Jan Čotar**

ŠOLSKI CENTER KRANJ
VIŠJA STROKOVNA ŠOLA
INFORMATIKA

DIPLOMSKO DELO

Kranj, maj 2023 **Jan Čotar**

ŠOLSKI CENTER KRANJ
VIŠJA STROKOVNA ŠOLA
INFORMATIKA

Diplomsko delo višjega strokovnega izobraževanja

RAZVOJ TOLMAČA PROGRAMSKEGA JEZIKA S SLOVENSKO SINTAKSO

Avtor: Jan Čotar
Ime podjetja: Generali zavarovalnica d. d.
Mentor v podjetju: Črt Brezovar, mag. org. inf.
Mentor v šoli: Jan Robas, mag. inž. rač. in inf.
Lektorica: Tina Praček, univ. dipl. sloven. in soc. kult.

Kranj, maj 2023

ZAHVALA

Zahvaljujem se vsem, ki so mi pri ustvarjanju diplomskega dela kakorkoli pomagali. Hvala partnerici, ki mi je stala ob strani in me bodrila v neprespanih nočeh. Hvala mentorju v šoli, Janu Robasu, mag. inž. rač. in inf., za izkazano zaupanje in usmeritve. In ne nazadnje hvala tudi mentorju v podjetju, Črtu Brezovarju, mag. org. inf., ki je dosledno, korektno in profesionalno sodeloval pri ustvarjanju diplomskega dela.

POVZETEK

Programski jezik je način sporazumevanja človeka s strojem, sintaksa programskih jezikov pa je večinoma angleška. V diplomskem delu smo raziskali osnovna načela delovanja programskih jezikov, spoznanja pa uporabili za izdelavo novega programskega jezika, čigar sintaksa je slovenska. Programski jeziki so v osnovi zgolj zaporedje znakov, ki je podvrženo pravilom skladnje in so kot taki računalniškemu procesorju nerazumljivi. Peljati jih moramo skozi procese, ki jih spremenijo v računalniku razumljivo strukturo. Ti procesi so v glavnem prevajanje in tolmačenje. Jezik, ki smo ga razvili, omogoča osnovne funkcionalnosti programiranja. Zanj smo v tehnologiji C# razvili tolmač, ki je prosto dostopen na internetu.

Ključne besede: slovenščina, programski, jezik, tolmač, interpreter, parser, leksikalna analiza, semantična analiza

ABSTRACT

Programming languages are a medium for communication between a person and a machine and their syntax is mostly English. In diploma we have studied the basic principles of programming languages and used the new acquired knowledge to develop a brand new programming language with Slovene syntax. Programming languages are but a sequence of characters that are subject to syntax rules and are as such unintelligible to a computer processor. They have to be transformed to a structure that a computer can understand, which is usually achieved through compilation or interpretation. The language that we created provides essential programming functionalities. The interpreter is developed in C# and is freely accessible on the internet.

Keywords: Slovene, programming, language, interpreter, parser, lexical analysis, semantic analysis

Kazalo vsebine

Kazalo slik

Kazalo tabel

Grafikoni

SEZNAM KRATIC IN SIMBOLOV

Kratica/simbol	Razlaga pomena kratice/simbola
AST	Abstraktno sintaksno drevo (angl. Abstract Syntax Tree)
EBNF	Extended Backus-Naur form
IEC	Mednarodna komisija za elektrotehniko (angl. International Electrotechnical Commision)
ISO	Mednarodna organizacija za standardizacijo (angl. International Organization for Standardization)
JIT	Just-In-Time Compiler
JVM	Java Virtual Machine

1 UVOD

Skozi zgodovino računalniškega razvoja naj bi bilo ustvarjenih skoraj 9000 programskih jezikov (povzeto po: Upson, 2022). Koliko od njih jih še danes uporabljamo, je težko reči, lahko pa naštejemo vsaj najbolj popularne. To so JavaScript, HTML/CSS, SQL, Python ipd. C# je na osmem mestu (https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/). Cilj diplomskega dela ni ustvariti programski jezik, ki bi lahko tekmoval z najpopularnejšimi in najzmogljivejšimi, temveč približati bralcu delovanje in koncepte jezikov skozi, sicer omejeno funkcionalen, praktičen izdelek. Poleg tega pa je sintaksa ustvarjenega jezika slovenska, zaradi česar izstopa med množico angleških.

V diplomskem delu smo ustvarili lasten programski jezik s slovensko sintakso, ki je v veliki meri podobna C# sintaksi. Poimenovali smo ga JanC. Vendar jezik kot takšen še ni dovolj, da ga lahko računalnik razume. Napišemo ga lahko tudi na papir. Poleg jezika moramo ustvariti tudi neke vrste abstraktno napravo ali program, ki bo ta jezik razumel. V splošnem lahko ustvarimo prevajalnik, ki jezik prevede v strojno kodo, ali pa tolmač, ki jezik razume in ga zna izvajati. Odločili smo se za tolmač, saj je njegova izdelava lažja. Za prevajanje bi namreč morali poznati tudi strojni jezik. Kako smo ga izdelali in iz česa smo izhajali, bo v diplomskem delu podrobno opisano.

1.1 OPREDELITEV KLJUČNIH POJMOV

LEKSIKALNI ANALIZATOR

Leksikalni analizator je prvi proces v prevajanju programskega jezika. Niz znakov razdeli v lekseme in jih dodeli žetonom.

LEKSEM

Leksem je enota jezika, ki nosi neki pomen. Je širši od besede, saj so tudi stalne besedne zveze leksemi (povzeto po: Vidovič Muha, 2021).

ŽETON

Žeton (angl. token) predstavlja kategorijo leksema. Nekateri žetoni lahko sprejmejo samo en določen leksem, nekateri pa zaključeno množico njih.

SINTAKTIČNI ANALIZATOR

Sintaktični analizator s pomočjo žetonov preveri skladnjo jezika in zgradi abstraktno sintaksno drevo, ki predstavlja potek in pomen programa.

ABSTRAKTNO SINTAKSNO DREVO

Abstraktno sintaksno drevo je izvorna koda, predstavljena v obliki drevesa. Izpuščene so vse značilnosti izvorne kode, ki programu ne dajejo nobenega pomena.

SEMANTIČNI ANALIZATOR

Semantični analizator preveri pomen programa. Pregleda ujemanje podatkovnih tipov, mesta uporabe spremenljivk ipd.

TOLMAČ

Tolmač namesto končnega prevoda v strojno kodo izvede program, ki je v obliki vmesnega jezika ali abstraktnega sintaksnega drevesa.

1.2 NAMEN DIPLOMSKEGA DELA

Namen diplomskega dela je omogočiti programiranje preprostejših programov v jeziku s slovensko sintakso. Tolmač za jezik je v obliki knjižnice dosegljiv komurkoli na svetu, ki uporablja C# in ima dostop do upravljalnika paketov NuGet ali račun na platformi za verzioniranje kode GitHub. Tolmač se lahko integrira s katerimkoli projektom, razvitim v tehnologiji C#.

Programski jezik, ki uporablja slovensko sintakso, je primeren tudi za poučevanje mladostnikov o osnovah programiranja.

Splošen namen diplomskega dela je osvetliti razumevanje delovanja programskih jezikov.

1.3 CILJI DIPLOMSKEGA DELA

- Ustvariti programski jezik s slovensko sintakso;
- v tem jeziku omogočiti osnovne funkcionalnosti (delo s spremenljivkami, vejitve, zanke, izpis, vnos, računanje osnovnih matematičnih operacij ipd.);
- dodati možnost razširitev;
- v C# razviti tolmač za jezik in

- tolmač izvoziti v obliki knjižnice, ki bo omogočala integracijo programskega jezika s preostalimi projekti v okolju .NET.

1.4 METODE DELA

Za doseganje ciljev smo uporabili metode študija literature in spletnih virov ter obstoječih programskih jezikov. Raziskali smo koncepte delovanja tolmačev in spoznanja uporabili pri izdelavi lastnega tolmača. Njegovo delovanje smo tudi testirali.

1.5 PREDSTAVITEV PODJETJA

Generali zavarovalnica d. d. je del skupine Generali, ki je ena izmed največjih svetovnih ponudnikov zavarovanj in upravljanja premoženja. Skupina Generali deluje že 190 let, ustanovljena pa je bila v Trstu. Skupaj ima približno 75.000 zaposlenih in posluje v 50 državah (https://www.generali.si/skupina-generali).

Avtor diplomskega dela je bil v podjetju zaposlen kot strokovni sodelavec za razvoj produktov in tehnoloških rešitev. Njegovo vsakdanje delo je bilo kodiranje poslovne logike in integriranje te z obstoječimi aplikacijami, ki jih zavarovalnica uporablja za sklenitev zavarovanj. Poslovna logika se je pisala v za to namenskih programskih orodjih s svojim programskim jezikom.

Na podoben način bi lahko razvili generično C# aplikacijo za sklepanje zavarovanj in uporabnikom aplikacije prepustili programiranje poslovne logike. Za to bi jim ponudili naš programski jezik. Izračun premije bi tako lahko v celoti napisali v jeziku JanC. Preprost primer za takšno aplikacijo je predstavljen v poglavju Primer uporabe jezika JanC. Prav tako bi lahko v jeziku JanC modelirali zavarovalniške produkte ali pa bi ga uporabili za aktuarske in informativne izračune. Dodatno bi razvili namenski grafični oz. spletni vmesnik, ki bi ponujal uporabniku prijazno razvojno okolje za programe v JanC.

2 SPLOŠNO O PROGRAMSKIH JEZIKIH

Programski jezik je način sporazumevanja človeka z računalnikom. Je neke vrste vmesni člen, ki prevaja človekove napotke v računalniku razumljive ukaze.

Programski jezik, tako kot naravne jezike, označujeta njegova sintaksa in semantika. Sintaksa je skladnja jezika oz. pravila, kako se enote jezika povezujejo med seboj. Semantika pa določa tako sestavljenim besedam neki točno določen pomen (povzeto po: Sebesta, 2016). Sintakso je precej lažje prepoznati kot semantiko. Ponazorimo na vsakdanjem primeru. Če vam nekdo pove nekaj v jeziku, ki ga ne razumete, mogoče lahko določite jezik, pomena pa nikakor ne.

Sintaksa še najbolj zahtevnega in kompleksnega programskega jezika je v primerjavi s katerimkoli naravnim jezikom popolnoma preprosta. Njegova semantika pa je praviloma nedvoumna, medtem ko je pri naravnem jeziku lahko neko sporočilo narobe razumljeno.

V splošnem lahko izvedbe programskih jezikov delimo na prevajane, tolmačene in hibridne. Programi, napisani v prevajanem jeziku, morajo biti pred izvajanjem prevedeni v strojni jezik, tolmačeni pa se izvajajo neposredno med izvajanjem programa, ki jih tolmači. Zaradi tega jih lahko hitreje zaženemo, so pa med samim izvajanjem počasnejši od prevedenih.

Tolmačenje je od 10- do 100-krat počasnejše kot izvajanje prevedenih programov, glavni razlog pa se skriva v vsakokratnem dekodiranju visokonivojskega jezika, ki je precej bolj zapleten kot strojni jezik (povzeto po: Sebesta, 2016). V nadaljevanju so opisane bistvene značilnosti izvedb.

2.1 PREVAJANJE

Izvorna koda je prevedena neposredno v strojni jezik. Ta je množica ukazov, ki jih razume procesor, in so praviloma najbolj elementarni možni ukazi. Ker so strojni jeziki odvisni od arhitekture procesorjev, mora biti tako preveden program ciljan na izbrano platformo. Če želimo program izvajati na drugi platformi, moramo izvorno kodo ponovno prevesti v

strojni jezik tega sistema. Preveden program je z lahkoto prenosljiv, upoštevaje, da se vedno uporablja na istem tipu platforme. Preveden program se tudi najhitreje izvaja, saj so vsi njegovi ukazi strojni opremi neposredno znani. Takšnemu načinu v angleščini pravimo Ahead-Of-Time Compilation ali predčasno prevajanje.

2.2 ČISTO TOLMAČENJE

V primeru čistega tolmačenja potrebujemo že preveden program, ki zna tolmačiti neki programski jezik. Izvorna koda se analizira in izvaja sproti, kar pomeni, da je začetek programa izveden, še preden je njegov konec sploh analiziran. Vsak programski stavek se analizira posebej in pripravi za neposredno izvajanje. Takšna koda ni prevedena, temveč so njeni visokonivojski ukazi preslikani v pripadajoče procedure v programu ali sistemu, ki jezik tolmači. Rečemo lahko, da dejansko izvajanje ustvarja tolmač, izvorni jezik pa ga zgolj krmili in mu podaja parametre.

Analiza izvorne kode v ozadju poteka na isti ali podoben način kot pri prevajanju, le da je tukaj celoten proces ponovljen za vsak ukaz posebej. Prednost tolmačenja je hiter zagon programa in lažja prenosljivost kot pri prevajanem jeziku, saj tu ne ciljamo arhitekture, toda program oz. tolmač. Seveda pa mora biti tolmač ponovno neki program, ki je prilagojen za vsako procesorsko arhitekturo posebej.

Za izvajanje programskega jezika moramo prej ali slej imeti prevajalnik, ki prevede izvorno kodo v strojno kodo, ali tolmač izvorne kode, ki je že preveden v strojno kodo (povzeto po: Mogensen, 2017).

2.3 HIBRIDNI NAČIN

Hibridni način je kombinacija prevajanja in tolmačenja. Izvorna koda se najprej v celoti analizira tako kot pri prevajanju, le da se na koncu ne prevede neposredno v strojni jezik, temveč v določen vmesni jezik, ki je nekje med izvorno visokonivojsko kodo in strojnim jezikom. Izvajanje kode je na koncu lahko prepuščeno tolmaču takega vmesnega jezika ali pa je z metodologijo JIT (angl. Just-In-Time) sproti med izvajanjem prevajana v strojno kodo. Takšen način uporabljata tudi jezika Java in C# oz. vse tehnologije .NET. Vendar pa

tudi tu izvedbe niso tako nujno ločene. JVM (angl. Java Virtual Machine), ki je odgovorna za izvajanje Java kode, ima tudi na koncu zanimivo kombinacijo tolmačenja in prevajanja. Prevedejo se namreč samo tisti deli kode, ki so pogosto uporabljeni, preostali pa so tolmačeni.

Hibridni način poskuša s pravo kombinacijo zakrpati luknje osnovnih dveh izvedb. Zaradi prevoda v vmesni jezik je koda hitreje izvajana, saj je vmesni jezik preprostejši za prevajanje v strojnega kot visokonivojski. Po drugi strani pa ni ciljana arhitektura procesorja, temveč virtualni računalnik ali okolje, ki bo to kodo prevajalo ali tolmačilo, kar ga dela bolj prenosljivega.

Pri tolmačenju jezika mora tolmač pred izvedbo neke operacije preveriti, katero operacijo bo sploh izvedel, ter zatem preveriti še vse njene argumente in njihovo ujemanje z zahtevami. Šele nato lahko izvede samo operacijo. V neki zanki torej vsakokrat znova preverja zahteve. Tako se več časa ukvarja z ugotavljanjem, kaj narediti, kot pa z izvajanjem operacije. Prevajani jezik pa samo pri prevodu preveri vse zahteve, med izvajanjem pa zgolj še izvaja operacije (povzeto po: Mogensen, 2017).

2.4 GENERACIJE PROGRAMSKIH JEZIKOV

Programski jeziki so se skozi zgodovino razvijali in danes jih lahko umestimo v več generacij. Vsaki generaciji so dodane nove funkcionalnosti, predvsem pa se skozi njih sintakse jezikov oddaljujejo od strojnega jezika in približujejo človeku prijaznejšim.

PRVA GENERACIJA

V prvo generacijo spadajo strojni jeziki. To je jezik, ki ga razume procesor. Od njegove arhitekture pa je odvisna sintaksa strojnega jezika. Takšen jezik je sestavljen samo iz števk 0 in 1. Programer, ki uporablja strojni jezik, mora poznati sintakso vseh procesorskih ukazov. Poleg tega pa lahko napiše samo en ukaz naenkrat. Že precej preprost program za kopiranje neke vrednosti je v takšnem jeziku dolg nekaj vrstic.

DRUGA GENERACIJA

Sem uvrščamo zbirne jezike. So precej podobni strojnim in prav tako lahko izvršijo samo en procesorski ukaz naenkrat. Vsebujejo pa že nekaj abstrakcije.

Njihova najbolj opazna značilnost je na primer zamenjava neposrednih numeričnih pomnilniških naslovov s človeku bolj berljivimi besedami ali imeni spremenljivk. Koda na ta način postane bolj berljiva in lažja za vzdrževanje (povzeto po: Sebesta, 2016).

TRETJA GENERACIJA

Jezikom tretje generacije pravimo visokonivojski jeziki, ker v veliki meri skrijejo značilnosti strojne opreme. Vsi jeziki naslednjih generacij so visokonivojski. Za razliko od prve in druge generacije pri pogledu na takšen jezik arhitektura procesorja ni več znana. V en ukaz, ali pravilneje v en stavek, lahko spravimo več strojnih ukazov in tako je razvoj programske opreme precej hitrejši. Tak jezik je tudi bolj berljiv. Primeri jezikov: Java, C# in Python.

ČETRTA GENERACIJA

V to generacijo umeščamo predvsem domensko specifične jezike. To so jeziki, ki niso ustvarjeni za splošen namen, temveč rešujejo specifične težave nekega področja. Primeri jezikov: SQL in R.

PETA GENERACIJA

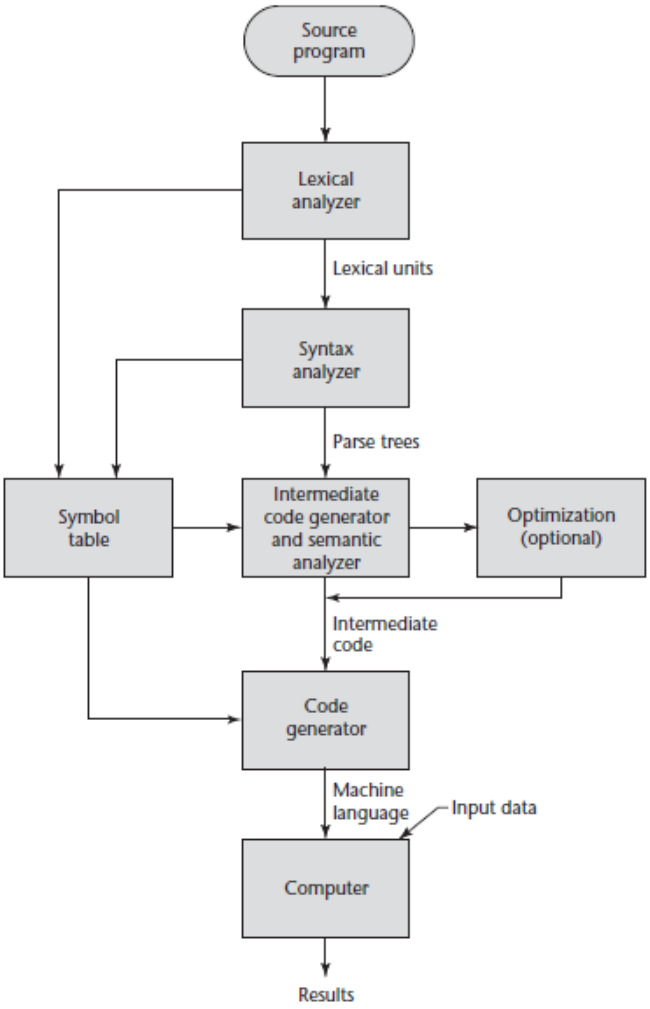
Ti jeziki se uporabljajo v kombinaciji z umetno inteligenco. Programi, ki so razviti v jeziku iz pete generacije, so namenjeni reševanju nalog, brez potrebe po pisanju specialnih algoritmov za razreševanje takšnih nalog. Program jo mora znati rešiti sam. Primeri jezikov: Prolog in OPS5.

ŠESTA GENERACIJA

Jeziki v šesti generaciji namesto pisne kode, kot jo poznamo, uporabljajo vizualne elemente, ki jih medsebojno povezujemo. Tako je neka zanka preprosto element, ki ga povlečemo iz nabora elementov in spustimo v okno s t. i. vizualno kodo. Primer jezika: Bubble.

2.5 DELOVANJE PROGRAMSKIH JEZIKOV

Procesi, ki se morajo odviti, da se program, napisan v nekem programskem jeziku, sploh lahko izvede, so na konceptualni ravni bolj ali manj enaki za vse vrste jezikov. Izvorna koda je zgolj zaporedje znakov, ki jih nato postopoma spremenimo v preveden ali tolmačen program. Čeprav smo v diplomskem delu ustvarili tolmačen jezik, si vseeno podrobneje oglejmo proces prevajanja jezika. Ta je namreč najzahtevnejši in z njim bomo dobili dober vpogled v delovanje programskih jezikov. Shema na sliki 1 predstavlja celoten proces prevajanja.



Slika 1: Shema procesa prevajanja jezika

Vir: Sebesta, 2016, 48

2.5.1 Leksikalni analizator

Leksikalni analizator sprejme niz znakov in jih razdeli v klasificirane žetone. Glavni namen leksikalne analize je razbremenitev dela sintaktične analize (povzeto po: Mogensen, 2017).

Izvorno kodo v obliki niza najprej prevzame leksikalni analizator (angl. lexer). Ta preveri vsak znak posebej in jih po vnaprej določenih pravilih združi v t. i. lekseme. Leksemi so na primer ključne besede (int, double itd.), operatorji in ločila (oklepaji, podpičja ipd.). Komentarji in presledki v izvorni kodi so že v tej fazi izločeni, saj so za kasnejše faze prevajanja neuporabni.

Leksemi so razvrščeni v skupine, ki jih poimenujemo oz. označimo z žetoni. Žeton tako predstavlja kategorijo leksema (povzeto po: Sebesta, 2016).

Poglejmo si primer prirejanja vrednosti spremenljivki v jeziku C#.

```
result = value / 100 + sum;
```

Ta stavek sestavljajo žetoni in leksemi, kot so v tabeli 1. Poudarimo, da ne poznamo pravih žetonov, ki jih uporablja C#. Za ta primer smo jih poimenovali tako, kot so običajno poimenovani.

Tabela 1: Žetoni in leksemi prireditvenega stavka

Žeton	Leksem
IDENTIFIER	result
ASSIGN_OPERATOR	=
IDENTIFIER	value
DIVISION_OPERATOR	/
LITERAL	100
ADDITION_OPERATOR	+
IDENTIFIER	sum
SEMICOLON	;

Nekateri žetoni so točno določeni in lahko sprejmejo samo ene lekseme. V zgornjem primeru so takšni žetoni ASSIGN_OPERATOR, DIVISION_OPERATOR, ADDITION_OPERATOR in SEMICOLON. Noben drug leksem se ne more kategorizirati v enega izmed teh žetonov. Število 100 je žeton LITERAL, ki predstavlja konstantno vrednost. Katerakoli konstantna vrednost se tako preslika v žeton LITERAL. Spremenljivke – v našem primeru result, value in sum – pa so vse istega tipa žetona, tj. IDENTIFIER. Ta tip žetona prejme lekseme, ki zgolj kažejo na nekaj, kar vsebuje vrednost. V tem primeru se imena spremenljivk shranijo v tabelo simbolov.

2.5.2 Tabela simbolov

Tabela simbolov je neke vrste slovar (angl. dictionary), v katerem prevajalnik hrani imena vseh spremenljivk in njihove informacije ter določa njihovo vidnost (povzeto po: Gabbrielli in Martini, 2010).

Informaciji o spremenljivkah v tabeli sta na primer podatkovni tip spremenljivke in njena vrednost. Vidnost pa je mišljena kot doseg spremenljivke (angl. scope), ki določa, kje v programu jo lahko uporabimo. Doseg spremenljivk lahko v osnovi implementiramo na dva načina: funkcionalen in imperativni. Pri funkcionalni izvedbi nobena operacija ne more uničiti tabele oz. uničiti dela nje. Pri imperativni pa je ravno obratno. Ko pri slednji zapustimo neki doseg, je ta doseg uničen, spremenljivke v njem pa ne obstajajo več. Na praktični ravni je to običajno doseženo z uporabo skladov, kjer vsak element predstavlja zaključen doseg. V tabeli so prav tako vse potrebne informacije o kakršnemkoli konstruktu, ki ga je uporabnik zapisal (npr. uporabniške funkcije ipd.).

Tabelo simbolov praviloma uporabljamo skozi celoten proces, od sintaktične analize pa do prevajanja.

2.5.3 Sintaktični analizator

Sintaktični analizator od leksikalnega analizatorja prejme žetone, ki jih uporabi za izgradnjo sintaksnega drevesa (angl. parse tree). Ta predstavlja sintaktično strukturo programa (povzeto po: Sebesta, 2016).

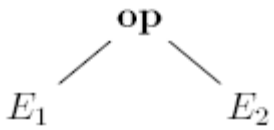
Poznan je tudi kot razčlenjevalnik ali še bolj pod angleškim izrazom parser. Sintaktična analiza izvorne kode preveri pravilnost sintaktične skladnje. Ponovno uporabimo podoben primer prirejanja vrednosti spremenljivki, ki smo ga uporabili pri opisu leksikalnega analizatorja, vendar ga tokrat malce spremenimo.

```
100 = result value / 55 sum +;
```

Leksikalni analizator ne ve, da je v tako napisanem izrazu karkoli narobe, sintaktični analizator pa takšne napake prepozna. Vemo namreč, da konstanti (100) ne moremo prirejati vrednosti. V praksi bi se sintaktična analiza ustavila že tukaj in javila napako, toda vseeno nadaljujmo. Žetonu IDENTIFIER (result) ne more neposredno slediti nov IDENTIFIER (value). Med njima bi namreč moral biti v najpreprostejšem primeru vsaj eden izmed aritmetičnih operatorjev. Prav tako žetonu LITERAL (55) ne more slediti IDENTIFIER (sum). Nazadnje pa še na desni strani operatorja za seštevanje (+) stoji žeton SEMICOLON (;), ki ga ne moremo uporabiti v računskih izrazih, temveč označuje zaključek vrstice.

Sintaktični analizator torej pozna pravilno skladnjo programskega jezika. Dokler je ta pravilna, pa iz žetonov ustvarja sintaksno drevo, ki je sestavljeno iz korena, vozlišč in listov. Listi so končni ali terminalni simboli, vozlišča in koreni pa nekončni ali neterminalni simboli. Več o njih bo predstavljeno v poglavju Specifikacija jezika JanC.

Obstajata dve vrsti dreves, konkretno in abstraktno sintaksno drevo (angl. Abstract Syntax Tree – AST). Konkretno vsebuje vse žetone in je v bistvu preslikava nekega stavka v obliko drevesa, ki predstavlja logično strukturo progama. Vsebuje vse posebne znake, kot so npr. oklepaji in podpičja. Abstraktno pa je zgrajeno samo iz žetonov, ki programu dajejo dejanski pomen. Posebne znake zavrže, saj za nadaljnjo analizo niso več potrebni. V izvorni kodi so samo zato, da s sintaktičnim analizatorjem lažje zgradimo pravilno obliko drevesa in da je sintaksa človeku prijaznejša. V sintaktičnih analizatorjih se običajno gradijo samo abstraktna sintaksna drevesa. V najbolj preprosti obliki je takšno kot na sliki 2.



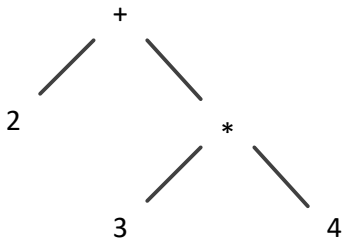
Slika 2: Osnovna oblika sintaksnega drevesa

Vir: Aho, et al., 2007, 93

Drevo predstavlja izraz s korensko operacijo, ki ji priredimo levi in desni del. To lahko predstavlja računsko operacijo med dvema členoma, prav tako pa se lahko nanaša na primer na zanko while, kjer je while operacija v korenu, njen pogoj je levi del, blok s stavki pa desni del. Levi in desni del sta lahko vozlišči novega drevesa, ki vsebuje svoja poddrevesa. Tako lahko gradimo drevo v globino, vse dokler ne pridemo do listov.

Ko sintaktični analizator naleti na neki žeton, se glede na produkcijska pravila odloči, kako v nadaljevanju graditi drevo. Več o produkcijskih pravilih je opisano v poglavju Specifikacija jezika JanC, na tem mestu pa omenimo zgolj, da določajo skladnjo jezika. Nekatera produkcijska pravila so preprosta, saj že po prvem žetonu natančno vemo, kateri žetoni morajo slediti. Bolj zahteven proces gradnje drevesa so računski izrazi, ki so lahko poljubno dolgi. Poleg tega pa mora hkrati upoštevati še pravila vrstnega reda operacij in pomen oklepajev. Poglejmo si preprost primer sintaksnega drevesa naslednjega izraza.

```
2 + 3 * 4
```



Slika 3: Sintaksno drevo računskega izraza

Slika 3 prikazuje sintaksno drevo danega izraza in je glede na izvedbo sintaktičnega analizatorja nedvoumna. Ljudem takšen izraz ne predstavlja težave, saj vemo, da ima

množenje prednost pred seštevanjem. Računalnik tega ne ve, zato mu takšen izraz podamo v obliki drevesa, ki ga izračuna od spodaj navzgor. Tako lahko vidimo, da najprej izračuna spodnji del, kjer je množenje, in šele nato produktu prišteje 2.

Sintaktični analizator je implementiran na enega izmed dveh načinov: top-down ali bottom-up. Top-down gradi drevo od zgoraj navzdol, torej od korena do listov, bottom-up pa od spodaj navzgor. V obeh primerih je njegov vhod v obliki žetonov prebran od leve proti desni (povzeto po: Aho, et al., 2007).

Top-down se odloča, kako graditi drevo glede na žeton, ki ga dobi, in glede na naslednji ali več naslednjih žetonov (odvisno od zahtevnosti produkcijskih pravil). Produkcijsko pravilo bo upošteval takoj, ko ga lahko določi, in nadaljnji žetoni bodo morali zadostovati izbranemu pravilu. Bottom-up sintaktični analizator pa šele po analizi vseh žetonov določi, kateremu produkcijskemu pravilu je zadoščeno. Če ni nobenemu, javi napako. Naprednejši bottom-up sintaktični analizatorji lahko hkrati gradijo več dreves, ki jih na koncu združijo (shift-reduce parser).

Top-down sintaktični analizator je lažji za implementacijo, saj hitreje prepozna produkcijsko pravilo, ki ga mora uporabiti. Morajo pa biti pravila zaradi tega preprostejša oz. nedvoumna (povzeto po: Neha, 2023).

Rekurzivno spuščajoči (angl. recursive-descent) sintaktični analizator je top-down analizator, sestavljen iz rekurzivnih podprogramov, ki gradijo drevo od zgoraj navzdol. Rekurzivni pristop odseva naravo programskih jezikov, kjer lahko na primer izrazi nastopajo kot deli drugih izrazov (povzeto po: Sebesta, 2016).

To je najlažje razvidno pri matematičnih izrazih. Zamislimo si zelo preprost programski jezik, ki omogoča seštevanje poljubnih vrednosti, hkrati pa zahteva, da seštejemo vsaj dve vrednosti. Stavek v takšnem jeziku lahko formuliramo kot:

```
vsota = izraz + izraz
```

kjer je izraz lahko število ali nov izraz. Torej kadarkoli sintaktični analizator naleti na izraz, rekurzivno izvede proceduro za razčlenjevanje izrazov. Primer:

```
6 = 1 + 2 + 3
```

To lahko zapišemo kot:

```
6 = izraz + 3
```

kjer je

```
izraz = 1 + 2
```

Rekurzija se mora ustaviti pri številki, ki ji ne sledi znak plus (+). To pomeni, da tega izraza ne moremo deliti na nadaljnje izraze. V našem primeru je številka končni simbol, izraz pa nekončni simbol.

Izdelava sintaktičnega analizatorja je običajno najzahtevnejši del celotnega procesa. Čeprav sta tukaj leksikalni in sintaktični analizator predstavljena kot ločena in zaporedna procesa, je leksikalni analizator v praksi običajno implementiran kot podproces sintaktičnega analizatorja. Leksikalni analizator torej ne ustvari celotnega niza žetonov vnaprej, temveč je nov oz. naslednji žeton ustvarjen šele, ko ga sintaktični analizator zahteva.

Če tekom sintaktične analize ni prišlo do napake, bo izhod iz sintaktičnega analizatorja abstraktno sintaksno drevo, ki ga uporabimo v naslednjih fazah.

2.5.4 Vmesni generator kode, semantična analiza in optimizacija

Vmesni generator kode izvorno kodo spremeni v vmesno kodo, ki je zelo podobna zbirnemu jeziku ali pa je že zbirni jezik. Na njej izvede tudi optimizacijo in jo s tem naredi

boljšo. Boljša koda običajno pomeni hitrejšo kodo, lahko pa tudi krajšo ali pa tako, ki potrebuje manj energije za izvajanje (povzeto po: Aho, et al., 2007).

V tem delu ima prav tako pomembno vlogo semantični analizator, ki išče napake, ki so v prejšnjih fazah težje razvidne. Njegova naloga je med drugim preverjanje ujemanja podatkovnih tipov (angl. type checking).

Type-checking preverja operande in ujemanje njihovih tipov. Ujemajoči tipi so tisti, ki jih z operatorji lahko združimo oz. so v pravilih specifičnega jezika dovoljeni. Takšna sta na primer int in double tipa v jeziku Java, kjer je int interno spremenjen v double in sta torej prepoznana kot ujemajoča se tipa (povzeto po: Sebesta, 2016). Sintaktično pravilen, toda semantično nepravilen je tudi naslednji primer programa v C#.

```
a = 15;
int a;
```

Sintaktični analizator v nobeni od dveh vrstic ne najde sintaktične napake. Skladnja stavkov je popolnoma pravilna, napačen pa je njun vrstni red. Spremenljivki "a" prirejamo vrednost, še preden je ta sploh deklarirana. V tem primeru najde napako semantični analizator. Če bi takšna napaka spolzela skozi prste, bi se med izvajanjem v prvem stavku sklicevali na spremenljivko, ki sploh ne obstaja, kar bi povzročilo sesutje programa.

2.5.5 Generator kode

Kot zadnja faza v prevajanju nastopi generator kode, ki prej pregledano in optimizirano vmesno oz. zbirno kodo prevede v strojno kodo. Ta koda se tako servira procesorju, ki jo izvede.

Ugotovili smo, da je proces prevajanja visokonivojskega jezika v strojno kodo razmeroma kompleksen in natančno razdelan proces, ki s stališča računalnika potrebuje precej časa in pomnilnika. Tudi s stališča človeka je takšen proces opazno zamuden, saj prevajanje že najbolj osnovnega programa (npr. konzolne aplikacije v C#) traja vsaj nekaj sekund.

Tolmačenje se od prevajanja razlikuje predvsem od faze vmesnega generatorja kode dalje. Tu se izvorna koda namreč ne prevede v nižje jezike, temveč jo tolmač kar na licu mesta izvede. Semantični analizator je praviloma edini od trojice, ki tu še igra vlogo.

2.5.6 Upravljanje s programskimi napakami

Neizbežno je, da bo v času razvoja programa v njem kakšna napaka. Nekatere so takšne, da jih prevajalnik brez težav odkrije in programa sploh ne prevede, druge pa so skrite globlje in jih je mogoče odkriti šele med izvajanjem, tekom prevajanja pa zelo težko. Takšna napaka je na primer možnost neomejene rekurzije.

Večina specifikacij programskih jezikov ne določa, kako naj se prevajalnik spopade z njimi. To je prepuščeno izvedbi prevajalnika.

Običajne napake so naslednje:

- leksikalne napake, kot so npr. uporaba nedovoljenih znakov v imenih spremenljivk;
- sintaktične napake, ki jih povzroči nepravilna kombinacija oz. zaporedje žetonov, kot na primer manjkajoče podpičje na koncu stavka;
- semantične napake, kot je npr. vračanje napačnega podatkovnega tipa v metodi oz. funkciji, in
- logične napake, kot je na primer uporaba spremenljivke, ki ni dosegljiva (povzeto po: Aho, et al., 2007).

Od sintaktičnega analizatorja pričakujemo, da javi kakršnokoli sintaktično napako na razumljiv način (povzeto po: Aho, et al., 2007). Misel lahko razširimo na celoten proces prevajanja. Vsaka faza procesa skrbi za svojo domeno in od nje želimo, da nam ob pojavu napak o njih tudi poroča. Najmanj, kar moramo vedeti, je vrstica v izvorni kodi, kjer se je napaka pojavila. Vsaka dodatna informacija pa je za uporabnika programskega jezika več kot dobrodošla, saj mu poenostavi odpravljanje napak.

3 RAZISKAVA NEANGLEŠKIH PROGRAMSKIH JEZIKOV

Razvoj računalništva se je v veliki meri odvijal v angleško govorečih delih sveta. Prav tako so iz tam izhajali pomembnejši preboji v razvoju programskih jezikov.

Kot prvi pravi visokonivojski jezik lahko štejemo Fortran, ki ga je skupaj z ekipo izdelal John Backus leta 1957 (povzeto po: Gabbrielli in Martini, 2010). John Backus je bil ameriški računalniški znanstvenik.

Grace Brewster Hopper – prav tako ameriška računalniška znanstvenica, ustvarjalka programskega jezika FLOW-MATIC in članica ekipe, ki je razvijala računalnik UNIVAC 1 – je leta 1953 predlagala, da naj se nekateri programi pišejo v angleščini. Najprej je naletela na neodobravanje, češ da računalniki ne razumejo angleščine. Kasneje jim je uspelo narediti prevajalnik, ki je angleščino razumel (povzeto po: Wexelblat, 1981).

To je eden izmed glavnih razlogov, da je večina programskih jezikov angleških. Kljub temu pa obstaja tudi obilica lokalnih programskih jezikov ali takšnih, ki podpirajo več jezikov. Nekateri izmed njih so:

- Phoenix. Visokonivojski objektni jezik, ki ima arabsko sintakso. Piše se iz desne proti levi (https://arxiv.org/ftp/arxiv/papers/1907/1907.05871.pdf#:~:text=Phoenix%20is%20a%20General%2DPurpose,experience%20in%20the%20Arabic%20language).
- Karel. Programski jezik, ki je namenjen začetnikom, predvsem otrokom. Preveden je v več jezikov (https://www.cs.mtsu.edu/~untch/karel/).
- Linotte. Jezik s francosko sintakso in razvojnim okoljem v francoščini. Namenjen je predvsem lažjemu učenju programiranja (http://langagelinotte.free.fr/wordpress/).
- BAIK. Skriptni jezik v indonezijščini (https://media.neliti.com/media/publications/101763-EN-baika-programming-language-based-on-indo.pdf).
- Citrine. Skriptni jezik, ki omogoča sintakso v 114 naravnih jezikih (https://citrine-lang.org/).

– Catrobat. Vizualni programski jezik, ki podpira obilico naravnih jezikov, med njimi tudi slovenščino (https://catrobat.org/).

Jezik Citrine (Citrine, 2021) se zgleduje po jeziku Smalltalk, ki je uvedel izraz objektno programiranje. Razvil ga je Gabor de Mooji leta 2014. Stremi k čim bolj preprosti sintaksi, ki mestoma uporablja tudi ikone.

Na sliki 4 vidimo primer prirejanja vrednosti spremenljivki "x", kjer stavek začnemo z ikono roke.

```
👉 x ≔ 'Hello' length.
👉 x ≔ 3 + 7.
👉 x ≔ Number between: 0 and: 10.
```

Slika 4: Programski jezik Citrine
Vir: https://citrine-lang.org/ (28. 2. 2023)

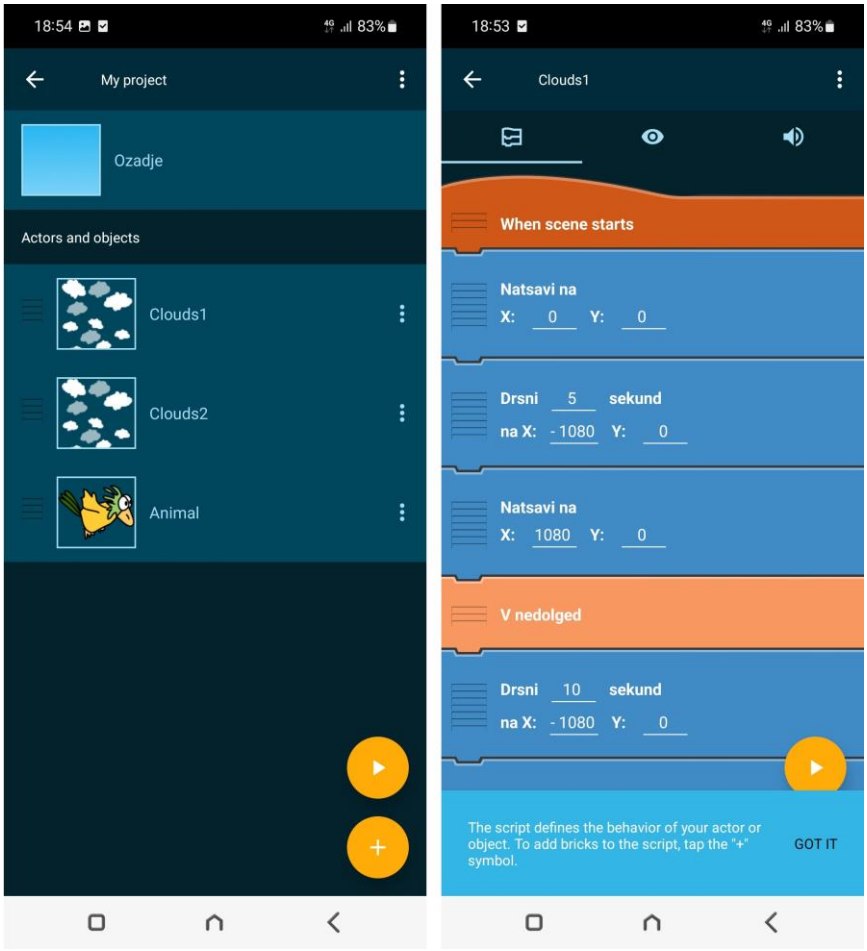
Jezik v času pisanja diplomskega dela podpira vsaj 114 naravnih jezikov, med njimi tudi slovenščino. Primer z uradne spletne strani jezika je na sliki 5.

Slika 5: Citrine v slovenščini
Vir: https://citrine-lang.org/ (28. 2. 2023)

Slovenščino podpira tudi jezik Catrobat. Catrobat je neprofitni projekt, s centrom v Avstriji, čigar cilj je ozaveščati o pomembnosti znanja programiranja in informacijsko

komunikacijskih tehnologij na splošno. Catrobat poskuša doseči mladostnike in jih na zabaven način vpeljati v svet programiranja. Prepričani so namreč, da bi moralo biti več mladostnikov soustvarjalcev digitalne sedanjosti in prihodnosti, ne pa da so samo v vlogi potrošnikov (https://catrobat.org/).

Preizkusili smo tudi njihovo mobilno aplikacijo Pocket Code, ki je dosegljiva v trgovini Google Play. Zanimalo nas je, koliko in kje se slovenščina v aplikaciji pojavlja. Ugotovili smo, da je vmesnik aplikacije malce zmeden, saj ga je nekaj v angleščini, nekaj pa v slovenščini. Nekatere slovenske besede so bile tudi napačno napisane. V aplikaciji na vizualen način programiramo igro, katere osnova pa je že vnaprej pripravljena. Na sliki 6 lahko vidimo prvo stran že pripravljenega projekta in stran, kjer z vizualnimi elementi programiramo igro.



Slika 6: Aplikacija Pocket Code

Vir: https://catrobat.org/ (28. 2. 2023)

Tekom raziskave smo na spletu našli omembo zgolj enega programskega jezika, ki uporablja slovensko sintakso in je bil izdelan v Sloveniji. Podobno kot tukaj je bil tudi ta razvit v sklopu diplomskega dela z naslovom Izdelava lastnega programskega jezika s slovensko sintakso (Kralj, 2020). Nismo pa zasledili, da bi se ta jezik kje uporabljal.

Ugotovili smo, da so lokalni programski jeziki mnogokrat uporabljeni predvsem za učenje mladostnikov. Pozitivni rezultati, ki naj bi iz tega izhajali, so bili zajeti tudi v raziskavah.

Sayamindu Dasgupta in Benjamin Mako Hill sta izvedla raziskavo v petih državah na vzorcu 15.000 uporabnikov vizualnega programskega jezika Scratch. Jezik je namenjen učenju otrok in je preveden v mnogo jezikov. Zaključila sta, da dojemanje konceptov programiranja poteka hitreje, ko se mladostniki učijo v programskem jeziku, ki podpira njihov materni jezik. Hkrati pa domnevata, da je zaradi tega tudi večja možnost, da se bodo mladostniki sploh začeli ukvarjati s programiranjem (povzeto po: Dasgupta in Hill, 2017).

4 SPECIFIKACIJA JEZIKA JANC

JanC je visokonivojski imperativen programski jezik, njegova sintaksa pa je zelo podobna jeziku C#, le da je v slovenščini. V sklopu diplomskega dela je bil implementiran v obliki tolmača, ki je napisan v C# in cilja .NET Standard verzije 2.0, kar pomeni, da ga lahko integriramo v katerikoli projekt, ki uporablja novejše različice tehnologije .NET (https://learn.microsoft.com/en-us/dotnet/standard/net-standard?tabs=net-standard-2-0). Jezik je skupaj s svojim tolmačem zanimiv predvsem kot vmesnik, ki uporabniku aplikacije ponuja možnost programiranja poslovne logike.

4.1 FUNKCIONALNOSTI JEZIKA

JanC ponuja osnovne funkcionalnosti, ki jih programer pričakuje od programskega jezika. V okviru njih pa jih lahko poljubno razširjamo z uporabo mehanizma registracije metod v C#. Je tipiziran (angl. strongly typed) jezik, ki razlikuje med malimi in velikimi črkami (angl. case-sensitive).

Funkcionalnosti jezika JanC so:

- uporaba podatkovnih tipov int, double in string;
- delo s spremenljivkami;
- omejen doseg spremenljivk;
- računanje algebraičnih izrazov;
- združevanje nizov;
- pogojni stavki (if in if/else);
- zanke (while);
- definicije funkcij;
- rekurzivni in medsebojni klici funkcij;
- integrirani funkciji za izpis in vnos;
- registracija poljubnih metod v C# s parametri int, double in string ter
- pisanje vrstičnih komentarjev v izvorni kodi.

Podatkovni tipi so najosnovnejši. Ne moremo deklarirati tabel, listov ali podobnih struktur. Prav tako ne moremo ustvarjati razredov in objektov.

Spremenljivke lahko deklariramo, inicializiramo ali pa definiramo. Deklaracija se mora zgoditi pred inicializacijo, podatkovni tip spremenljivke pa ne dovoli shranjevanja drugačnega tipa vrednosti. To naredi jezik tipiziran. Tu so sicer izjeme, ki so podobne kot v C#. V tip double namreč lahko shranimo int, ne pa obratno. Prav tako lahko v string shranimo neko spremenljivko z drugačnim podatkovnim tipom, saj se bo njena vrednost interno spremenila v string. Primeri deklaracije, inicializacije in definicije spremenljivke v JanC so naslednji:

```
celo a;                          Deklaracija
a = 1;                           Inicializacija
decimalno d = 2.3;               Definicija
```

Doseg spremenljivk je omejen. Spremenljivka je dosegljiva samo v bloku, v katerem je deklarirana, in v vseh gnezdenih blokih znotraj njega. Po zaključku bloka, kjer je deklarirana, lahko spremenljivko ponovno deklariramo. Prav tako so spremenljivke v funkcijah dosegljive samo znotraj nje. V spodnjem primeru pogojnega stavka v JanC je z rdečo označena neveljavna deklaracija, z zeleno pa veljavna deklaracija spremenljivke.

```
Program
{
	niz n;

	če(1 == 1)
	{
		niz n;
		celo a;
	}

	celo a;
}
```

Matematični izrazi lahko kot operande vsebujejo konstantne vrednosti (števila), spremenljivke, funkcije z ustreznim tipom vračanja in registrirane metode. Izvajamo lahko

operacije seštevanja, odštevanja, množenja in deljenja, kjer tolmač pozna vrstni red operacij, pravilno pa obravnava tudi oklepaje.
Nize lahko združujemo z znakom plus (+). Če želimo v niz shraniti vrednost neke spremenljivke, jo shranimo tako, da plusu sledi ime spremenljivke. Tolmač v tem primeru ve, da prirejamo vrednost tipu string, zato vrednost spremenljivke interno spremeni v string.

Tok programa lahko usmerjamo s pogojnimi stavki (if in if/else). V pogoju lahko izvedemo eno primerjavo. Primerjamo lahko samo iste podatkovne tipe (razen izjeme pri int in double), vseeno pa je, ali so v obliki konstante ali spremenljivke. Primerjati ne moremo vrednosti, ki jo vrača funkcija ali registrirana metoda. Uporabimo lahko vse operatorje za primerjanje (==, !=, <, >, <=, >=). Prav tako lahko z njimi primerjamo tudi stringe. Pogojne stavke lahko poljubno gnezdimo.

Jezik ponuja tudi najosnovnejšo zanko, tj. zanka while. Za njen pogoj velja isto kot za pogojni stavek. Tudi tu lahko poljubno gnezdimo. Primer zanke v JanC je:

```
Program
{
	celo i = 0;

	dokler(i < 10)
	{
		i = i + 1;
	}
}
```

Definiramo lahko tudi lastne funkcije, ki vračajo vrednost enega izmed treh podatkovnih tipov ali pa ne vračajo ničesar (angl. void). Ko vračajo, moramo uporabiti ključno besedo "vrni", ki lahko vrača konstanto ali spremenljivko oz. njeno vrednost. Vrnemo lahko samo tik pred zaključkom funkcije (pred njenim zaključnim zavitim oklepajem). Funkciji lahko določimo poljubno število vhodnih parametrov, ki so lahko kateregakoli izmed treh tipov. Število funkcij je neomejeno, definirati pa jih moramo nad glavnim programom. V C# bi to bilo videti tako, kot da bi metode definirali nad metodo Main. Imena funkcij se morajo

razlikovati od rezerviranih besed in imen registriranih metod. Primer definicije funkcije, ki vrača vrednost podatkovnega tipa niz:

```
funkcija niz PonavljajNiz(niz n, celo štPonovitev)
{
	niz zaVrniti = "";
	celo i = 0;

	dokler(i < štPonovitev)
	{
		zaVrniti = zaVrniti + n;
	}

	vrni zaVrniti;
}
```

Funkcije lahko kličemo rekurzivno ali pa znotraj funkcije kličemo drugo funkcijo. Pri tem ni pomembno, katera funkcija je v izvorni kodi definirana prej. Kot že omenjeno, pa morajo biti vse funkcije definirane pred programom.

V osnovi sta že integrirani dve metodi, in sicer za izpis in vnos v obliki Action in Func<string>. Lahko ju tudi prepišemo. Prednastavljeno je metoda za izpis prazna, metodo za vnos pa moramo pred uporabo definirati. V nasprotnem primeru nam njen klic vrne napako. Primer klica metod:

```
	Izpiši("Pozdravljen, svet!");

	niz vnos;
	Vnos(vnos);
```

Implementiran je tudi način za registracijo poljubnih metod v C#, ki jih tolmaču podamo v obliki delegatov. Registriranim metodam moramo določiti ime, ki ne sme biti prazno in mora biti različno od rezerviranih besed. Z registriranimi metodami razširimo funkcionalnosti jezika, saj lahko v telesu podane metode uporabljamo vse zmogljivosti C#. Edina omejitev je, da jih lahko definiramo samo s parametri int, double in string, prav tako pa lahko vračajo samo enega izmed teh tipov. V kodi JanC preprosto pokličemo

registrirano metodo preko njenega imena, ji podamo zahtevane parametre in metoda v C# se bo tekom tolmačenja izvršila.

V jeziku JanC so omogočeni tudi vrstični komentarji, ki jih začnemo s številskim znakom (#). Primer:

```
# To je vrstični komentar.
```

4.2 SINTAKSA JEZIKA

V nadaljevanju so predstavljene sintaktične značilnosti jezika JanC. Ogledali si bomo rezervirane besede in simbole ter produkcijska pravila, ki so opisana s sintakso EBNF (angl. Extended Backus-Naur form). Da jezik uporablja slovensko sintakso, pomeni, da so njegove ključne besede slovenske. Imena spremenljivk, funkcij in registriranih metod so lahko poljubna, tako kot v drugih programskih jezikih. V slovenskem jeziku so izpisane tudi napake, ki se pojavijo pri tolmačenju, toda to ni več del jezika, temveč tolmača.

4.2.1 Rezervirane besede in simboli

Kot v vsakem programskem jeziku so nekatere besede oziroma simboli rezervirani tudi v JanC, kar pomeni, da jih ne moremo uporabiti za poimenovanje spremenljivk ali funkcij. V tabeli 2 so vse rezervirane besede in simboli, ki se uporabljajo v jeziku JanC.

Tabela 2: Rezervirane besede in simboli

Beseda/simbol	**Razlaga besede/simbola**
Program	Označuje vstopno točko programa. V C# je to metoda Main.
funkcija	Deklaracijo funkcije začnemo z besedo "funkcija".
vrni	Označuje vračanje vrednosti v funkciji.
prazno	Uporabimo, ko funkcija ne vrača ničesar. Uporabimo jo lahko samo v definiciji funkcije. Void v C#.
niz	Podatkovni tip. Vrednost je karkoli med dvojnimi narekovaji ("). V nizu so lahko tudi rezervirane besede in simboli. String v C#.
celo	Podatkovni tip, ki zajame celo število. Int v C#.
decimalno	Podatkovni tip, ki zajame decimalno število. Double v C#.
Izpiši	Generična metoda za izpis.

Vnos	Generična metoda za vnos.
če	Označuje vejitev. Potrebuje pogoj. Če je pogoj izpolnjen, se izvede koda v "če" bloku. If stavek.
sicer	Označuje vejitev, ki se zgodi, če pogoj v "če" ni izpolnjen. Else stavek.
dokler	Zanka, katere koda v bloku se ponavlja, dokler je njenemu pogoju zadoščeno. Zanka while.
;	Označuje konec stavka.
(	Uporabljamo za ovitje pogoja v "če" ali "dokler" stavkih in za združevanje parametrov v deklaracijah ali argumentov v klicih funkcij oz. registriranih metod. Uporabljamo tudi v matematičnih izrazih.
)	Zaključuje oklepaj.
{	Označuje začetek bloka. Program, funkcije, "če", "sicer" in "dokler" se morajo vsi nadaljevati z blokom. Ta je sicer lahko prazen.
}	Zaključuje začetni zaviti oklepaj.
,	Vejica ločuje parametre v deklaracijah funkcij in argumente v klicih funkcij in registriranih metod. V JanC je za decimalno ločilo uporabljena pika (.), in ne vejica.
=	Enačaj označuje prirejanje vrednosti.
+	Znak za seštevanje ali združevanje nizov.
-	Znak za odštevanje.
*	Znak za množenje. Ima prednost pred seštevanjem in odštevanjem.
/	Znak za deljenje. Ima prednost pred seštevanjem in množenjem.
==	Dvojni enačaj se uporablja za primerjanje enakosti.
!=	Primerjanje neenakosti.
<	Manjši kot.
>	Večji kot.
<=	Manjši ali enak kot.
>=	Večji ali enak kot.
#	Številski znak, ki označuje začetek vrstičnega komentarja.

V C# lahko za primerjanje nizov uporabimo samo dva primerjalna operatorja (== in !=), za preostalo uporabljamo metodo String.CompareTo. V JanC lahko nize primerjamo z vsemi primerjalnimi operatorji.

4.2.2 Produkcijska pravila

Produkcijska pravila določajo veljavno kombinacijo besed in simbolov. S programskim jezikom pišemo stavke, ki morajo biti v skladu s produkcijskimi pravili, v nasprotnem primeru se program ne more izvesti.

V nadaljevanju so produkcijska pravila zapisana s sintakso EBNF. EBNF je metajezik, ki omogoča opis slovnice nekega jezika. Kot zanimivost povejmo, da lahko EBNF opiše tudi samega sebe, vendar to ni tako nenavadno, kot se sliši, kajti tudi slovenski jezik je opisan s slovenskimi besedami in slovničnimi pravili. Enako velja za vse naravne jezike.

Metajezik EBNF je standardiziran s strani Mednarodne organizacije za standardizacijo (angl. International Organization for Standardization – ISO) in Mednarodne komisije za elektrotehniko (angl. International Electrotechnical Commision – IEC) v standardu ISO/IEC 14977:1996. Standard smo uporabili za zapis EBNF našega programskega jezika.

Jezik EBNF sestavljajo:

- nekončni (angl. nonterminal) simboli,
- končni (angl. terminal) simboli,
- začetni simbol in
- produkcijska pravila.

Nekončni simboli so tisti, ki jih lahko delimo dalje na nekončne ali končne simbole. V našem zapisu so nekončni simboli obarvani zeleno in vsak izmed njih je definiran.

Končnih simbolov ne moremo deliti. Obarvani so rdeče in definicije kateregakoli nekončnega simbola se prej ali slej končajo pri definiciji, ki vsebuje samo še končne simbole. Izjema je samo pri definiciji znaka (angl. character), ki je lahko katerikoli vidni oz. nevidni znak.

Začetni simbol je nekončni simbol, ki je definiran, toda ni uporaben v nobeni drugi definiciji ali produkcijskem pravilu. V našem primeru je to "program".

Črno obarvani znaki so EBNF simboli z določenimi pomeni. Kot taki niso del opisanega jezika, temveč se uporabljajo samo znotraj sintakse EBNF. V tabeli 3 so vsi uporabljeni EBNF simboli in pripadajoči pomeni, zatem pa produkcijska pravila.

Tabela 3: Legenda EBNF simbolov

Simbol	Razlaga simbola
=	Definicija.
,	Simbol za združevanje. Ločuje elemente jezika.
;	Konec definicije.
|	Ali.
" … "	Označuje terminalni simbol, zapisan natanko tako, kot je med dvojnima narekovajema. Terminalni simbol ne more vsebovati dvojnega narekovaja.
' … '	Označuje terminalni simbol, zapisan natanko tako, kot je med enojnima narekovajema. Terminalni simbol ne more vsebovati enojnega narekovaja.
(…)	Združevanje.
[…]	Opcijsko. Neobvezen ali največ enkrat.
{ … }	Ponavljanje. Neobvezno ali poljubno število ponavljanj.
(* … *)	Komentar. Ni del opisanega jezika.
? … ?	Posebna vrsta, ki je izven dosega EBNF sintakse.
*	Znak za ponavljanje. Razumemo ga kot znak za množenje.
-	Znak za razen. Pomeni vse na njegovi levi strani, razen tistega, kar je na njegovi desni.

```
program
= { function } , "Program" , block ;

function
= "funkcija" , ("celo" | "decimalno" | "niz") , identifier ,
  "(" , ( | (parameter , { "," , parameter }) ) , ")" , function-block
| "funkcija" , "prazno" , identifier ,
  "(" , ( | (parameter , { "," , parameter }) ) , ")" , block ;

block
= "{" , { statement } , "}" ;

function-block
= "{" , { statement } , "vrni" , (literal | identifier) , "}" ;

parameter
= ("celo" | "decimalno" | "niz") , identifier ;
```

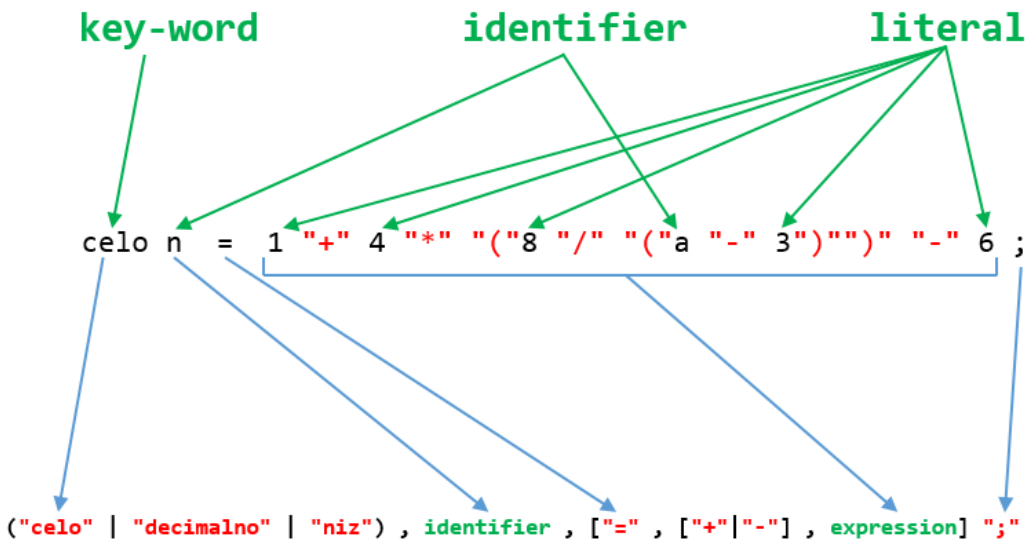
```
statement
= ("celo" | "decimalno" | "niz") , identifier , ["=" , ["+"|"-"] , expression] ";"
| identifier , "=" , ["+"|"-"] , expression , ";"
| "Izpiši" , "(" , (literal | identifier) , { "+" , (literal | identifier) } , ")" ,
  ";"
| "Vnos" , "(" , identifier , ")" , ";"
| "če" , "(" , condition , ")" , block , ["sicer" , block]
| "dokler" , "(" , condition , ")" , block ;

(*
  V primeru prirejanja vrednosti nizu, lahko za združevanje nizov uporabljamo samo
  prvo pravilo in znak plus (+). To je semantična omejitev, zato je v sintaktičnih
  pravilih nismo eksplicitno definirali.
*) expression
= (literal | identifier | function-call) , [("+"|"-"|"*"|"/") , expression]
| "(" , expression , ")" , [("+"|"-"|"*"|"/") , expression] ;

condition
= (literal | identifier) , ("=="|"!="|"<"|"<="|">"|">=") , (literal | identifier) ;

literal
= number | string ;

(*
  Identifier se lahko začne samo s črko ali podčrtajem (_). Sledi lahko največ 199
  dovoljenih znakov. Identifier ne sme biti isti kot ključna beseda (key-word).
*) identifier
= ((letter | "_") , 199 * { (digit | letter | "_") }) - key-word ;

(*
  Klic funkcije ali registrirane metode je sintaktično isti,
  zato se to pravilo uporablja za oba primera.
*) function-call
= identifier , "(" , ( | ((literal | identifier) ,
  { "," , (literal | identifier) }) ) , ")" ;

number
= "0" | natural-number | decimal-number ;

natural-number
= digit-excluding-zero , { digit } ;
```

```
(*
  Čeprav je sintaksa slovenska, zaradi lažje izvedbe za decimalno ločilo
  uporabljamo piko (.), in ne vejice (,).
*) decimal-number
= natural-number , "." , digit , { digit }
| "0" , "." , { digit } , natural-number ;

digit
= "0" | digit-excluding-zero ;

digit-excluding-zero
= "1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9" ;

(*
  String (niz) je lahko prazen ali pa vsebuje poljubno število poljubnih znakov, razen
  dvojnega narekovaja ("), ki ga uporabljamo za zaključevanje niza.
*) string
= '"' , { character - '"' } , '"' ;

letter
= "A"|"B"|"C"|"Č"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"|"N"|"O"|"P"|"R"|"S"|"Š"|"T"
 |"U"|"V"|"Z"|"Ž"|"Q"|"W"|"X"|"Y"|"a"|"b"|"c"|"č"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"
 |"m"|"n"|"o"|"p"|"r"|"s"|"š"|"t"|"u"|"v"|"z"|"ž"|"q"|"w"|"x"|"y" ;

key-word
= "Program"|"funkcija"|"vrni"|"prazno"|"niz"|"celo"|"decimalno"|"Izpiši"|"Vnos"|"če"
 |"sicer"|"dokler" ;

character
= ? any visible and invisible character ? ;
```

Oglejmo si uporabo produkcijskih pravil na primeru naslednjega stavka v jeziku JanC:

```
celo n = 1 + 4 * (8 / (a - 3)) - 6;
```

S tem stavkom spremenljivki "n", ki je tipa celo (angl. int), priredimo vrednost na desni strani. Na sliki 7 smo stavek razčlenili na produkcijska pravila. Stavek uporabi prvo pravilo definicije za statement, ki je napisano v spodnjem delu slike.



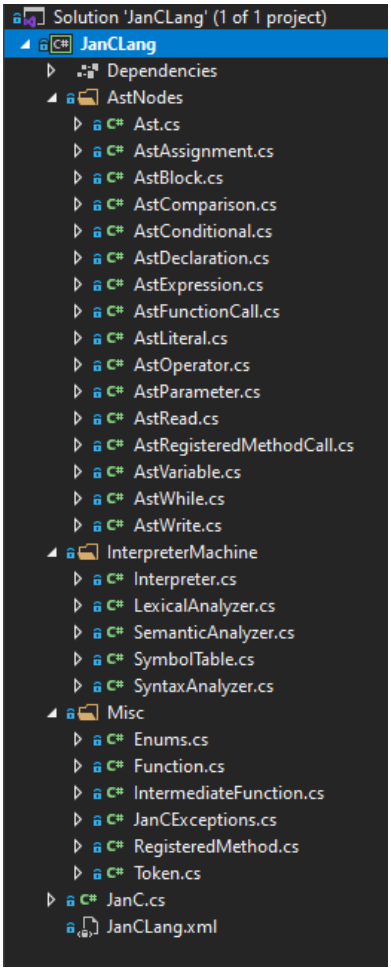
Slika 7: Členitev prireditvene izjave na produkcijska pravila

V nadaljevanju je na sliki 8 shematski prikaz prirejanja vrednosti z uporabo produkcijskih pravil. Koraki, kjer upoštevamo enega izmed pravil, so označeni s številko, za njo pa je pravilo v celoti napisano. Z rdečo so zapisani končni simboli, ki se pojavijo v zgornjem prireditvenem stavku. Z modro črto so podčrtani končni in nekončni simboli, ki se ujemajo z delom našega stavka. Zelene puščice kažejo tok branja stavka (od leve proti desni), z rdečo puščico pa se vračamo v prejšnja pravila oz. korake. Vidimo lahko, da se takšen prireditveni stavek tekom sintaktične analize preverja rekurzivno.

Slika 8: Prikaz prirejanja vrednosti z uporabo produkcijskih pravil

5 DELOVANJE TOLMAČA JANC

Tolmač jezika JanC je razvit s tehnologijo C#. Projekt je v obliki knjižnice, ki cilja .NET Standard 2.0 in se imenuje JanCLang. Naše tolmačenje deluje na podoben način kot prevajanje, ki smo ga spoznali v teoretičnem delu. Razlika je v fazi, kjer prevajanje ustvari vmesno kodo. V našem primeru v tej fazi uporabimo samo semantično analizo, nato pa tolmač izvede program JanC. Pomembnejša razlika je tudi v fazi leksikalne analize, kjer je leksikalni analizator običajno del sintaktičnega analizatorja. V praksi to pomeni, da sintaktični analizator za vsak naslednji žeton pokliče leksikalni analizator, ki mu ta žeton dostavi. Pri nas je leksikalni analizator popolnoma ločen del, ki kot prvi v celotnem procesu ustvari vrsto (angl. queue) vseh žetonov in jih posreduje semantičnemu analizatorju. Na sliki 9 vidimo sestavo projekta JanCLang.



Slika 9: Sestava projekta JanCLang

V mapi AstNodes so razredi, ki jih sintaktični analizator uporabi za sestavo abstraktnega sintaksnega drevesa. Kjerkoli v nadaljevanju uporabljamo besedno zvezo sintaksno drevo, s tem mislimo abstraktno sintaksno drevo. Mapa InterpreterMachine vsebuje razrede, ki predstavljajo "napravo" za tolmačenje oz. faze tolmačenja. Tabela simbolov (angl. symbol table) je edini del, ki ni faza sam po sebi, temveč neke vrste skladišče za vse informacije, ki so potrebne za tolmačenje. V mapi Misc so ostali "nerazvrščeni" razredi, ki jih pa prav tako potrebujemo za tolmačenje. V njej so na primer enumi in vse napake, ki jih sistem lahko javi tekom tolmačenja. Razred JanC je edini javno (angl. public) dostopen razred, preko katerega registriramo metode v C# in zaženemo kodo JanC. Celoten projekt uporablja angleščino za poimenovanje razredov, spremenljivk ipd., saj ga lahko uporablja kdorkoli na svetu. Izjemi sta samo opis napak in podatkovnih tipov enum. Na sliki 10 je javno dostopen razred JanC.

```
namespace JanCLang
{
    /// <summary>
    /// Provides methods for running JanC source code, defining builtin methods for write (Izpisi) and read (Vnos) and registering C# delegates into the JanC environment.
    /// </summary>
    0 references
    public class JanC
    {
        private SymbolTable SyT = new SymbolTable();
        private HashSet<Type> _validDataTypes = new HashSet<Type> { typeof(int), typeof(double), typeof(string) };
        private Queue<string> _lexemes = new Queue<string>();

        /// <summary>
        /// Method for registering JanC 'Izpisi' output action. Default is empty action.
        /// </summary>
        0 references
        public void RegisterWrite(Action<string> action)...

        /// <summary>
        /// Method for registering JanC 'Vnos' input function. If not registered, use of 'Vnos' keyword throws an exception.
        /// </summary>
        0 references
        public void RegisterRead(Func<string> function)...

        /// <summary>
        /// Method for registering an arbitrary action or function. Only int, double and string input/output data types are valid.
        /// </summary>
        0 references
        public void RegisterMethod(string methodName, Delegate method)...

        /// <summary>
        /// Interprets and runs JanC source code.
        /// </summary>
        0 references
        public void Run(string sourceCode)
        {
            try
            {
                var lexicalAnalyzer = new LexicalAnalyzer();
                Queue<Token> tokens = lexicalAnalyzer.RunLexicalAnalyzer(_lexemes, sourceCode, SyT);

                var syntaxAnalyzer = new SyntaxAnalyzer();
                AstBlock programAST = syntaxAnalyzer.RunParser(tokens, SyT);

                var semanticAnalyzer = new SemanticAnalyzer();
                semanticAnalyzer.RunSemanticAnalyser(programAST, SyT);

                var interpreter = new Interpreter();
                interpreter.RunInterpreter(programAST, SyT);
            }
            catch (JanCException)
            {
                throw;
            }
            catch (Exception e)
            {
                throw new UnknownException(e.Message);
            }
        }
    }
}
```

Slika 10: Razred JanC

Da slika ne zavzame preveč prostora, smo metode za registracijo izpisa (RegisterWrite), vnosa (RegisterRead) in poljubnih metod v C# (RegisterMethod) stisnili.

V metodi Run vidimo, kako poteka tolmačenje. Metodo pokličemo tako, da ji podamo izvorno kodo JanC v obliki niza (angl. string). Najprej inicializiramo leksikalni analizator, ki vrne vrsto žetonov. Te žetone nato prevzame sintaktični analizator, ki vrača sintaksno drevo. Zatem semantični analizator pregleda sintaksno drevo in če je vse v skladu z zahtevami, inicializiramo objekt interpreter, ki ga dejansko izvede.

Na vrhu razreda smo inicializirali tudi tabelo simbolov, v metodi Run pa lahko vidimo, da jo uporabljajo vse faze.

Kodo v metodi Run smo ovili v varnostni blok try/catch, kjer ulovimo vse lastne definirane napake in tudi morebitne nepredvidene.

Tolmačenje sestavljajo naslednji moduli:

- tabela simbolov,
- leksikalni analizator,
- sintaktični analizator,
- semantični analizator in
- tolmač.

5.1 TABELA SIMBOLOV

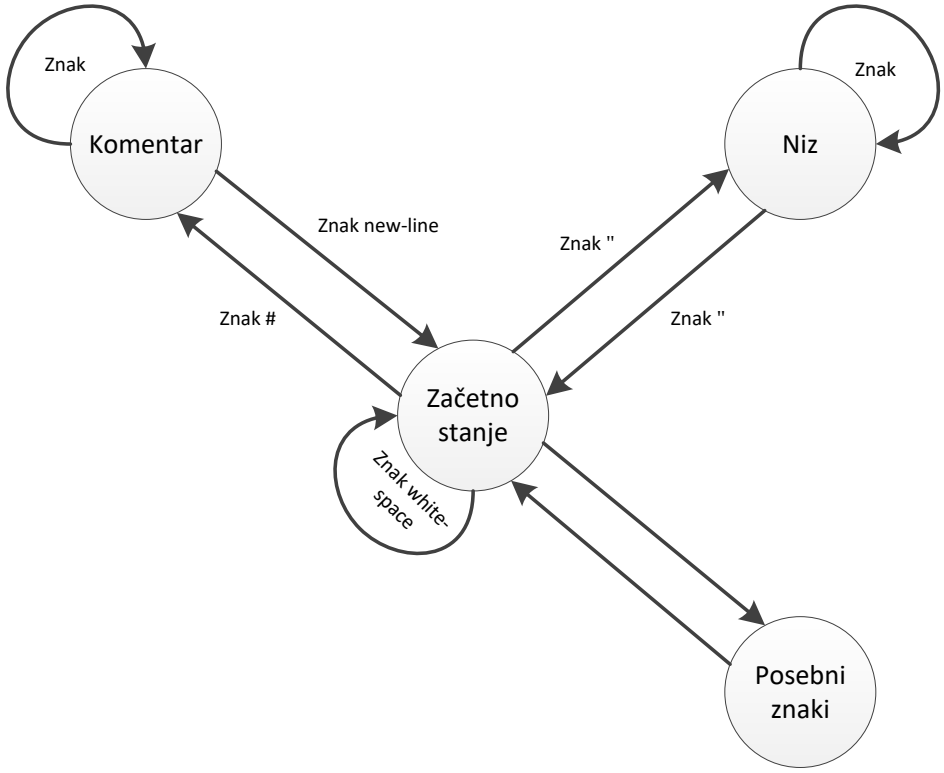
Tabela simbolov vsebuje definiciji metod za izpis in vnos, slovar (angl. dictionary) registriranih metod in funkcij ter sklad (angl. stack) slovarjev spremenljivk (angl. variable). Sklad predstavlja doseg spremenljivk. Poleg naštetega so v tabeli simbolov še vsi dovoljeni znaki ter rezervirane besede in simboli. Na sliki 11 je celotna koda tabele simbolov.

```
// Stores information that all the processes for interpreting share.
10 references
internal class SymbolTable
{
    internal Action<string> Write = _ => { };
    internal Func<string> Read;
    internal Dictionary<string, RegisteredMethod> RegisteredMethods = new Dictionary<string, RegisteredMethod>();
    internal Dictionary<string, Function> Functions = new Dictionary<string, Function>();

    // Stack of variables used to define their scope.
    // Program and each user function uses it's own element in the stack.
    // Start of the program creates the program scope and each user function call
    // pushes it's scope to the top of the stack. After function exits, the scope is removed.
    internal Stack<Dictionary<string, AstVariable>> Variables = new Stack<Dictionary<string, AstVariable>>();

    // Allowed characters for identifiers.
    internal HashSet<char> allowedChar = new HashSet<char> {
          '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
          'a', 'b', 'c', 'č', 'd', 'e', 'f', 'g', 'h', 'i',
          'j', 'k', 'l', 'm', 'n', 'o', 'p', 'r', 's', 'š',
          't', 'u', 'v', 'z', 'ž', 'q', 'w', 'x', 'y', '_'
        };

    // Reserved symbols are used for registering methods and lexical analysis.
    internal Dictionary<string, (TokenType type, int operatorPrecedence)>
        ReservedSymbols = new Dictionary<string, (TokenType, int)> {
            // Keywords
            { "Program", (TokenType.Program, -1) },
            { "funkcija", (TokenType.Function, -1) },
            { "vrni", (TokenType.Return, -1) },
            { "prazno", (TokenType.Void, -1) },
            { "niz", (TokenType.String, -1) },
            { "celo", (TokenType.Int, -1) },
            { "decimalno", (TokenType.Double, -1) },
            { "Izpiši", (TokenType.OutputMethod, -1) },
            { "Vnos", (TokenType.InputMethod, -1) },
            { "če", (TokenType.If, -1) },
            { "sicer", (TokenType.Else, -1) },
            { "dokler", (TokenType.While, -1) },

            // Separators
            { ";", (TokenType.EOL, -1) },
            { "(", (TokenType.L_Brack, -1) },
            { ")", (TokenType.R_Brack, -1) },
            { "{", (TokenType.L_Curly, -1) },
            { "}", (TokenType.R_Curly, -1) },
            { ",", (TokenType.Comma, -1) },

            // Operators and comparators
            { "=", (TokenType.Eql, -1) },
            { "+", (TokenType.Operator, 0) },
            { "-", (TokenType.Operator, 0) },
            { "*", (TokenType.Operator, 1) },
            { "/", (TokenType.Operator, 1) },
            { "==", (TokenType.Comparator, -1) },
            { "!=", (TokenType.Comparator, -1) },
            { "<", (TokenType.Comparator, -1) },
            { ">", (TokenType.Comparator, -1) },
            { "<=", (TokenType.Comparator, -1) },
            { ">=", (TokenType.Comparator, -1) },
    };
}
```

Slika 11: Izvedba tabele simbolov

5.2 LEKSIKALNI ANALIZATOR

Leksikalni analizator je končni avtomat (angl. finite-state machine), ki je naenkrat lahko samo v enem izmed končno mnogih stanj. Med njimi prehaja glede na vnaprej določena pravila. V našem primeru leksikalni analizator prejme niz z izvorno kodo JanC in ga znak za znakom analizira. Iz izvorne kode zna izločiti lekseme, iz katerih na koncu ustvari žetone. Slika 12 prikazuje leksikalni analizator kot končni avtomat z vsemi njegovimi stanji in prehodi med njimi.



Slika 12: Leksikalni analizator kot končni avtomat

Začnemo v začetnem stanju, kjer se leksem ustvari iz vseh znakov, ki so prišli pred katerimkoli nevidnim znakom (angl. white-space). Postopek se ponavlja, dokler ne preide v katerega izmed preostalih možnih stanj. Ko na primer leksikalni analizator prebere dvojni narekovaj ("), gre v stanje za obdelavo nizov. V tem stanju ostaja, dokler ponovno ne prebere dvojnega narekovaja. Takrat ustvari leksem, v katerem je celoten prebran niz, vključno z nevidnimi znaki, in se vrne v začetno stanje. Posebni znaki v našem primeru so +, -, *, /, ;, (,), {, } in ,. Za vsak posebni znak ustvari leksem. Pred tem pa ustvari še

leksem, ki ga je polnil v začetnem stanju, preden je naletel na posebni znak. Komentar se začne s številskim znakom (#) in se nadaljuje do konca vrstice. Iz komentarjev ne ustvarimo nobenih leksemov in jih že v tej fazi zavržemo.

Ko leksikalni analizator prebere vse znake izvorne kode, vrsto ustvarjenih leksemov preda procesu tokenizacije, ki je del njega samega. Tokenizacija iz leksemov ustvari žetone (angl. tokens). Prepozna ključne besede in posebne znake ter preveri pravilen zapis števil (ničelne številske vrednosti ne moremo zapisati kot 00 ali 0.0, prav tako naravnih števil ne moremo zapisati kot na primer 035). Hkrati preveri tudi dolžino leksema, če je ta prepoznan kot identifier. Iz produkcijskega pravila za identifier sledi, da je lahko največje število znakov 200. Tokenizacija nazadnje ustvari še poseben žeton, imenovan End-Of-File (žeton za konec datoteke), ki označuje konec izvorne kode. Poglejmo si nekaj stavkov v jeziku JanC.

```
# Priredimo vrednost spremenljivki n, ki je tipa niz
# in izpišimo njeno vrednost.
niz n = "To je neki niz.";
Izpiši(n);
```

Leksikalni analizator najprej prebere znak #, ki označuje začetek komentarja, zato vse znake do konca vrstice zavrže. Isto ponovi v drugi vrstici. V nadaljevanju pa ustvari lekseme in žetone, kot so v tabeli 4.

Tabela 4: Primeri leksemov in žetonov

Leksem	**Žeton**
`niz`	String
`n`	Identifier
`=`	Eql
`"To je neki niz."`	Literal
`;`	EOL
`Izpiši`	OutputMethod
`(`	L_Brack
`n`	Identifier
`)`	R_Brack
`;`	EOL

Med leksemoma "niz" in "n" ločimo zaradi presledka med njima. Če presledka ne bi bilo, bi ustvarili leksem "nizn", ki bi ga dodelili žetonu Identifier. Kasneje bi semantični analizator javil napako, da spremenljivka z imenom "nizn" ni deklarirana.

Žetoni shranjujejo tip žetona, vrednost žetona, ki je ista vrednosti leksema, in vrstico v izvorni kodi, kjer se leksem nahaja.

5.3 SINTAKTIČNI ANALIZATOR

Sintaktični analizator prejeme žetone, ki jih je ustvaril leksikalni analizator in preverja njihovo zaporedje ter s tem ujemanje s produkcijskimi pravili. Sproti gradi sintaksno drevo in v primeru napake ustavi izvajanje ter napako tudi javi. V slednji pove, da je naletel na nepričakovan simbol, kjer se izpišeta tako simbol kot vrstica v izvorni kodi.

Osnovni oz. korenski gradnik sintaksnega drevesa je razred AstBlock, ki drži listo (angl. list) stavkov. Kjerkoli ustvarimo nove bloke kode (pogojni stavki, zanke in funkcije) ponovno uporabimo razred AstBlock.

Na sliki 13 je del kode sintaktičnega analizatorja, ki predstavlja začetek sintaktične analize bloka kode. V vrstici 158 preverimo, če je tip žetona L_Curly (vrednost: {), s katerim se začne blok. Če je žeton katerikoli drug, javimo napako (vrstica 160). Zatem v vrstici 163 inicializiramo nov objekt razreda AstBlock in ga podamo metodi ParseBlock, ki je naslednji korak v analiziranju bloka.

```
156     5 references
        private AstBlock Block()
157     {
158         if (_token.Type != TokenType.L_Curly)
159         {
160             throw new UnexpectedTokenException(_token);
161         }
162
163         var block = new AstBlock();
164         ParseBlock(block);
165
166         return block;
167     }
168
```

Slika 13: Sintaktična analiza bloka kode

Slika 14 prikazuje metodo ParseBlock, ki preusmerja izvajanje sintaktičnega analizatorja v metode za vsako produkcijsko pravilo izjave (angl. statement) posebej. Izjema je žeton tipa Return, ki ga potrebujemo pri analiziranju funkcij. V metodi je zanka do/while, ki se izvaja, vse dokler ne prebere žetona R_Curly (vrednost: }). Ta označuje konec bloka (vrstica 261). Na koncu vsakega stavka, zajetega v case, pričakujemo žeton EOL (angl. End-Of-Line, vrednost ;), ki predstavlja zaključek vrstice.

Slika 14: Sintaktična analiza stavkov v bloku

5.3.1 Sintaktična analiza inicializacije spremenljivke

Natančneje si oglejmo, kako poteka inicializacija spremenljivke. Spremenljivko inicializiramo tako, da ji priredimo neko vrednost. V računskem izrazu moramo upoštevati vrstni red operacij in oklepaje. Za to smo uporabili algoritem Shunting-Yard (Wolf in Oser, 2023), ki običajno matematično notacijo, kjer je operator med operandoma (angl. infix), spremeni tako, da je operator za svojima operandoma (angl. postfix). Takšni notaciji v angleščini pravimo Reverse Polish Notation.

Za algoritem potrebujemo vrsto (angl. queue) in sklad (angl. stack). V vrsto shranjujemo operande, v sklad pa operatorje in oklepaje. Operatorje po določenih pravilih iz sklada prestavljamo v vrsto, oklepaje pa pri zaklepaju odstranjujemo in vse operatorje med njima prav tako prestavimo v vrsto. Začnemo s praznima vrsto in skladom ter celotnim izrazom. Predstavimo algoritem na preprostem primeru. Dan je naslednji izraz:

```
8 / 4 + 2 * (4 - 2)
```

V tabeli 5 so koraki algoritma, ki ga izvedemo nad izrazom. V označuje vrsto, S pa sklad. V nadaljevanju so koraki tudi opisani.

Tabela 5: Algoritem Shunting-Yard

Korak 1	**Korak 2**	**Korak 3**
8 / 4 + 2 * (4 - 2)	8 / 4 + 2 * (4 - 2)	8 / 4 + 2 * (4 - 2)
V 8 **S**	**V** 8 **S** /	**V** 8 4 **S** /
Korak 4	**Korak 5**	**Korak 6**
8 / 4 + 2 * (4 - 2)	8 / 4 + 2 * (4 - 2)	8 / 4 + 2 * (4 - 2)
V 8 4 / **S** +	**V** 8 4 / 2 **S** +	**V** 8 4 / 2 **S** + *
Korak 7	**Korak 8**	**Korak 9**
8 / 4 + 2 * (4 - 2)	8 / 4 + 2 * (4 - 2)	8 / 4 + 2 * (4 - 2)
V 8 4 / 2 **S** + * (	**V** 8 4 / 2 4 **S** + * (	**V** 8 4 / 2 4 **S** + * (-
Korak 10	**Korak 11**	**Korak 12**
8 / 4 + 2 * (4 - 2)	8 / 4 + 2 * (4 - 2)	8 / 4 + 2 * (4 - 2)
V 8 4 / 2 4 2 **S** + * (-	**V** 8 4 / 2 4 2 - **S** + *	**V** 8 4 / 2 4 2 - * + **S**

Korak 1: Vzamemo prvo število, tj. 8, in ga prestavimo v vrsto.

Korak 2: Operator za deljenje prestavimo v sklad.

Korak 3: Število 4 damo v vrsto.

Korak 4: Operator za seštevanje prestavimo v sklad, toda ker je pred njim operator s prednostjo (/), najprej tega prestavimo v vrsto in šele nato lahko + damo v sklad.

Korak 5: Število 2 damo v vrsto.

Korak 6: Prestavimo znak za množenje (*) v sklad. Pred tem + ne premaknemo v vrsto, saj ima * prednost.

Korak 7: Oklepaj prestavimo v sklad.

Korak 8: Število 4 prestavimo v vrsto.

Korak 9: Znak za odštevanje (-) prestavimo v sklad. Čeprav je pred njim operator *, ki ima prednost, je med njima oklepaj, zato * ne prestavimo v vrsto.

Korak 10: Prestavimo število 2 v vrsto.

Korak 11: Prestavimo še zaklepaj v sklad. Zaklepaj zapira zadnji oklepaj, zato v vrsto prestavimo vse operatorje med njima. V našem primeru je to samo operator za odštevanje (-). Zatem zavržemo tako oklepaj kot zaklepaj.

Korak 12: Ostaneta še operatorja + in *, ki ju iz desne proti levi prestavljamo v vrsto.

Z uporabo algoritma smo dobili izraz

```
8 4 / 2 4 2 - * +
```

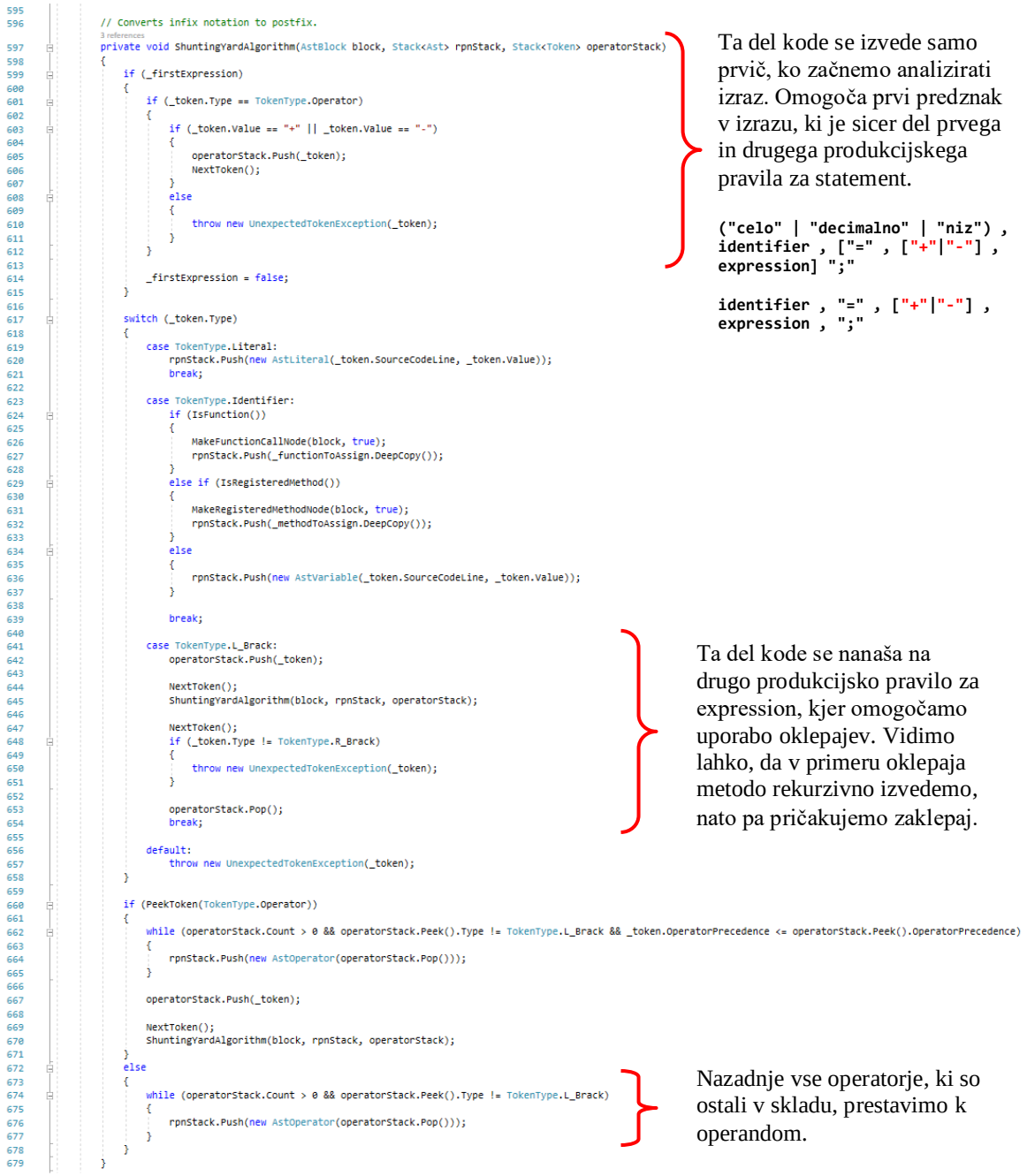
Tak izraz izračunamo tako, da ga beremo iz leve proti desni. Ko najdemo prvi operator, vzamemo zadnji dve števili pred njim in nad njima izvedemo operacijo, ki jo označuje operator. Ponovno beremo z začetka in postopek ponavljamo, dokler ni nobenega operatorja več.

Rezultat našega izraza z notacijo infix je 6. Preverimo še rezultat istega izraza z notacijo postfix, ki smo ga dobili z uporabo algoritma Shunting-Yard. Izračunamo ga po korakih ter z rdečo označimo števili in uporabljeno operacijo. Rezultat je v naslednjem koraku označen s sivo.

```
8 4 / 2 4 2 - * +
  2 2 4 2 - * +
    2 2 2 * +
      2 4 +
        6
```

Dobili smo enak rezultat.

Sintaktični analizator z uporabo algoritma preuredi izraze, ki jih nato sestavimo v sintaksno drevo. Na sliki 15 je celotna izvedba algoritma Shunting-Yard. Poudarimo, da smo namesto vrste uporabili sklad, ker ga kasneje v metodi, kjer iz notacije postfix gradimo sintaksno drevo, lažje uporabimo. Notacije postfix sicer ne bi bilo treba spreminjati v sintaksno drevo, saj bi lahko tolmač na zgoraj opisan način izračunal izraz z notacijo postfix. Za spremembo v sintaksno drevo smo se odločili z namenom zagotavljanja enotne zgradbe podatkovne strukture, ki jo prejme tolmač.



Slika 15: Izvedba algoritma Shunting-Yard

Sintaksno drevo izraza, ki smo ga z zgornjo izvedbo algoritma Shunting-Yard spremenili v izraz z notacijo postfix, zgradi metoda ParseRPN, prikazana na sliki 16. Metoda gradi drevo od zgoraj navzdol (angl. top-down parsing) in vedno najprej napolnimo desno vejo drevesa. Šele ko pridemo do desnega lista (končni simbol), začnemo polniti levo stran.

```
680
681         // Makes an AST out of rpn notation. It reads it from right to left.
            4 references
682         private void ParseRPN(Stack<Ast> rpn, AstExpression expression)
683         {
684             if (rpn.Count == 0 && _operandToCarry == null)
685             {
686                 return;
687             }
688
689             Ast operand;
690
691             if (_operandToCarry == null)
692             {
693                 operand = rpn.Pop();
694             }
695             else
696             {
697                 operand = _operandToCarry;
698                 _operandToCarry = null;
699             }
700
701             if (operand.AstType == AstType.Operator)
702             {
703                 if (expression.Operator == null)
704                 {
705                     expression.Operator = operand as AstOperator;
706                 }
707                 else if (expression.Right == null)
708                 {
709                     expression.Right = new AstExpression(_token.SourceCodeLine);
710                     (expression.Right as AstExpression).Operator = operand as AstOperator;
711
712                     ParseRPN(rpn, (AstExpression)expression.Right);
713                 }
714                 else if (expression.Left == null)
715                 {
716                     expression.Left = new AstExpression(_token.SourceCodeLine);
717                     (expression.Left as AstExpression).Operator = operand as AstOperator;
718
719                     ParseRPN(rpn, (AstExpression)expression.Left);
720                 }
721                 else
722                 {
723                     _operandToCarry = operand;
724                     return;
725                 }
726             }
727             else
728             {
729                 if (expression.Right == null)
730                 {
731                     expression.Right = operand;
732                 }
733                 else if (expression.Left == null)
734                 {
735                     expression.Left = operand;
736                 }
737                 else
738                 {
739                     _operandToCarry = operand;
740                     return;
741                 }
742             }
743
744             ParseRPN(rpn, expression);
745         }
746
```

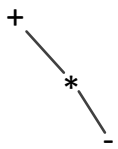
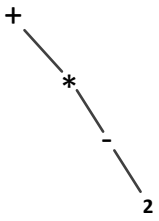
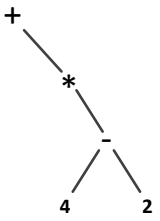
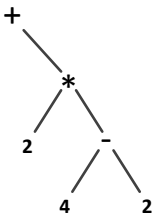
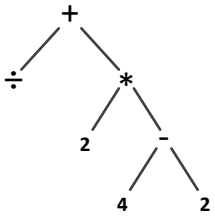
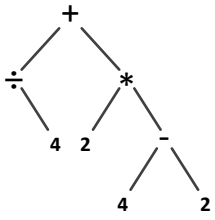
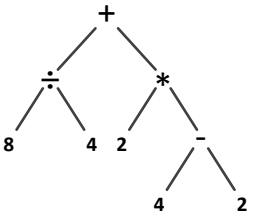
Slika 16: Metoda za gradnjo sintaksnega drevesa izraza

Metodo izvedemo rekurzivno (angl. recursive desent parser) vsakič, ko iz sklada "rpn" (parameter metode, vrstica 682) vzame operator. Takrat drevesu pripnemo nov izraz (vrstici 712 in 719). V preostalih primerih poskušamo operand postaviti najprej v desno vejo oz. levo, če je desna že zasedena. Če sta obe zasedeni, pa ga shranimo v spremenljivko "_operandToCarry" in ga postavimo na prvo prosto mesto višje v drevesu. Iz izraza

```
8 4 / 2 4 2 - * +
```

ki smo ga dobili v zgornjem primeru, tako postopoma zgradimo sintaksno drevo, kot je prikazano v tabeli 6. V vsakem koraku izrazu vzamemo člen na skrajni desni in ga pripnemo drevesu.

Tabela 6: Postopek gradnje sintaksnega drevesa izraza

8 4 / 2 4 2 - *	**8 4 / 2 4 2 -**	**8 4 / 2 4 2**
		
8 4 / 2 4	**8 4 / 2**	**8 4 /**
		
8 4	**8**	
		

5.3.2 Sintaktična analiza funkcij in klica funkcij

Deklaracija funkcije v JanC se začne s ključno besedo "funkcija", ki ji sledijo podatkovni tip vračanja, ime funkcije in njeni parametri. Zatem je v bloku zapisana njena koda. Vse deklaracije se morajo zvrstiti pred programom. Klic funkcije se izvede tako, da zapišemo njeno ime in v oklepajih podamo argumente.

Naša izvedba sintaktičnega analizatorja žetone analizira samo enkrat. To pomeni, da že v prvem prehodu čez izvorno kodo sproti shranjuje pomembne informacije o programu in gradi sintaksno drevo. Hkrati pa je to tudi omejitev, saj takšen sintaktični analizator ne more vedeti, kaj bo prišlo v nadaljevanju kode. Omejitev je v našem primeru najbolj opazna pri deklaracijah in klicih funkcij. Pri slednjih moramo namreč poznati ime funkcije in število parametrov, da sintaktični analizator prepozna produkcijsko pravilo. Ponovimo dve produkcijski pravili, ki sta lahko v tem primeru zavajajoči. Prvo pravilo je druga definicija za statement, drugo pravilo pa je definicija klica funkcije.

```
statement
= identifier , "=" , ["+"|"-"] , expression , ";"

function-call
= identifier , "(" , ( | ((literal | identifier) ,
  { "," , (literal | identifier) }) ) , ")" ;
```

Obe definiciji se začneta z nekončnim simbolom identifier. V tem primeru sintaktični analizator preveri, ali identifier (ime spremenljivke, funkcije ali registrirane metode) že obstaja. Če identifier kaže na spremenljivko, uporabimo pravilo za statement; če kaže na funkcijo, pa uporabimo pravilo za function-call. Funkcija mora biti v tem primeru shranjena oz. poznana pred njenim klicem. Ponazorimo na konkretnem primeru. Spodaj deklariramo dve preprosti funkciji v jeziku JanC. Rdeče je obarvan nemogoč klic funkcije, zeleno pa mogoč. Da bo primer kasneje bolj razumljiv, smo na dnu dodali tudi program, ki pa vsebuje prazen blok. Zaradi primera zanemarimo dejstvo, da bi klic katerekoli izmed teh funkcij vodil v neomejeno rekurzijo.

```
funkcija prazno Prva()
{
	Druga();
}

funkcija prazno Druga()
{
	Prva();
}

Program {}
```

Sintaktični analizator najprej analizira funkcijo Prva, saj je ta v izvorni kodi deklarirana prej. To naredi tako, da shrani njen podpis in blok. Šele nato analizira funkcijo Druga. Ker je med analiziranjem funkcije Druga podpis funkcije Prva že znan, lahko v Druga kličemo Prva. Obratno ni mogoče, ker med analiziranjem funkcije Prva Druga še ni poznana.

Opisano omejitev lahko odpravimo tako, da najprej analiziramo podpise funkcij in šele nato njihove bloke. Z našo izvedbo rešitve lahko hkrati ohranimo lastnost sintaktičnega analizatorja, da vse žetone analizira zgolj enkrat. Rešitev je predstavljena v nadaljevanju, najprej pa navedimo vse žetone, ki izhajajo iz zgornje izvorne kode. Levo je tip žetona, desna pa njegova vrednost.

Funkcija Prva		**Funkcija Druga**		**Program in konec**	
Function:	funkcija	Function:	funkcija	Program:	Program
Void:	prazno	Void:	prazno	L_Curly:	{
Identifier:	Prva	Identifier:	Druga	R_Curly:	}
L_Brack:	(	L_Brack:	(		
R_Brack:	)	R_Brack:	)	EOF:	end of file
L_Curly:	{	L_Curly:	{		
Identifier:	Druga	Identifier:	Prva		
L_Brack:	(	L_Brack:	(		
R_Brack:	)	R_Brack:	)		
EOL:	;	EOL:	;		
R_Curly:	}	R_Curly:	}		

Žetoni, ki jih sintaktični analizator prejme, so shranjeni v vrsti (angl. queue). Analizator vzame prvi žeton iz vrste, ga analizira in postopek ponavlja, dokler ne pride do žetona

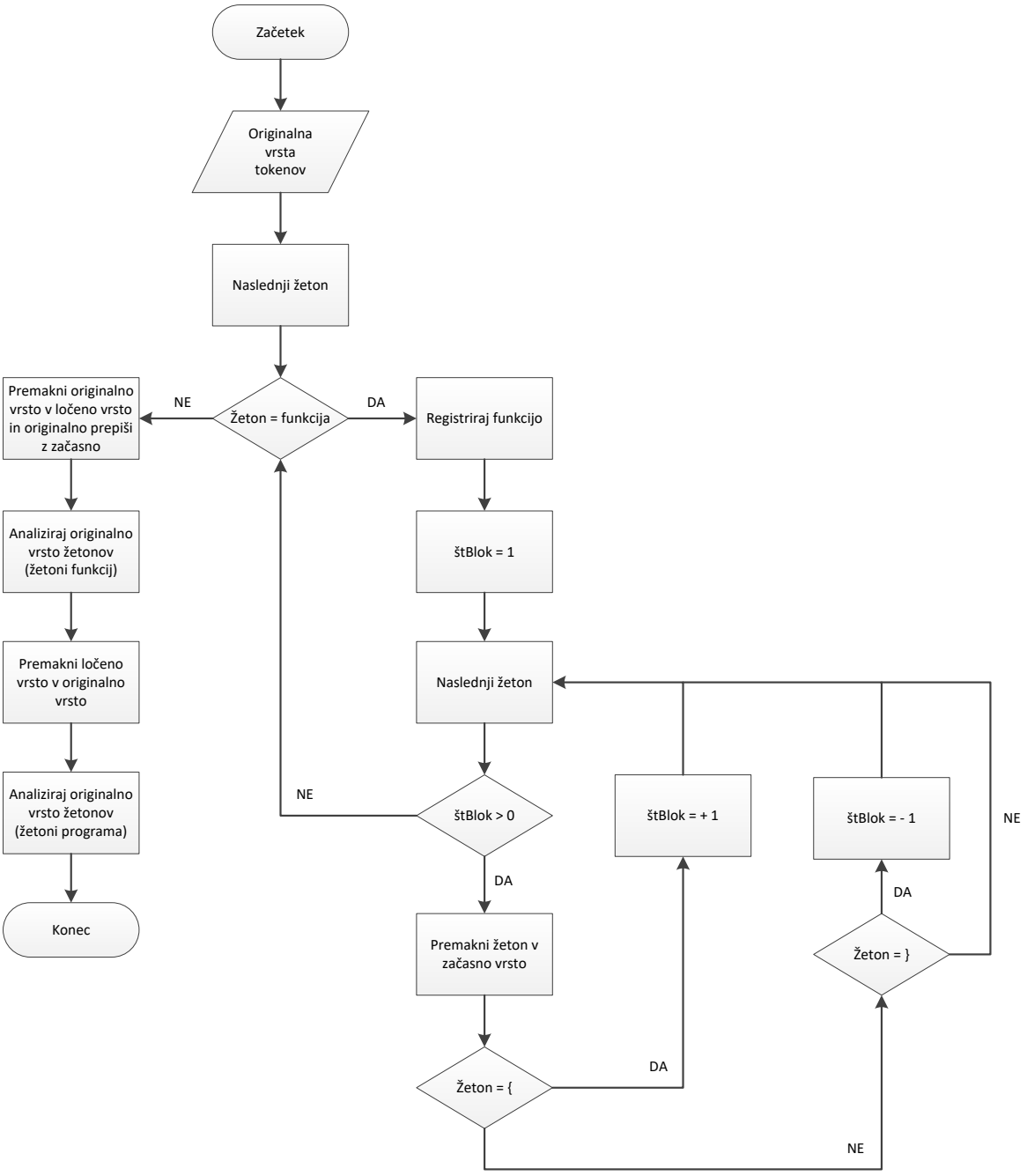
End-Of-File (EOF). Ker so vse funkcije deklarirane na začetku izvorne kode, najprej preveri, ali je dobil žeton, ki označuje funkcijo (Vrsta žetona: Function; vrednost žetona: funkcija). Če je tako, preide v metodo za analizo in shranjevanje funkcij. Ko analizira celoten podpis funkcije, vse žetone, ki predstavljajo njen blok, premakne v začasno vrsto. Postopek ponavlja za vse funkcije, dokler ne dobi žetona Program. Takrat originalno vrsto žetonov shrani v ločeno vrsto, originalno pa prepiše z začasno vrsto, kjer so žetoni funkcij in jih analizira. Po končanem analiziranju vseh blokov funkcij ponovno napolni originalno vrsto s preostalimi žetoni in analizira še program.

Pri premikanju žetonov funkcij v začasno vrsto štejemo zavite oklepaje. Vemo, da se blok začne z levim zavitim oklepajem, zato postavimo števec na 1. Ko dobimo desni zaviti oklepaj, števcu odštejemo 1. Žetone premikamo, dokler je števec večji od 0. Primer:

```
funkcija celo VrniVečje(celo a, celo b)
{                                              štBlok = 1
      če(b > a)
      {                                        štBlok = 2
            a = b;
      }                                        štBlok = 1

      vrni a;
}                                              štBlok = 0
```

Zadnji zaviti oklepaj postavi števec blokov na 0. Tako vemo, da smo premaknili vse žetone, ki pripadajo bloku funkcije. Če je kje manjkajoči zaviti oklepaj, bo sintaktični analizator kasneje tekom analize to tudi ugotovil.

Na sliki 17 je procesna shema zgoraj opisanega postopka registracije in sintaktične analize funkcij. Na desni strani sheme je proces registracije funkcije in shranjevanja žetonov v ločeno vrsto, na levi pa analiza bloka funkcije in programa.



Slika 17: Procesna shema registracije in sintaktične analize funkcij

Slika 18 prikazuje kodo metode RunParser, ki je vstopno mesto v sintaktično analizo. Med vrsticama 48 in 51 vidimo klic metode za registracijo funkcij (RegisterFunction), ki se ponovi za vsako funkcijo. Vrstica 53 originalno vrsto žetonov shrani v ločeno vrsto, naslednja vrstica pa analizira žetone, ki pripadajo blokom funkcij. Zatem v vrstici 55 v originalno vrsto ponovno premaknemo žetone iz ločene vrste in v nadaljevanju analiziramo program.

```
38
39      // ENTRY POINT
40      // If there are any functions defined, they are registered first and their block tokens are stored
41      // in the IntermediateFunction object. After registering all the functions, stored tokens are parsed.
42      // Program is parsed last.
        1 reference
43      internal AstBlock RunParser(Queue<Token> tokens, SymbolTable syt)
44      {
45          _tokenQueue = tokens;
46          SyT = syt;
47
48          while (PeekToken(TokenType.Function))
49          {
50              RegisterFunction(true);
51          }
52
53          _tokenQueueTmp = _tokenQueue;
54          ParseFunctions();
55          _tokenQueue = _tokenQueueTmp;
56
57          NextToken();
58          if (_token.Type != TokenType.Program)
59          {
60              throw new UnexpectedTokenException(_token);
61          }
62          NextToken();
63
64          AstBlock program = Block();
65          ExpectToken(TokenType.EOF);
66
67          return program;
68      }
```

Slika 18: Vstopno mesto v sintaktično analizo

V metodi RegisterFunction kličemo metodo AllocateTokens, ki shrani žetone bloka funkcije v začasno vrsto. Metoda AllocateTokens je prikazana na sliki 19.

```
        1 reference
118     private void AllocateTokens(IntermediateFunction function)
119     {
120         function.SaveToken(_token);
121
122         int blockCount = 1;
123         while (blockCount > 0)
124         {
125             NextToken();
126             function.SaveToken(_token);
127             switch (_token.Type)
128             {
129                 case TokenType.L_Curly:
130                     blockCount++;
131                     break;
132                 case TokenType.R_Curly:
133                     blockCount--;
134                     break;
135             }
136         }
137     }
```

Slika 19: Metoda za začasno shranjevanje žetonov bloka funkcije

Objekt function, razreda IntermediateFunction, je uporabljen samo začasno. Vsaka nova funkcija instancira nov objekt, kjer se hrani njeno ime, podatek ali funkcija vrača in vrsta (angl. queue) žetonov.

Po registraciji in analizi žetonov funkcije to shranimo v objekt razreda Function. Njegova polja so na sliki 20.

```
internal string Name;
internal List<AstParameter> Parameters = new List<AstParameter>();
internal AstBlock Block;
internal bool Returns;
internal DataType ReturnType;
internal Ast ReturnAst;
```

Slika 20: Polja razreda Function

Vse tako instancirane funkcije se shranijo v tabelo simbolov. Pri klicu funkcije je ne pripnemo sintaksnemu drevesu programa, temveč ustvarimo samo klic funkcije. Ta je zajet v razredu AstFunctionCall, katerega polja so na sliki 21.

```
internal string Name;
internal List<AstParameter> Parameters = new List<AstParameter>();
internal DataType ReturnType;
```

Slika 21: Polja razreda AstFunctionCall

Pri klicu moramo poznati zgolj ime funkcije, ki jo kličemo, s katerimi argumenti jo kličemo (shranjeni v listo) in kakšen je podatkovni tip vračanja. Podatkovni tip vračanja potrebuje semantični analizator pri preverjanju ujemanja tipov. Ko tolmač (razred Interpreter) naleti na klic funkcije, jo v tabeli simbolov poišče po imenu in izvede kodo, ki je shranjena v njenem polju Block.

5.4 SEMANTIČNI ANALIZATOR

Naloga semantičnega analizatorja je razbrati pomen programa (do neke mere), ki ni razviden iz njegove sintakse. Sintaksa se namreč ne ukvarja s pomenom programa, ampak samo z njegovo skladnjo. Slovenski stavek "Okrogla kocka je trikotna." je sintaktično

pravilen, vendar je brez pomena. Take sintaktično pravilne programske stavke, ki pa so brez pomena, ujame semantični analizator, še preden se program dejansko izvede.

Semantični analizator od sintaktičnega analizatorja prejme sintaksno drevo. Preveri ujemanje podatkovnih tipov in doseg spremenljivk. V računskih izrazih preveri tudi prisotnost deljenja s konstanto 0 (nič). Izvedba semantičnega analizatorja je zelo podobna tolmaču, le da programa ne izvaja. Pri računskih izrazih to pomeni, da izraza ne izračuna, temveč samo preveri tipe spremenljivk in konstant. Tudi ne prepozna, da deljenje z izrazom "(x - x)" v resnici pomeni deljenje z 0. Takšna napaka se pokaže šele med samim izvajanjem, torej v tolmaču. Se pa semantični analizator po sintaksnemu drevesu premika na isti način kot tolmač.

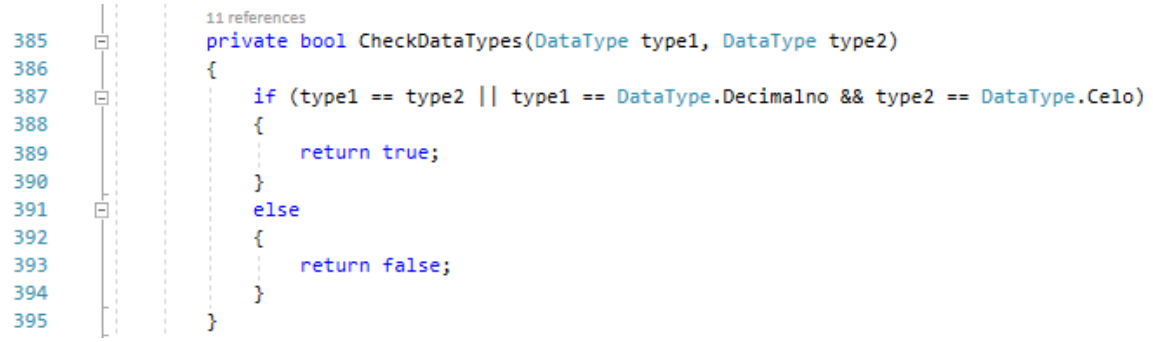
Na primeru pokažimo semantične napake, ki jih semantični analizator prepozna. Spodaj je koda v JanC, ki vsebuje kar nekaj semantičnih napak, ampak je kljub temu sintaktično pravilna. Z rdečo so označene semantične napake.

```
funkcija niz VrniNiz(celo koliko)
{
	niz zaVrniti = "Vrednost je " + koliko;
	vrni zaVrniti;
}

Program
{
	celo a = 1 + VrniNiz(6.5);

	b = 3.4 / 0;
	decimalno b;

	če(a > "vrednost")
	{
		celo d = 1;
	}
	sicer
	{
		celo d = 2;
	}

	Izpiši(d);
}
```

Najprej želimo v programu spremenljivki "a" prirediti vrednost, ki jo vrača funkcija VrniNiz. Ta vrača vrednost podatkovnega tipa niz, spremenljivka "a" pa je podatkovnega tipa celo. Hkrati funkciji VrniNiz podajamo argument tipa decimalno, njen parameter pa je tipa celo. Potem je spremenljivka "b" uporabljena, preden je deklarirana, pri prirejanju vrednosti pa še delimo z 0. V pogoju "če" stavka primerjamo vrednost spremenljivke "a" (podatkovni tip celo) in niza. Nazadnje želimo izpisati vrednost spremenljivke "d", ki je sicer pred tem že deklarirana v "če"/"sicer" stavku, vendar izven njunih blokov ni dosegljiva. Ena izmed pomensko pravilnih variacij zgornjega programa bi bila kot v nadaljevanju. Z zeleno so označene spremembe oz. dopolnitve.

```
funkcija niz VrniNiz(decimalno koliko)
{
	niz zaVrniti = "Vrednost je " + koliko;
	vrni zaVrniti;
}

Program
{
	niz a = "1" + VrniNiz(6.5);

	decimalno b;
	b = 3.4;

	celo d = 2;

	če(a > "vrednost")
	{
		d = 1;
	}

	Izpiši(d);
}
```

Na sliki 22 je metoda, ki preverja ujemanje podatkovnih tipov. Najbolj običajen primer je, da je prvi argument metode tip spremenljivke, ki ji prirejamo vrednost, drugi argument pa tip same vrednosti. V vrstici 387 določimo, da lahko vedno prirejamo vrednost med istima podatkovnima tipoma, dopustimo pa tudi prirejanje vrednosti tipa celo spremenljivki tipa decimalno.



Slika 22: Metoda za preverjanje ujemanja podatkovnih tipov

Naša izvedba semantičnega analizatorja ne prepozna pomenov programa, ki se lahko razkrijejo zgolj med samim izvajanjem. Ena takšnih pomenskih napak je že omenjeno deljenje z izrazom, katerega rezultat je 0. Podobno tudi ne prepozna zanke, katere izvajanje se nikoli ne zaključi. Takšna zanka je na primer:

```
dokler(1 == 1)
{
	Izpiši("Pozdravljen, svet!");
}
```

Visual Studio, ki se uporablja kot integrirano razvojno okolje za C#, takšno zanko prepozna in vso kodo za njo označi kot nedosegljivo. To sicer ni pomenska napaka, ki preprečuje prevajanje in izvajanje programa, je pa kljub temu takšna, da se ji želimo izogniti. Semantični analizator prav tako ne prepozna pomena naslednjega programa:

```
celo a;
če(1 == 1)
{
	a = 1;
}
Izpiši(a);
```

Očitno je, da bo v vseh primerih spremenljivka "a", ko jo podamo kot argument metodi Izpiši, inicializirana, vendar bo semantični analizator kljub temu javil napako, da ni inicializirana. Program bi bil pravilen, če bi jo skupaj z deklaracijo tudi inicializirali.

5.5 TOLMAČ

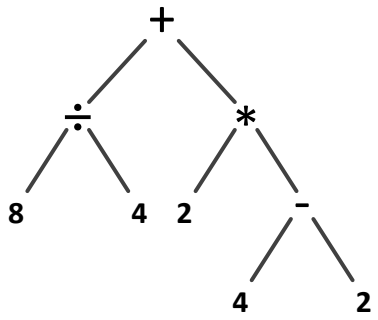
Tolmač je zadnja faza v tolmačenju jezika JanC. Za delovanje ravno tako kot semantični analizator potrebuje sintaksno drevo, ki ga je zgradil sintaktični analizator. Kot že omenjeno, je njegova izvedba zelo podobna semantičnemu analizatorju. Razlika je v tem, da program samo še izvaja in ne preverja podatkovnih tipov in dosegov spremenljivk. V nadaljevanju predstavimo tolmačenje matematičnega izraza in klica registriranih metod.

5.5.1 Računanje izraza

Za prikaz delovanja tolmačenja ali računanja matematičnega izraza uporabimo izraz, ki smo ga navedli v opisu delovanja sintaktičnega analizatorja.

```
8 / 4 + 2 * (4 - 2)
```

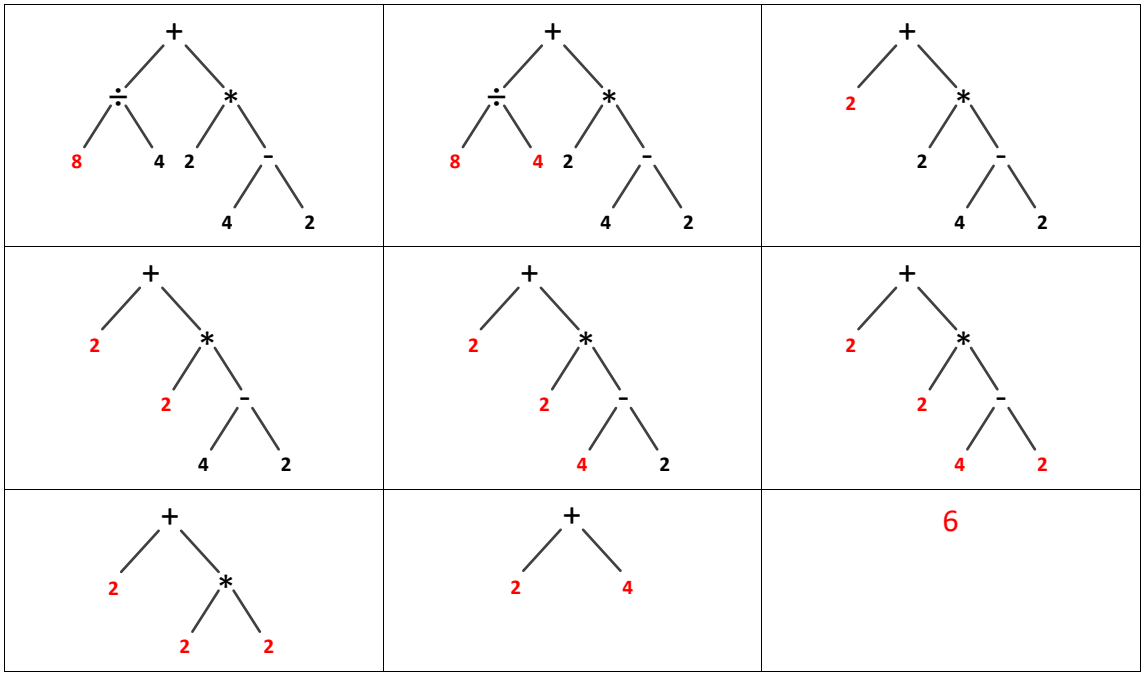
Sintaktična analiza je iz izraza zgradila sintaksno drevo, ki je prikazano na sliki 23.



Slika 23: Sintaksno drevo izraza

Tolmač drevo izraza izračuna tako, da najprej vzame levi operand, nato desnega in nad njima izvede operacijo, ki je v korenu drevesa oz. vozlišča. Vedno najprej preveri levo stran. Če najde literal (konstanto) ali identifier (spremenljivko), ga shrani v objekt, ki je osnovnega tipa v C#, tj. object. Če pa najde nov izraz, se torej nahaja v vozlišču in postopek rekurzivno ponovi. Tako deluje, dokler ne pride do listov drevesa, kjer se nahajajo samo literali ali identifierji. Ko ima shranjen levi objekt, isti postopek ponovi za desnega. Primer računanja našega izraza je v tabeli 7.

Tabela 7: Računanje izraza

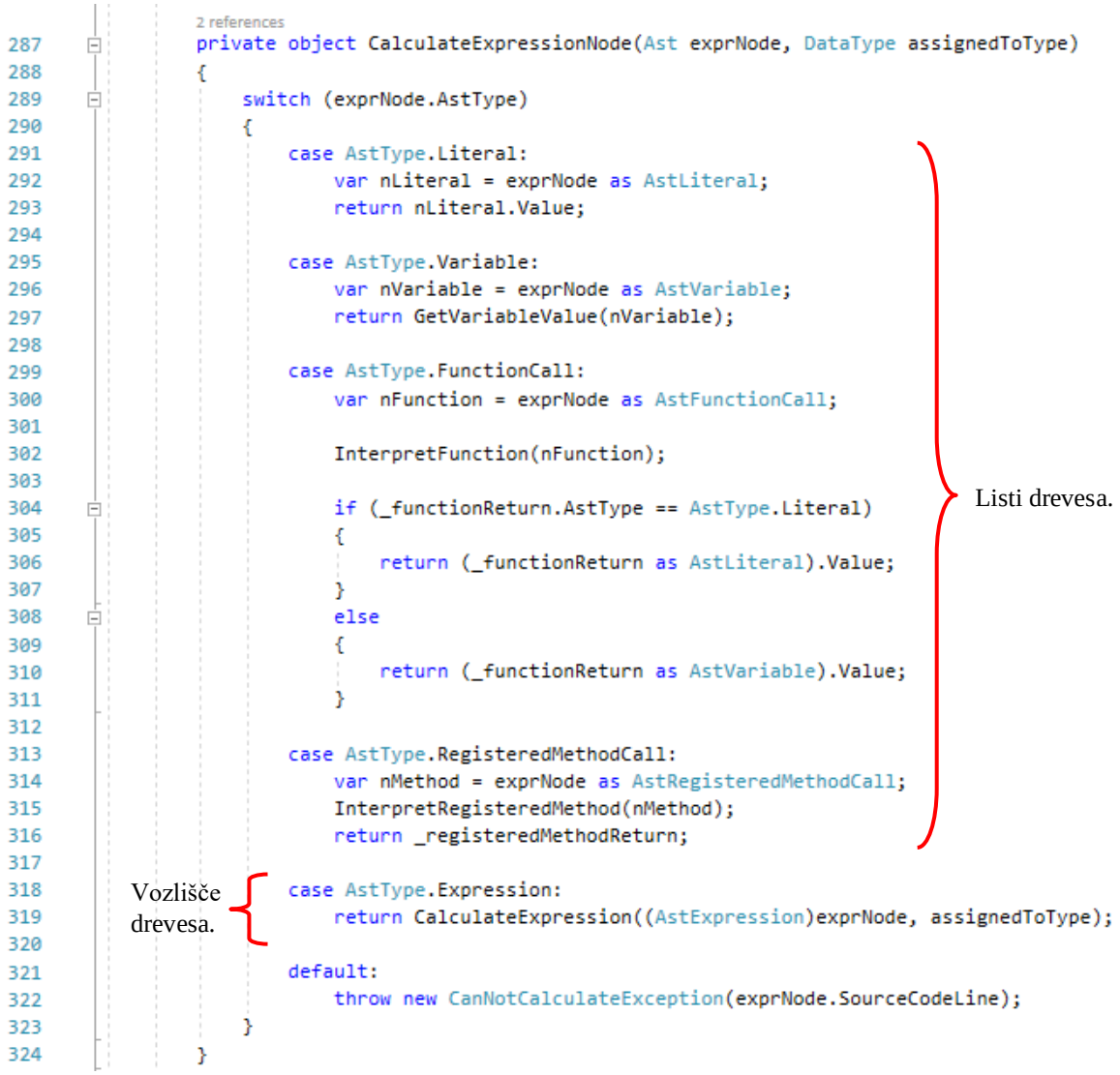


Tolmač drevesa v resnici ne uniči, kot smo to prikazali v postopku, saj si dobljene rezultate shranjuje v ločene objekte. Metoda CalculateExpression na sliki 24 je prvi korak za izračun izraza (angl. expression).

```
2 references
275     private object CalculateExpression(AstExpression expression, DataType assignedToType)
276     {
277         object left = null;
278         if (expression.Left != null)
279         {
280             left = CalculateExpressionNode(expression.Left, assignedToType);
281         }
282
283         object right = CalculateExpressionNode(expression.Right, assignedToType);
284         return CalculateBinary(left, right, expression.Operator, assignedToType, expression.SourceCodeLine);
285     }
```

Slika 24: Metoda za izračun izraza (1. korak)

Izraz vsebuje levi in desni del ter operator. Če prirejamo vrednost, ki ni v obliki računa (samo en literal ali identifier), je v izrazu poln samo desni del. To v kodi preverimo v vrstici 278. Če levi del izraza ni prazen (angl. null), njegovo vrednost dobimo s klicem metode CalculateExpressionNode, ki je na sliki 25. Isto metodo uporabimo za desni del izraza (vrstica 283). Dejanska vrednost izraza se izračuna v metodi CalculateBinary, ki jo kličemo v vrstici 284.



Slika 25: Metoda za izračun izraza (2. korak)

5.5.2 Izvajanje registriranih metod

Registrirane metode so napisane v C# in so v okolje JanC vstavljene v obliki delegatov. Ker ne vemo vnaprej, katera metoda se bo klicala in kakšen je njen podpis, jih moramo klicati dinamično. To pomeni, da klic metode povežemo z metodo šele med samim izvajanjem.

Registrirana metoda je instanca razreda RegisteredMethod, ki je prikazan na sliki 26.

```
// Holds a registered method information.
8 references
internal class RegisteredMethod
{
    internal string Name;
    internal Delegate Method;
    internal List<DataType> ParametersTypes = new List<DataType>();
    internal DataType ReturnType;
    internal bool Returns = true;

    private Dictionary<Type, DataType> _dataTypesMapping = new Dictionary<Type, DataType>
    {
        { typeof(int), DataType.Celo },
        { typeof(double), DataType.Decimalno },
        { typeof(string), DataType.Niz }
    };

    1 reference
    internal RegisteredMethod(string methodName, Delegate method)
    {
        Name = methodName;
        Method = method;

        foreach (ParameterInfo parameter in method.Method.GetParameters())
        {
            Type pType = parameter.ParameterType;
            ParametersTypes.Add(_dataTypesMapping[pType]);
        }

        Type rType = method.Method.ReturnType;
        if (_dataTypesMapping.ContainsKey(rType))
        {
            ReturnType = _dataTypesMapping[rType];
        }
        else
        {
            Returns = false;
        }
    }
}
```

Slika 26: Razred RegisteredMethod

Metoda v C# je preko delegata shranjena v polju Method. Listo podatkovnih tipov parametrov (ParametersTypes) dobimo z uporabo refleksije, ki v času izvajanja preveri tipe parametrov (v okolju C#) in jih preslikamo v podatkovne tipe okolja JanC.

Klic registrirane metode je shranjen v objektu razreda AstRegisteredMethodCall, čigar polji sta na sliki 27. Poznati moramo samo ime metode, ki jo kličemo, in podane argumente.

```
internal string Name;
internal List<Ast> PassedValues = new List<Ast>();
```

Slika 27: Polji razreda AstRegisteredMethodCall

Tolmač metodo zažene tako, da jo preko imena najde v tabeli simbolov, vrednost podanih argumentov shrani v tabelo (angl. array) in metodo dinamično pokliče. Dinamičen klic metode oz. delegata, ki to metodo drži, kot argument sprejme tabelo objektov, ki predstavljajo argumente delegata (metode). Izvedba klica registrirane metode je na sliki 28.

```
2 references
private void InterpretRegisteredMethod(AstRegisteredMethodCall node)
{
    RegisteredMethod rMethod = SyT.RegisteredMethods[node.Name];

    // List of values passed to a method.
    var argsList = new List<object>();

    // Populates argsList with values, that are later passed to a method as an object[].
    // Method signature is known only at runtime so it has to be invoked dynamicaly, which also requires
    // dynamicaly preparing the passed values.
    for (int pIndex = 0; pIndex < node.PassedValues.Count; pIndex++)
    {
        switch (node.PassedValues[pIndex].AstType)
        {
            case AstType.Literal:
                var literal = node.PassedValues[pIndex] as AstLiteral;
                argsList.Add(literal.Value);
                break;

            case AstType.Variable:
                var variable = GetVariable((node.PassedValues[pIndex] as AstVariable).Name);
                argsList.Add(variable.Value);
                break;
        }
    }

    try
    {
        _registeredMethodReturn = rMethod.Method.DynamicInvoke(argsList.ToArray());
    }
    catch (Exception e)
    {
        throw new RegisteredMethodExternalException(node.Name, e.InnerException.Message, node.SourceCodeLine);
    }
}
```

Pridobivanje metode iz tabele simbolov.

Pridobivanje vrednosti podanih argumentov.

Dinamičen klic metode.

Slika 28: Klic registrirane metode

Če se napaka zgodi tekom izvajanja registirane metode, jo ujamemo in podamo dalje znotraj varnostnega bloka try/catch.

6 PRIMER UPORABE JEZIKA JANC

Jezik JanC lahko na primer uporabimo kot vmesnik v aplikaciji, ki njenemu uporabniku omogoča programiranje poslovne logike.

Zamislili smo si preprosto konzolno aplikacijo za izračun premije za življenjsko zavarovanje. Aplikacija je razvita v C#, poslovna logika, ki izračuna premijo, pa je napisana v JanC. V aplikaciji inicializiramo tri objekte iz razreda Zavarovalec, ki predstavlja zavarovano osebo z imenom, letom rojstva in leti uporabe storitev zavarovalnice. V okolje JanC registriramo metode, ki dostopajo do podatkov osebe. V JanC so določeni osnova in faktorji za izračun premije, po izračunu pa rezultat s klicem registrirane metode vrnemo aplikaciji, ki znesek premije izpiše v konzolo.

Inicializacija oseb nadomešča pridobivanje oseb iz baz podatkov, kot bi to delovalo v realnosti. Poudarimo, da primer ne temelji na nobenem realnem primeru in da so osnova za izračun premije, faktorji in način izračuna premije v polnosti plod naše domišljije ter namenjeni zgolj prikazu delovanja JanC. Na sliki 29 je razred Zavarovanec.

```
12 references
9   class Zavarovanec
10  {
11      public string Ime;
12      public int LetoRojstva;
13      public int LetaZvestobe;
14
        1 reference
15      public Zavarovanec() { }
16
        3 references
17      public Zavarovanec(string ime, int letoRojstva, int letaZvestobe)
18      {
19          Ime = ime;
20          LetoRojstva = letoRojstva;
21          LetaZvestobe = letaZvestobe;
22      }
23  }
```

Slika 29: Razred Zavarovanec

Slika 30 prikazuje metodo Main. V vrstici 91 preberemo izvorno kodo JanC, nato inicializiramo razred JanC, registriramo metode in zaženemo tolmač (vrstica 101).

Slika 30: Program aplikacije za izračun premije

Celotno definicijo registriranih metod, inicializacije razreda JanC, registracijo metod in klic tolmača smo ovili v varnostni blok try/catch, da lahko izpišemo napake, ki jih javimo tekom tolmačenja. Spodaj je koda v JanC, ki izračuna premijo in vrednost vrne aplikaciji. Kodo smo za lažje branje obarvali sami.

```
funkcija decimalno FaktorNaStarost(celo starost)
{
	# Faktor za do 25 let starosti
	decimalno faktor = 1;

	če(starost > 25)
	{
		če(starost < 50)
		{
			faktor = 1.1;
		}
		sicer
		{
			faktor = 1.2;
		}
	}

	Izpiši("Faktor na starost " + faktor);
	vrni faktor;
}

funkcija decimalno FaktorNaLetaZvestobe(celo letaZvestobe)
{
	# Faktor za manj kot 5 let zvestobe
	decimalno faktor = 1;

	če(letaZvestobe > 5)
	{
		če(letaZvestobe < 15)
		{
			faktor = 0.9;
		}
		sicer
		{
			faktor = 0.8;
		}
	}

	Izpiši("Faktor na leta zvestobe: " + faktor);
	vrni faktor;
}
```

```
funkcija decimalno FaktorNaZavarovalnoDobo(celo trajanje)
{
	# Faktor za 1 leto zavarovalne dobe
	decimalno faktor = 1.5;

	če(trajanje == 10)
	{
		faktor = 1.2;
	}

	če(trajanje == 100)
	{
		faktor = 1.05;
	}

	Izpiši("Faktor na zavarovalno dobo: " + faktor);
	vrni faktor;
}

Program
{
	Izpiši("----------------------- Debug");

	# Podatki o zavarovancu
	celo starost = TrenutnoLeto() - LetnikZavarovanca();
	celo letaZvestobe = LetaZvestobe();
	celo zavarovalnaDoba = ZavarovalnaDoba();

	Izpiši("Starost: " + starost);
	Izpiši("Leta zvestobe: " + letaZvestobe);
	Izpiši("Zavarovalna doba: " + zavarovalnaDoba);
	Izpiši("");

	# Osnova za izračun premije
	decimalno osnova = 25;
	Izpiši("Osnova: " + osnova);

	decimalno premija =
		osnova
		* FaktorNaStarost(starost)
		* FaktorNaLetaZvestobe(letaZvestobe)
		* FaktorNaZavarovalnoDobo(zavarovalnaDoba);

	VrniPremijo(premija);

	Izpiši("----------------------- Debug");
}
```

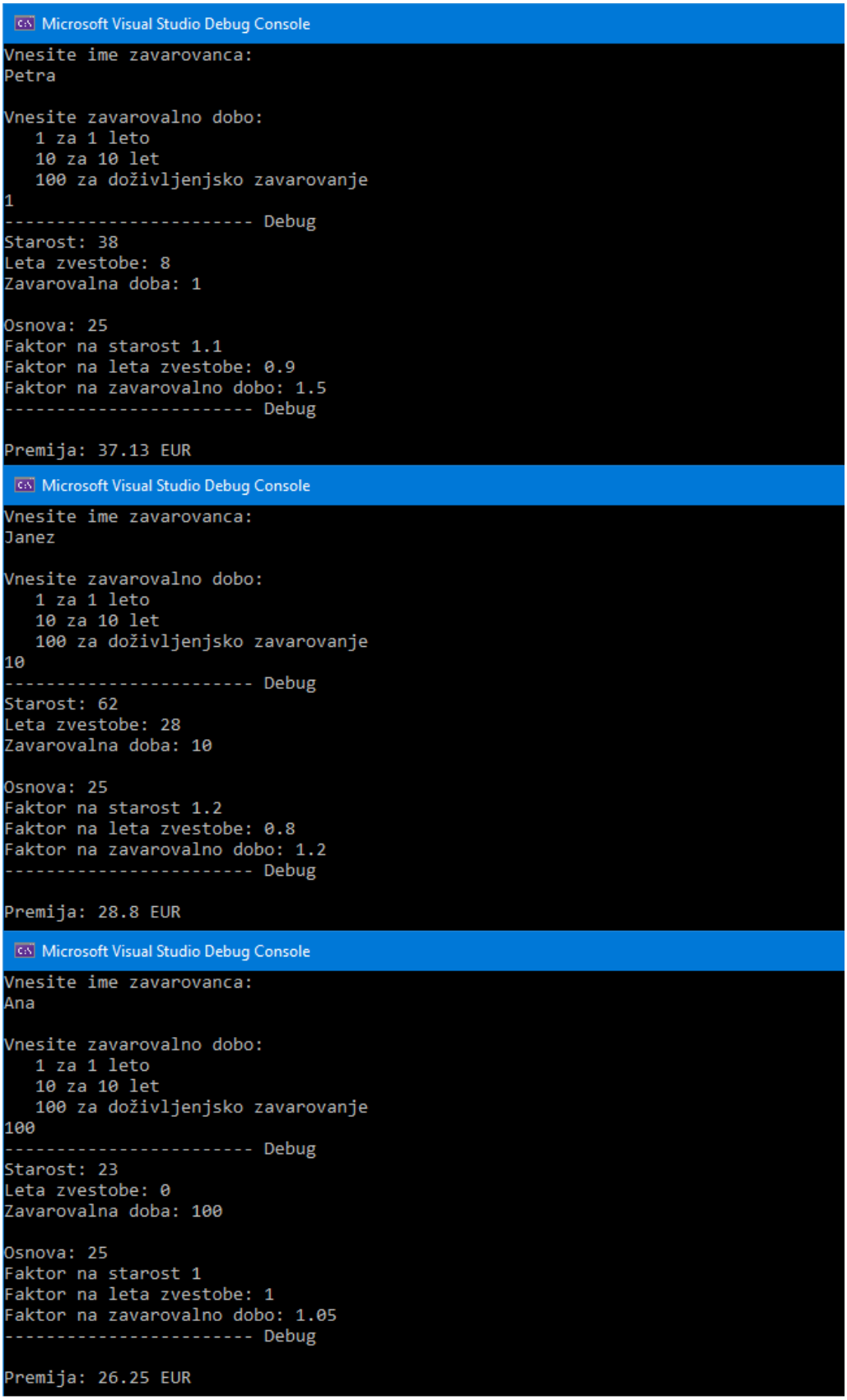
Za razumljivejši prikaz iz programa JanC tudi izpisujemo (v konzoli na sliki 31 izpis med Debug). Če želimo v kodi JanC pustiti vse klice za izpis, toda nočemo, da bi se vrednost tudi dejansko izpisala, lahko samo zakomentiramo registracijo metode za izpis v metodi

Main (vrstica 94 na sliki 30). Pred izvajanjem programa pa v tabeli 8 navedimo vse podatke, ki jih potrebujemo za izračun premije. Napisane so tudi pričakovane premije.

Tabela 8: Podatki za izračun premije

Zavarovane osebe			
Zavarovanec	**Leto rojstva**	**Starost**	**Leta zvestobe**
Petra	1985	38	8
Janez	1961	62	28
Ana	2000	23	0
Faktor na starost zavarovanca			
Starost	**Faktor**		
Do 25	1		
Med 25 in 50	1,1		
Več kot 50	1,2		
Faktor na leta zvestobe			
Leta zvestobe	**Faktor**		
Do 5	1		
Med 5 in 15	0,9		
Več kot 15	0,8		
Faktor na zavarovalno dobo			
Zavarovalna doba	**Faktor**		
1 leto	1,5		
10 let	1,2		
Doživljenjsko	1,1		
Osnova za izračun premije [EUR]			
Osnova	25		
Pričakovane premije glede na osebo in izbrano zavarovalno dobo [EUR]			
Osebe	**1 leto**	**10 let**	**Doživljenjsko**
Petra	37,13	29,70	25,99
Janez	36,00	28,80	25,20
Ana	37,50	30.00	26,25

Zaženimo program in si pri vsaki osebi izberimo drugo zavarovalno dobo. Na sliki 31 so prikazani trije rezultati. Premije se ujemajo s predvidenimi, kar pomeni, da naš program deluje pravilno.



Slika 31: Rezultati aplikacije za izračun premije

7 TESTIRANJE IZVAJANJA JANC

7.1 PRIMERJAVA IZVAJANJA C# IN JANC

Primerjali smo hitrost izvajanja in velikost zasedenega pomnilnika istih programov v jezikih C# in JanC. Najprej smo primerjali posamezne funkcionalnosti, na koncu pa krajši program. Za merjenje smo uporabili knjižnico BenchmarkDotNet (https://www.nuget.org/packages/BenchmarkDotNet), ki jo dobimo z upravljalnikom paketov NuGet. BenchmarkDotNet večkrat izvede metodo, ki jo označimo za testiranje ([Benchmark] nad metodo), in izračuna povprečne vrednosti. Te na koncu testa izpiše v konzolo. Program moramo zagnati v načinu Release brez razhroščevanja, saj je zaradi slednjega izvajanje počasnejše.

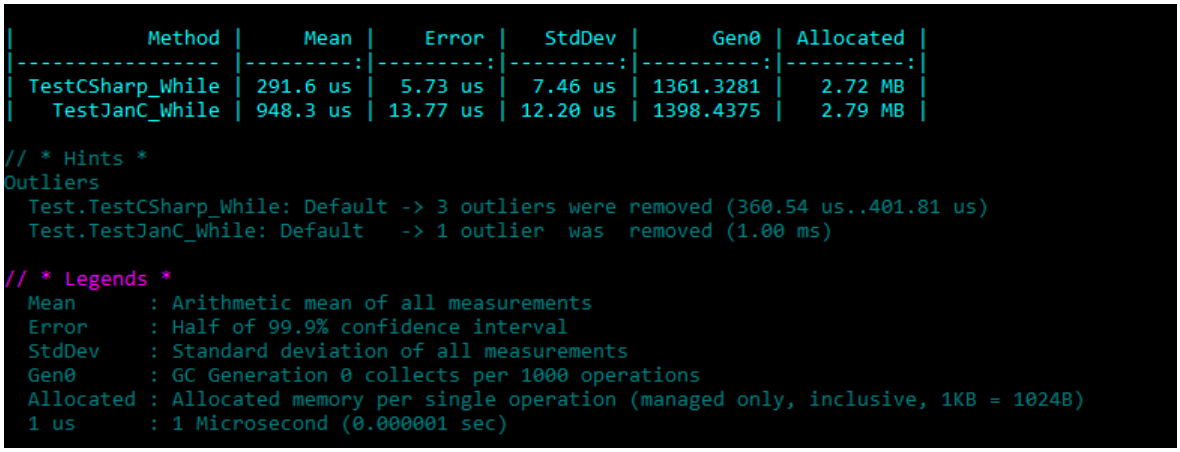
Napisali smo tri metode v C# in iste programe v JanC. Metode v C# bi lahko napisali učinkoviteje, vendar smo jih prilagodili na zmogljivosti jezika JanC, tako da so sintaktično identične kodi v JanC. V vseh testih smo uporabili zanko while. Brez nje je izvajanje tako hitro, da BenchmarkDotNet javi, da je hitrost izvajanja metode zelo podobna hitrosti izvajanje prazne metode. Na sliki 32 sta metodi, s katerima primerjamo hitrost izvajanja zanke in združevanja nizov. Pod sliko je izvorna koda v JanC.

```
[Benchmark]
0 references
public void TestCSharp_While()
{
    string append = "";
    int i = 0;

    while (i < 1000)
    {
        append = append + i;
        i = i + 1;
    }
}

[Benchmark]
0 references
public void TestJanC_While()
{
    string sourceCode = File.ReadAllText("JanC_While.txt");

    JanC program = new JanC();
    program.Run(sourceCode);
}
```

Slika 32: Primerjalni test zanke while in združevanja nizov

```
Program
{
	niz append = "";
	celo i = 0;

	dokler(i < 1000)
	{
		append = append + i;
		i = i + 1;
	}
}
```

Na sliki 33 je rezultat prvega testa. Metoda v C# je za izvajanje potrebovala povprečno 291,6 µs (mikrosekund), v jeziku JanC se je isti program izvajal povprečno 948,3 µs, kar je približno 3-krat počasneje. Poudarimo, da je metoda v C# že prevedena in se mora samo izvesti, medtem ko moramo program v JanC prebrati iz datoteke in izvorno kodo pripraviti za izvajanje. Stolpec Allocated prikazuje porabo pomnilnika. Tako C# kot JanC sta potrebovala približno enako količino pomnilnika.



```
|           Method |     Mean |    Error |   StdDev |      Gen0 | Allocated |
|----------------- |---------:|---------:|---------:|----------:|----------:|
| TestCSharp_While | 291.6 us |  5.73 us |  7.46 us | 1361.3281 |   2.72 MB |
|    TestJanC_While | 948.3 us | 13.77 us | 12.20 us | 1398.4375 |   2.79 MB |

// * Hints *
Outliers
  Test.TestCSharp_While: Default -> 3 outliers were removed (360.54 us..401.81 us)
  Test.TestJanC_While: Default   -> 1 outlier  was  removed (1.00 ms)

// * Legends *
  Mean      : Arithmetic mean of all measurements
  Error     : Half of 99.9% confidence interval
  StdDev    : Standard deviation of all measurements
  Gen0      : GC Generation 0 collects per 1000 operations
  Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
  1 us      : 1 Microsecond (0.000001 sec)
```

Slika 33: Rezultat testa zanke while in lepljenja nizov

Slika 34 prikazuje metodi, ki 1000-krat izračunata isti izraz. V izrazu tudi referenciramo spremenljivko.

```
[Benchmark]
0 references
public void TestCSharp_Calculate()
{
    int i = 0;

    while (i < 1000)
    {
        int a = 5;
        int b = 1 + 4 * (8 / (a - 3)) - 6;
        i = i + 1;
    }
}

[Benchmark]
0 references
public void TestJanC_Calculate()
{
    string sourceCode = File.ReadAllText("JanC_Calculate.txt");

    JanC program = new JanC();
    program.Run(sourceCode);
}
```

Slika 34: Primerjalni test računanja izraza

```
Program
{
	celo i = 0;

	dokler(i < 1000)
	{
		celo a = 5;
		celo b = 1 + 4 * (8 / (a - 3)) - 6;
		i = i + 1;
	}
}
```

Na sliki 35 je rezultat testa zgornjih metod. Tu je razlika očitna in znaša že več redov velikosti. Program v JanC se je izvajal več kot 2500-krat počasneje kot metoda v C#. Metoda v C# je porabila tako malo pomnilnika, da ga BenchmarkDotNet ni mogel izpisati oz. je zanemarljivo majhen. Medtem je naš jezik potreboval 0,26 MB.

```
|               Method |         Mean |        Error |       StdDev |     Gen0 | Allocated |
|--------------------- |-------------:|-------------:|-------------:|---------:|----------:|
| TestCSharp_Calculate |     359.9 ns |      7.25 ns |      9.93 ns |        - |         - |
|   TestJanC_Calculate | 968,078.4 ns | 19,132.44 ns | 16,960.41 ns | 123.0469 |  257462 B |

// * Hints *
Outliers
  Test.TestCSharp_Calculate: Default -> 3 outliers were removed (405.04 ns..436.23 ns)
  Test.TestJanC_Calculate: Default   -> 3 outliers were removed (1.05 ms..1.17 ms)

// * Legends *
  Mean      : Arithmetic mean of all measurements
  Error     : Half of 99.9% confidence interval
  StdDev    : Standard deviation of all measurements
  Gen0      : GC Generation 0 collects per 1000 operations
  Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
  1 ns      : 1 Nanosecond (0.000000001 sec)
```

Slika 35: Rezultat testa zahtevnosti računskega izraza

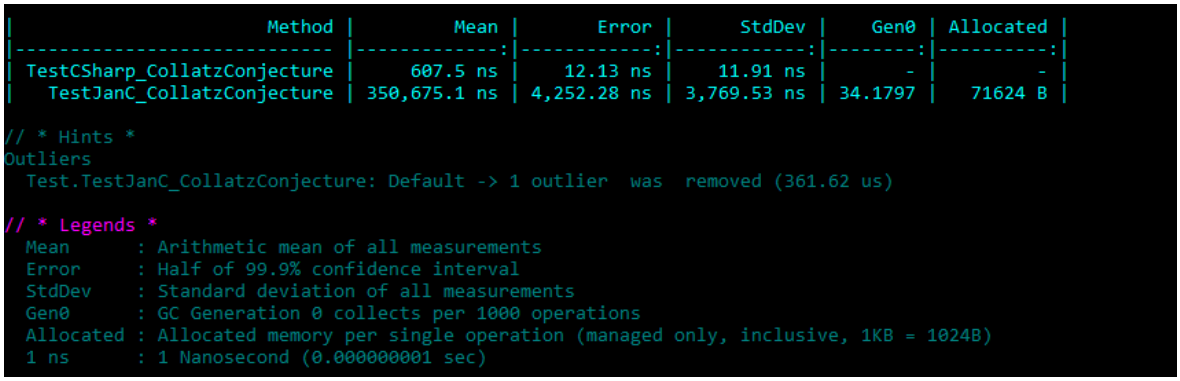
Zadnji test je izvedba Collatzeve domneve (Weisstein, 2023). Domneva trdi, da se številsko zaporedje, ki je podvrženo dvema določenima operacijama, vedno konča pri številu 1, ne glede na začetno vrednost. Zaporedje se začne s poljubnim naravnim številom. Če je število sodo, ga delimo z 2, če je liho, pa ga množimo s 3 in zmnožku prištejemo 1. Postopek ponavljamo, dokler ne pridemo do števila 1. Do danes še niso uspeli najti začetne vrednosti, ki bi domnevo ovrgla (Honner, 2020). Slika 36 prikazuje izvedbo Collatzeve domneve z začetno vrednostjo 97531.

```
[Benchmark]
0 references
public void TestCSharp_CollatzConjecture()
{
    int number = 97531;
    int divide;
    int multiply;

    while (number != 1)
    {
        divide = number / 2;
        multiply = divide * 2;

        if (multiply == number)
        {
            number = divide;
        }
        else
        {
            number = 3 * number + 1;
        }
    }
}

[Benchmark]
0 references
public void TestJanC_CollatzConjecture()
{
    string sourceCode = File.ReadAllText("JanC_CollatzConjecture.txt");

    JanC program = new JanC();
    program.Run(sourceCode);
}
```

Slika 36: Primerjalni test izvedbe Collatzeve domneve

```
Program
{
	celo number = 97531;
	celo divide;
	celo multiply;

	dokler(number != 1)
	{
		divide = number / 2;
		multiply = divide * 2;

		če(multiply == number)
		{
			number = divide;
		}
		sicer
		{
			number = 3 * number + 1;
		}
	}
}
```

Rezultat testa je na sliki 37. Tudi tu je jezik JanC počasnejši, in to skoraj 600-krat. Ponovno je C# potreboval neznatno količino pomnilnika, medtem ko je JanC vzel 0,07 MB.



Method	Mean	Error	StdDev	Gen0	Allocated
TestCSharp_CollatzConjecture	607.5 ns	12.13 ns	11.91 ns	-	-
TestJanC_CollatzConjecture	350,675.1 ns	4,252.28 ns	3,769.53 ns	34.1797	71624 B

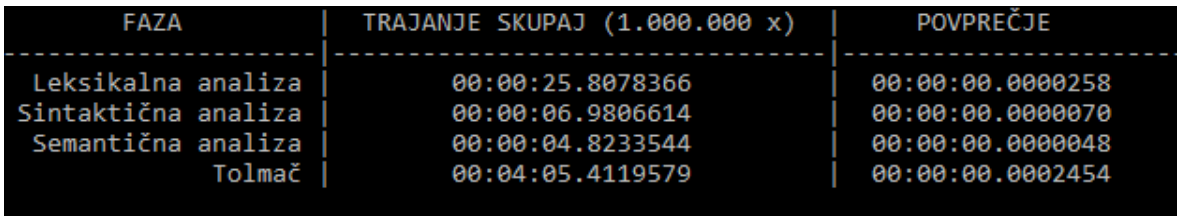
```
// * Hints *
Outliers
  Test.TestJanC_CollatzConjecture: Default -> 1 outlier  was  removed (361.62 us)

// * Legends *
  Mean      : Arithmetic mean of all measurements
  Error     : Half of 99.9% confidence interval
  StdDev    : Standard deviation of all measurements
  Gen0      : GC Generation 0 collects per 1000 operations
  Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
  1 ns      : 1 Nanosecond (0.000000001 sec)
```

Slika 37: Rezultat testa izvedbe Collatzeve domneve

Primerjalni testi so pokazali, da je naš jezik precej počasnejši od C# in da potrebuje več pomnilnika. Največja razlika je bila pri računskem izrazu. Pri prvem testu pa je bil naš jezik zgolj 3-krat počasnejši.

7.2 ČASOVNA ZAHTEVNOST FAZ TOLMAČA JANC

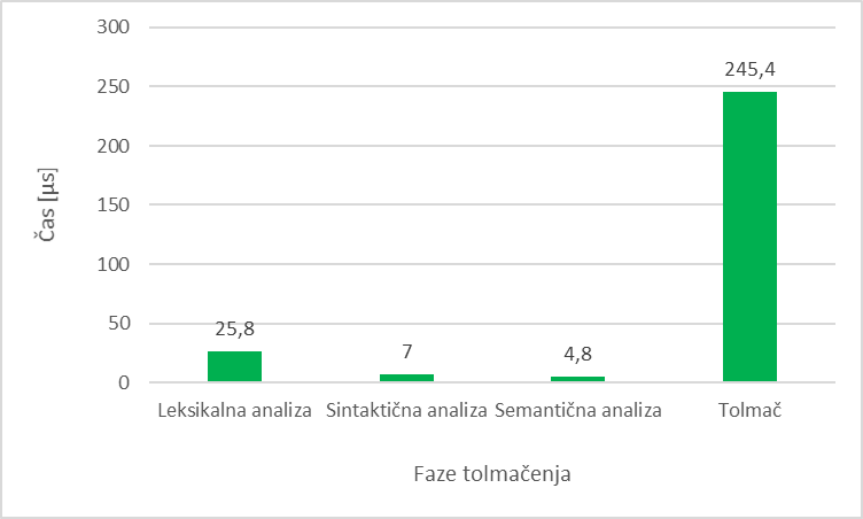
S C# razredom System.Diagnostics.Stopwatch smo merili čas izvajanja faz tolmača. Za kodo JanC smo ponovno uporabili Collatzevo domnevo. Tolmač smo v zanki pognali milijonkrat in v vsaki iteraciji merili čas posamezne faze. Na koncu smo skupen čas po fazah delili z 1.000.000 in dobili povprečen čas izvajanja. Izvajanje smo testirali v načinu Release brez razhroščevanja, rezultate pa izpisali v konzolo, kot je prikazano na sliki 38.



FAZA	TRAJANJE SKUPAJ (1.000.000 x)	POVPREČJE
Leksikalna analiza	00:00:25.8078366	00:00:00.0000258
Sintaktična analiza	00:00:06.9806614	00:00:00.0000070
Semantična analiza	00:00:04.8233544	00:00:00.0000048
Tolmač	00:04:05.4119579	00:00:00.0002454

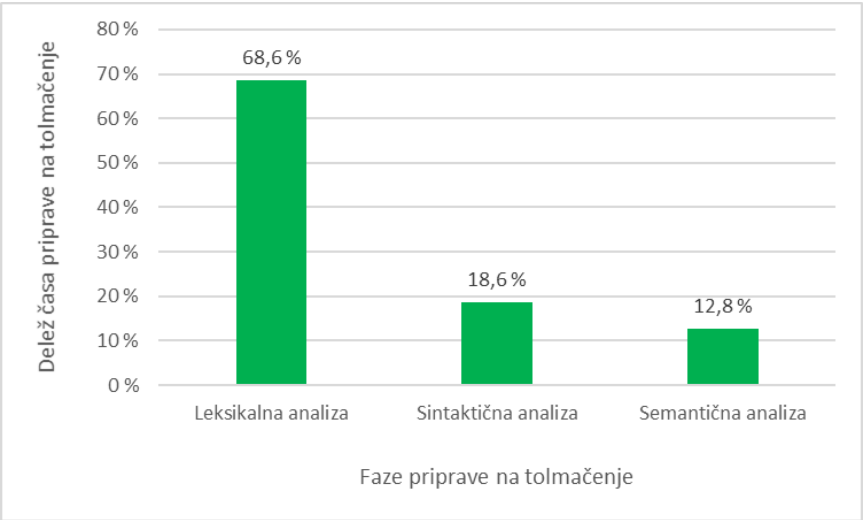
Slika 38: Rezultati testa časovne zahtevnosti faz tolmača JanC

Graf 1 prikazuje povprečne čase izvajanja posameznih faz, pretvorjene v mikrosekunde. Tolmačenje je v našem primeru vzelo poglavitni delež celotnega procesa, saj je bilo kar 86,7 % časa namenjenega njemu. Drugi časovno najzahtevnejši proces je bila leksikalna analiza.



Graf 1: Povprečni časi izvajanja faz tolmačenja

Vendar pa je čas tolmačenja odvisen od samega programa v JanC. Naš program vsebuje zanko, ki se večkrat ponovi, zato je tudi čas tolmačenja toliko daljši. Zanima nas predvsem priprava na samo tolmačenje, torej prve tri faze. Kot je razvidno iz grafa 2, leksikalna analiza vzame 68,6 % skupnega časa prvih treh faz in je kot taka ozko grlo našega procesa.



Graf 2: Deleži časa priprave na tolmačenje

8 ZAKLJUČEK

V diplomskem delu smo raziskali osnove teorije programskih jezikov in njihovega prevajanja oz. tolmačenja. Spoznanja smo uporabili za praktičen izdelek, kjer smo ustvarili lasten programski jezik in njegov tolmač.

Tolmačenje programskega jezika poteka skozi več faz. Začne se z leksikalno analizo, ki prebere niz z izvorno kodo in iz nje ustvari žetone. Naslednja faza je sintaktična analiza, ki preverja skladnjo izvorne kode in iz žetonov ustvari abstraktno sintaksno drevo. Tega zatem v fazi semantične analize pregledamo in ugotovimo morebitne pomenske napake. Nazadnje program v obliki sintaksnega drevesa dejansko izvedemo.

Jezik lahko tudi nadaljnje razvijamo. Test izvajanja JanC je pokazal, da je časovno zamuden predvsem v fazi leksikalne analize. Operacije z nizi so sicer procesorsko zahtevne, a bi kljub temu prvi nadaljnji razvoj temeljil predvsem na optimizaciji te faze. Hkrati bi leksikalno analizo spremenili v podproces sintaktične analize, kot je to običajno v praksi, in s tem prihranili čas pri programih, ki bi javili sintaktično napako. Popolnoma odveč je izvesti celotno leksikalno analizo, če bo že kmalu na začetku sintaktični analizator našel napako in ustavil izvajanje.

V nadaljevanju bi za začetek lahko dodali podatkovni tip bool in tabelo (angl. array) ali listo (angl. list). Prav tako bi lahko pogoje v "če" in "dokler" stavkih spremenili tako, da bi lahko navedli več pogojev, ki bi jih združevali z logičnimi operatorji. Podobno kot definicije funkcij pa bi lahko implementirali tudi razrede, iz katerih bi inicializirali objekte. Da bi bila sintaksa v polnosti slovenska, pa bi morali za decimalno ločilo določiti vejico, in ne pike, kot je v trenutni verziji.

Tako dodatno razvijan jezik bi postajal čedalje bolj časovno in prostorsko kompleksen, zato bi se morali še bolj poglobiti v teorijo programskih jezikov in vedno imeti v mislih optimizacijo.

Izdelava lastnega programskega jezika terja kar nekaj raziskovanja, vztrajnosti in učenja iz napak. Med ustvarjalcem jezika in računalnikom se vzpostavi "intimen" odnos, če si smemo privoščiti ta izraz. Ustvarjalec namreč poskuša računalniku razložiti pomen nečesa zanj popolnoma novega, s čimer bolje spoznava delovanje računalnika in preizprašuje svoj način razmišljanja.

8.1 DOSEGANJE NAMENOV IN CILJEV

Tolmač jezika JanC je pod imenom JanCLang dosegljiv preko upravljalnika paketov NuGet (https://www.nuget.org/packages/JanCLang/) ali na platformi GitHub (https://github.com/JanCotar/JanCLang). Oba dostopa sta javna in objavljena pod licenco MIT License (https://opensource.org/license/mit/), ki dovoljuje neomejeno uporabo za kakršnekoli namene z edino zahtevo, da je kopija licence vedno vključena.

V jeziku JanC in njegovem tolmaču so slovenske samo rezervirane besede in javljene napake. Ker je izvorna koda jezika javno dosegljiva, jih lahko kdorkoli spremeni v poljuben jezik in s tem ponudi novo lokalizirano verzijo jezika JanC.

Dosegli smo vse cilje, ki smo si jih zadali, in sicer:

- ustvarili smo programski jezik s slovensko sintakso;
- v ustvarjenem jeziku smo omogočili delo s spremenljivkami treh različnih podatkovnih tipov, vejitve, zanke, metode za vnos in izpis, računanje algebraičnih izrazov in definiranje poljubnih funkcij;
- z mehanizmom registracije metod, napisanih v C#, smo omogočili razširitve;
- v C# smo razvili tolmač za jezik in
- tolmač smo v obliki knjižnice javno ponudili preko upravljalnika paketov NuGet ali platforme GitHub.

9 LITERATURA IN VIRI

Aho, A. V. et al. *Compilers: principles, techniques, and tools*. ZDA: Edison-Wesley, 2007.

Bassil, Y. Phoenix – The Arabic Object-Oriented Programming Language (online). (citirano 28. 2. 2023). Dostopno na naslovu: https://arxiv.org/ftp/arxiv/papers/1907/1907.05871.pdf#:~:text=Phoenix%20is%20a%20General%2DPurpose,experience%20in%20the%20Arabic%20language

BenchmarkDotNet. *Paket NuGet* (online). 2023. (citirano 24. 3. 2023). Dostopno na naslovu: https://www.nuget.org/packages/BenchmarkDotNet

Catrobat. *Domača stran* (online). 2021. (citirano 28. 2. 2023). Dostopno na naslovu: https://catrobat.org/

Citrine. *Domača stran* (online). 2021. (citirano 28. 2. 2023). Dostopno na naslovu: https://citrine-lang.org/

Dasgupta, S. in Hill, B. M. *Learning to Code in Localized Programming Languages* (online). 2017. (citirano 1. 3. 2023). Dostopno na naslovu: https://dl.acm.org/doi/10.1145/3051457.3051464

Gabbrielli, M., in Martini, S. *Programming Languages: Principles and Paradigms*. London, Anglija: Springer-Verlag London Limited, 2010.

Generali. *Skupina Generali* (online). 2023. (citirano 18. 3. 2023). Dostopno na naslovu: https://www.generali.si/skupina-generali

Hasanudin, H. *BAIK – Programming Language Based on Indonesion Lexical Parsing For Multitier Web Developement* (online). 2023. (citirano 28. 2. 2023). Dostopno na naslovu: https://media.neliti.com/media/publications/101763-EN-baika-programming-language-based-on-indo.pdf

Honner, P. The Simple Math Problem We Still Can't Solve (online). *Quanta Magazine*: 2020. (citirano 24. 3. 2023). Dostopno na naslovu: https://www.quantamagazine.org/why-mathematicians-still-cant-solve-the-collatz-conjecture-20200922/

Information technology – Syntactic metalanguage – Extended BNF. International Organization for Standardization in International Electrotechnical Commission, ISO/IEC 14977 (1996).

JanCLang projekt. *Repozitorij GitHub* (online). 2023. (citirano 26. 3. 2023). Dostopno na naslovu: https://github.com/JanCotar/JanCLang

JanCLang. *Paket NuGet* (online). 2023. (citirano 26. 3. 2023). Dostopno na naslovu: https://www.nuget.org/packages/JanCLang/

Karel The Robot. *Domača stran* (online). 2023. (citirano 28. 2. 2023). Dostopno na naslovu: https://www.cs.mtsu.edu/~untch/karel/

Kralj, A. *Izdelava lastnega programskega jezika s slovensko sintakso*. Diplomsko delo. Maribor. Fakulteta za elektrotehniko, računalništvo in informatiko, Univerza v Mariboru, 2020. Dostopno na naslovu: https://dk.um.si/IzpisGradiva.php?id=77172

Linotte. *Domača stran* (online). 2023. (citirano 28. 2. 2023). Dostopno na naslovu: http://langagelinotte.free.fr/wordpress/

Microsoft. *.NET Standard* (online). 2022. (citirano 26. 3. 2023). Dostopno na naslovu: https://learn.microsoft.com/en-us/dotnet/standard/net-standard?tabs=net-standard-2-0

Mogensen, T. *Introduction to Compiler Design*. Cham, Švica: Springer International Publishing AG, 2017.

Neha, T. *Top-Down Parsing in Compiler Design* (online). 2022. (citirano 11. 3. 2023). Dostopno na naslovu: https://binaryterms.com/top-down-parsing-in-compiler-design.html#AdvantagesofTop-DownParsing

Open Source Initiative. *The MIT License* (online). 2023. (citirano 26. 3. 2023). Dostopno na naslovu: https://opensource.org/license/mit/

Sebesta, R. W. *Concepts Of Programming Languages.* Harlow, Essex, Anglija: Pearson Education Limited, 2016.

Statista Inc. *Most used programming languages among developers worldwide as of 2022* (online). 2023. (citirano 15. 2. 2023). Dostopno na naslovu: https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/

Upson, M. *How Many Coding Languages Are There?* (online). 2022. (citirano 15. 2. 2023). Dostopno na naslovu: https://www.bestcolleges.com/bootcamps/guides/how-many-coding-languages-are-there/#:~:text=According%20to%20the%20Online%20Historical,in%20the%20commonly%20used%20group

Vidovič Muha, A. *Slovensko leksikalno pomenoslovje* (e-izdaja). Ljubljana. Znanstvena založba Filozofske fakultete Univerze v Ljubljani, 2021. Dostopno na naslovu: https://e-knjige.ff.uni-lj.si/

Weisstein, E. W. *Collatz Problem* (online). 2023. (citirano 24. 3. 2023). Dostopno na naslovu: https://mathworld.wolfram.com/CollatzProblem.html

Wexelblat, R. L. (ed.). *History of programming languages.* New York: Academic Press, 1981.

Wolf, C. E., in Oser, P. *The Shunting Yard Alghoritm* (online). 2023. (citirano 1. 3. 2023). Dostopno na naslovu: https://mathcenter.oxford.emory.edu/site/cs171/shuntingYardAlgorithm/

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Diplomant: **Jan Čotar**,

rojen: **8. 4. 1988** v kraju **Postojna**,

i z j a v l j a m,

da sem diplomsko delo z naslovom

RAZVOJ TOLMAČA PROGRAMSKEGA JEZIKA S SLOVENSKO SINTAKSO

izdelal in napisal samostojno.

Kranj, maj 2023 Podpis študenta:

ZAŠČITA DIPLOMSKEGA DELA

A. Copyright **c**: - diplomant ..... podpis:

Kopiranje in vsakršen drug način razmnoževanja v celoti ali posameznih delov nista dovoljena brez predhodnega pisnega dovoljenja nosilcev te pravice.

B. Glede na Zakon o avtorski in sorodnih pravicah (UL RS št. 21/95, 9/01, 30/01, 85/01, 43/04, 58/04, 94/04, 17/06, 44/06, 139/06, 16/07) in Zakon o industrijski lastnini (UL RS št. 13/92, 13/93, 27/93, 34/97, 75/97) velja še naslednje:

Diplomsko delo – arhivski izvod si je možno ogledati samo v prostorih knjižnice Šolskega centra Kranj, in sicer s pisnim dovoljenjem:

avtorja – diplomanta

diplomant ..... podpis avtorja – diplomanta:

Če ni avtorjevega – diplomantovega podpisa, je diplomsko delo v knjižnici Šolskega centra Kranj nedostopno za vpogled.

Pojasnilo k točki B:

Lastnoročni podpis avtorja – diplomanta dovoljuje ogled diplomskega dela le v knjižnici Šolskega centra Kranj.

Brez lastnoročnega podpisa avtorja – diplomanta ogled diplomskega dela ni možen.

Kranj, maj 2023