Fakulta matematiky, fyziky a informatiky, Univerzita Komenského
v Bratislave

Diplomová práca

Ondrej Krško

Bratislava 2005

Fakulta matematiky, fyziky a informatiky, Univerzita Komenského
v Bratislave

Robot Karol.NET

Návrh jazyka a implementácia

Autor: Ondrej Krško
Vedúci dipl. práce: RNDr. Marián Vittek, PhD.
Bratislava apríl 2005

Čestne prehlasujem, že diplomovú prácu som vypracoval samostatne s použitím literatúry a elektronických dokumentov uvedených v zozname použitej literatúry.

. . . . . .

Ondrej Krško

Ďakujem svojmu diplomovému vedúcemu RNDr. Mariánovi Vittekovi, PhD., za cenné rady a pripomienky pri vypracovávaní diplomovej práce.

Obsah

Obsah 5
Úvod 6
Cieľ 6
História 7
Karol na Slovensku 11
Systém .NET 14
Opis jazyka 18
Základné prvky jazyka 19
Premenné 20
Polia 21
Operátory 22
Riadenie behu kódu 24
Vetvenie 24
Cykly 26
Výnimky 27
Kontrolovaný a nekontrolovaný kód 28
Triedy 30
Vnorené triedy 32
Rozhrania 34
Metódy tried 35
Atribúty tried 37
Konštanty 39
Indexéry 40
Preťažovanie operátorov 42
Vymenovanie 43
Konverzie typov 45
Prvky jazyka pre podporu .NET 46

Záver k opisu jazyka 48
Kompilátor 49
Systémový design 49
Lexikálna analýza 50
Rozpoznávanie slov 51
Implementácia prechodových diagramov 52
Bufferovanie vstupu 53
Syntaktická analýza 55
Ošetrenie chýb 59
Správa typov a strom dedičností pre rozhrania a triedy 60
Generovanie prvkov jazyka 61
Optimalizácia 64
Záver 65
Referencie 66
Prílohy 69

Kapitola 1
Úvod

V dnešnej dobe sa práca s počítačom berie ako samozrejmá vec. Výučba sa postupne zavádza aj na základné školy. Na vyučovanie programovania detí základných škôl treba používať „špeciálne“ pedagogické postupy a „metodické pomôcky“. Tieto metodické pomôcky sa nazývajú malé programovacie jazyky (alebo detské) a sú špeciálne tým, že spôsob výučby programovania je prispôsobený deťom. Nevýhodou voľby takéhoto jazyka je, že v ďalších stupňoch výučby treba prechádzať na iný jazyk, na Slovensku je to zväčša Pascal. Žiak sa opäť musí naučiť, ako vyzerajú jazykové konštrukcie, ktoré sa už raz učil a až potom môže pokračovať v ďalšom vzdelávaní.
Autor jednej z verzií Karla a vedúci pri tvorbe Smalltalku Alan Kay z Palo Alto Research Center firmy Xerox v roku 1979 povedal, že detský programovací jazyk „by mal mať nízky prah a vysoký strop“. Teda dieťa by sa malo ľahko oboznámiť s prostredím a jazykom, naučiť sa v ňom pracovať, ale nemalo by príliš skoro naraziť na jeho hranice.

Cieľ

Robot Karol je programovací jazyk určený na výučbu programovania. V súčasnosti existuje niekoľko jeho verzií. Mojou snahou bude rozšíriť existujúce návrhy o premenné a podporu objektovo orientovaného programovania a napísať kompilátor pre tento rozšírený jazyk generujúci kód pre platformu Microsoft .NET. V tejto úvodnej kapitole bližšie opíšem, čo je cieľom mojej práce, čo to je Robot Karol, ako vznikol a čo je to platforma .NET.

Cieľom mojej práce je vytvorenie ľahko použiteľného jazyka s rozšíreniami pre pokročilejších používateľov a implementácia jeho kompilátora. Budem sa snažiť

o zachovanie jednoduchosti pôvodných jazykov určených na ovládanie Robota Karola a rozšírenie niektorých jeho prvkov na úroveň dnes používaných objektovo orientovaných jazykov. Jazyk by mal zabezpečiť, že žiak môže počas výučby ostať celý čas pri jednom jazyku a môže sa pomocou neho naučiť všetko od základov programovania až po zložitejšie techniky.

V existujúcich malých jazykoch sa žiaci učia ovládať viditeľný objekt, ktorým môže byť napríklad korytnačka alebo robot v mikrosvete (model miestnosti, pieskoviska, atď). Objekt dokáže vykonávať niekoľko jednoduchých príkazov a reagovať na základné príkazy vracajúce hodnotu[1]. Tento preverený princíp by som chcel zachovať, rozšírenia musia byť navrhnuté tak, aby ho neporušili.

Na svete sa nachádza množstvo detských programovacích jazykov, ale iba zopár z nich je použiteľných na Slovensku, pretože nie sú preložené do slovenčiny[11]. Históriu týchto jazykov podávam v nasledujúcej podkapitole.

História

V minulosti sa objavilo už niekoľko pokusov o vývoj špeciálnych jazykov uľahčujúcich prvé kroky pri vyučovaní dieťaťa. Medzi najznámejšie patrí korytnačia grafika (Korytnačka Logo, Papert, 1980). V polovici roku 1960 Seymour Papert, matematik, ktorý pracoval so Jeanom Piagetom v Ženeve, po príchode do USA spolu s Marvinom Minskym založili Laboratórium pre umelú inteligenciu na Masatchusetskom technologickom inštitúte (MIT). Papert pracujúci v tíme s Boltom, Beranekom a Newmanom, pod vedením Wallace Feurzeiga, v roku 1967 vytvoril prvú verziu Loga.

Do konca roku 1970 sa Logo rozšírilo na MIT a niekoľko ďalších výskumných miest: Edinburgh, Škótsko, Tasmániu a Austráliu. V miestnych školách sa viedli malé bádateľské aktivity. Dan Watt a iní výskumníci z MIT dokumentovali svoje práce niekoľkými žiakmi základných škôl, kde používali Logo. Ich výsledky boli v tom čase publikované v MIT.

[1] Definícia mini-jazykov podľa [1]

Rozmach zažilo Logo koncom 70-tych rokov spolu so vstupom prvých osobných počítačov na scénu. MIT Logo skupina vyvinula verziu Loga pre dva typy počítačov: Apple II a Texas Instruments TI 99/4. Samotný programovací jazyk Logo bol v obidvoch verziách totožný, rozdiely boli len vo využití.
Nové verzie Loga boli implementované vo viac ako dvanástich jazykoch na rozmanitých počítačoch. V severnej Amerike boli populárne Atari a Commodore Logo. V tom čase bol veľký záujem o Logo ako seriózny programovací jazyk, zvlášť na nových počítačoch Macintosh. Coral Software vyvinuli objektovo orientovanú verziu Loga, nazvanú Object Logo. Aj napriek tomu, že obsahovala kompilátor, nestala sa veľmi populárnou.
Pomerne veľký úspech malo Logo v Latinskej Amerike, kde sa každé dva roky v inej krajine konal Logo kongres. Ďalšie krajiny, kam zavítalo Logo, sú Japonsko, Anglicko, Španielsko a samozrejme Slovensko (Comenius Logo).

Programovací jazyk Logo, "nárečie" Lisp-u nebol navrhnutý špeciálne na výučbu programovania a na používanie korytnačej grafiky, ale podmnožina Loga sa ukázala ako veľmi vhodná[1]. Napriek úspechu korytnačej grafiky jej možno vytknúť jednu chybu: na rozdiel od iných malých jazykov je korytnačka akoby slepá a nedokáže odpovedať na požiadavky, kontrolovať stav svojho sveta.

Prvý a veľmi populárny malý jazyk bol „Karel the Robot" navrhnutý **Richardom Pattisom** ako úvod k vysokoškolským prednáškam o jazyku Pascal. Pattisov „Karel the Robot" sa „narodil" v roku 1981, teda je rovnako starý ako autor týchto riadkov. Bol opísaný v knihe s rovnakým názvom[3] . Pattis rovnako navrhol prvé programovacie prostredie pre Karola. Jazyk obsahoval všetky dôležité štruktúry z Pascalu na riadenie behu. Dal sa použiť na výučbu sekvenčného vykonávania programu, procedúr, podmienok a cyklov. Zložitosť programovania bola znížená vypustením premenných, typov alebo výrazov. Ovládaný objekt – robot Karel sa pohyboval po miestnosti, kde sa nachádzali steny, tehly a ďalšie objekty, Karel mohol vyberať tehly zo svojho vreca. Medzi základné príkazy, na ktoré vedel robot reagovať, patrili: krok, vľavo, vpravo, zober a polož. Celkom 18 predikátov

umožňovalo Karlovi (programátorovi) kontrolovať stav svojho mikrosveta (prítomnosť tehál v okolí, stien, smer, atď).

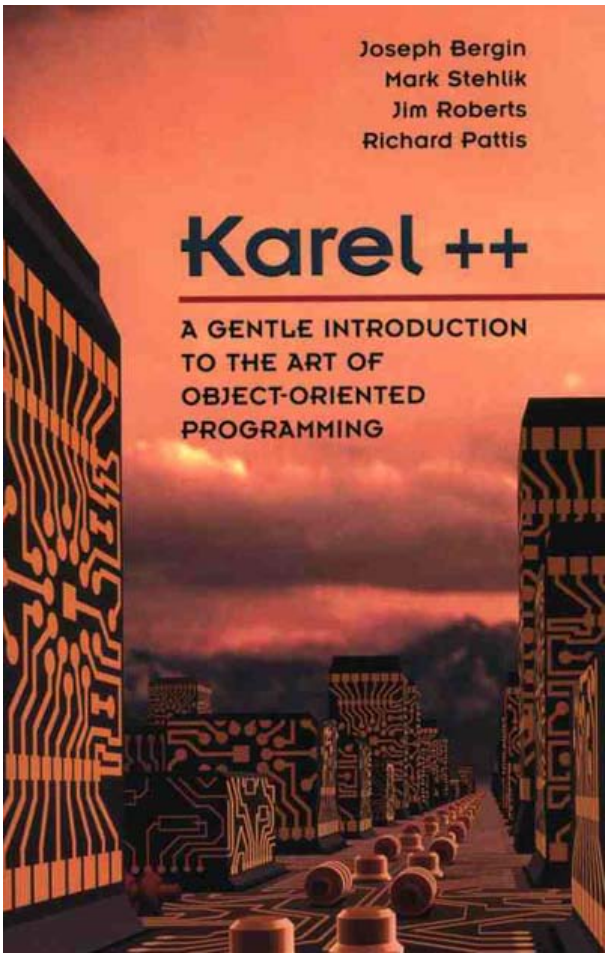
Robot dostal meno podľa známeho českého spisovateľa Karla Čapka. Ten ako prvý v scenári divadelnej hry R.U.R. pomenoval stroj podobajúci sa človeku. Túto divadelnú hru prvýkrát uviedli v pražskom Národnom divadle 25. januára 1921. Vtedy ešte určite netušil, že jeho nápad sa bude používať aj ako motivačný prvok a základy detského programovania sa budú vyučovať pomocou robota, ktorý bude niesť meno po svojom objaviteľovi.

Karel the Robot inšpiroval ďalších autorov k tvorbe ďalších malých jazykov, medzi priamo ovplyvnené patria:

- **Martino** (Olimpo et al., 1985) a **Marta** (Calabrese, 1989) v Taliansku. Marta je robot podobný Karolovi, základom ktorého je však Logo. Môže sa použiť ako ľahký úvod do programovania pre školákov i dospelých. Medzi vlastnosti rôzne od Karla patrí:
    - Marta môže byť riadená klávesmi, čo je vhodné pre menších žiakov
    - môže stavať svoj svet, vie postaviť a zrušiť stenu
    - môže byť riadená potme, keď nevidno prekážky a značky
    - ľahko sa dajú definovať nové príkazy a operácie, napísaním Logovských procedúr
    - programovať sa dá na niekoľkých úrovniach – jednoriadkové programy, viacriadkové programy a Logovské procedúry
    - Marta je ohraničená mriežkou rozmeru 7x15
    - na jeden priesečník môže položiť najviac 4 značky
- **Darel** (Kay & Tyler, 1993) v Austrálii
- **Jozef the Robot** (Tomek, 1982-1983)

Pôvodný Karel the Robot sa dnes používa na mnohých univerzitách a školách v USA. Nová edícia knihy Karel the Robot vyšla v roku 1995 (Pattis, Roberts, & Stehlik, 1995). Karel sa neudomácnil len medzi deťmi.

Napríklad pre paralelné programovanie je používaný projekt Robot Brothers (Olimpio, 1998), príkladom objektovo-orientovaného prostredia sú Playground (Fenton and Beck, 1989), Gravitas (Sellman, 1992) a KidSim (Smith a kol., 1994). Niektorí autori považujú objektovo-orientované malé jazyky za najlepšiu možnosť, ako vyučovať mladších žiakov základy programovania [2], [14].



Autori kníh „*Karel++ A Gentle Introduction to the Art of Object-Oriented Programming*“ vydanej v roku 1997 a „*Karel J. Robot, A Gentle Introduction to the Art of Object-Oriented Programming in Java*“ vytvorili niekoľko tried v jazyku Java, ktoré reprezentujú Karla, tehly, miestnosť atď. Knihy sú súčasťou prednášok na Pace University. Na adrese [15] možno nájsť aj celý kurz. Autori začínajú s vysvetlením jednoduchých funkcií, ako položenie tehly, v ďalších prednáškach objasňujú pojmy z OOP, ako napríklad polymorfizmus, ...

K tomuto projektu vytvoril Christoph Bockisch z Darmstadtskej technologickej univerzity preprocesor, ktorý prekladá jeho vlastné zjednodušenia priamo do jazyka Java[16]. Napríklad tento program:

```
1. class RightTurner extends karelTherobot.ur_Robot {
2.         public void turnRight() {
3.                 turnLeft();
```

```
4.                      turnLeft();
5.                      turnLeft();
6.          }
7.  }
8.  task {
9.          RightTurner karel = new RightTurner( 2, 2, infinity, North );
10.         karel.putBeeper();
11.         karel.move();
12.         karel.turnRight();
13.         karel.move();
14.         karel.turnLeft();
15.         karel.putBeeper();
16.         karel.move();
17. }
```

sa preloží pomerne priamočiaro[2] na program v jazyku Java, ktorý už možno skompilovať pomocou kompilátora Javac.

Karol na Slovensku

Karol sa na Slovensku objavil v polovici osemdesiatych rokov. Po návrate Jozefa Hvoreckého zo Spojených štátov amerických sa myšlienka využitia Karla dostala aj na Slovensko. Pod vedením Andreja Blaha naprogramoval prvú verziu v roku 1986 Marián Vittek. Bola určená pre počítače PMD 85-2.
Keďže išlo o jeden z prvých programovacích jazykov, ktorý bol v slovenskom jazyku a počítače PMD boli najrozšírenejšie osobné počítače na Slovensku, Karel 3D sa stal veľmi rozšírený. Vyučoval sa na niekoľkých základných školách aj na niektorých gymnáziách. V časopise „Zenit pionierov“ vychádzala jedna strana venovaná práve jemu. Boli tam publikované rôzne zaujímavé úlohy a pekné riešenia[11].

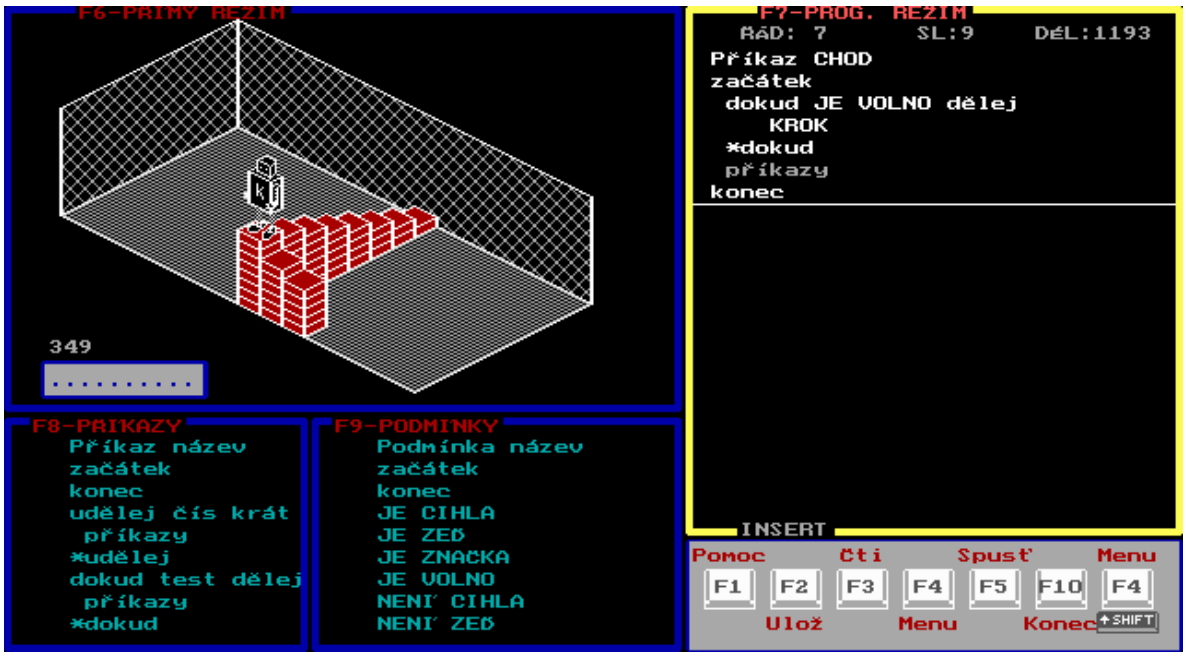
O Karolovi vyšlo niekoľko publikácií, prvou učebnicou bola kniha „Kamaráti robota Karla“ od Ľuby Gašparovičovej a Jozefa Hvoreckého[12], ktorá bola návodom na

[2] Preprocesor pridáva napríklad riadok „*import kareltherobot.*;*“ do každého kompilovaného programu.

používanie PC verzie Karla 3D z vtedajšej Katedry didaktiky informatiky z Matematicko fyzikálnej fakulty UK. Túto implementáciu na základnej škole používal aj autor tejto práce.

Karel 3D je upravený pre žiakov základných a stredných škôl. Pôvodného Karla prevyšuje niektorými vlastnosťami:

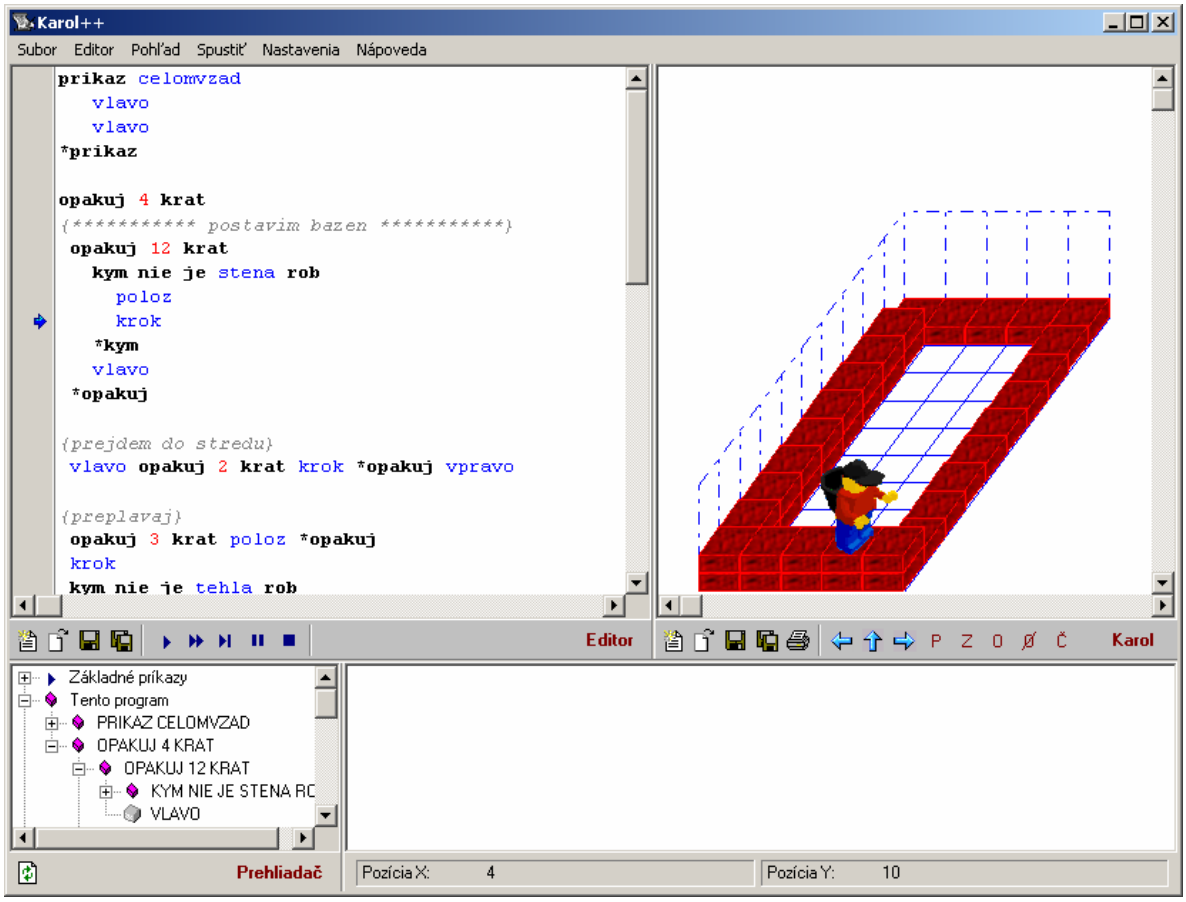
- robot sa pohybuje v trojrozmernom priestore zobrazenom v samostatnom okne. Ostatné okná sa využívajú na písanie programov, na menu a na kontextovo senzitívnu nápovedu.
- robot môže položiť tehlu, postaviť sa na ňu a hýbať sa hore a dolu – pohyb robota je trojdimenzionálny
- robot je riadený buď klávesmi (riadiaci, navigačný spôsob) alebo príkazmi (programovací spôsob)



Karel 3D bol dokonca preložený do niekoľkých jazykov. Vo verzii pre MS DOS mal syntaktický editor, ktorý zvýrazňoval slová v závislosti od syntaxe. Program v Karlovi sa dal krokovať.

Neskôr vznikla diplomová práca, obhajovaná v roku 1997, venovaná Karlovi 3D a v rámci nej aj verzia Karla pre Windows 3.11. Autorom tejto práce je Ľubomír Košút[17]. Jej cieľom bolo preskúmať možnosti tvorby detských programovacích jazykov vo Windows. Vybraným detským programovacím jazykom bol Karel 3D.

Potom nasledovalo dlhé obdobie, keď sa Karlovi nikto nevenoval. Na základných školách sa začalo presadzovať najmä Logo a na stredných bol pozorovateľný najmä prechod na vyššie programovacie jazyky (Pascal, C++). Jediná aktivita, ktorá vznikla v tom čase, bol môj ročníkový projekt z roku 2001, ktorý som nazval Robot Karol++. K projektu udržiavam aj internetovú stránku na adrese: www.robotkarol.sk. Jazyk bol pomerne jednoduchý, neobsahoval premenné, ale umožňoval vytváranie vlastných príkazov a podmienok (podmienka bol príkaz, ktorý dokázal vrátiť hodnotu: pravda alebo nepravda), rekurziu a poznal základné štruktúry na riadenie behu kódu.



Návštevnosť na stránke dodnes dokazuje záujem o robota, ako aj množstvo pozitívnych reakcií, ktoré som dostal. Stránku našiel aj Ulli Freiberger z Nemecka, ktorý na základe slovenskej verzie vytvoril rozšírený nemecký preklad [13]. Nová vetva programu je dodnes aktívne vyvíjaná a bola rozšírená napríklad o 2D náhľad na Karolovu miestnosť. Verzia je používaná na niekoľkých školách v Bavorsku.

Poslednou implementáciou bola diplomová práca Michala Zemana z roku 2004. Autor sa venoval najmä pedagogickému prínosu a zameral sa na využitie Karola pre najmenšie deti.

Systém .NET

Ako platformu, na ktorej bude bežať nový Robot Karol.NET a ním kompilované programy, som zvolil .NET. Je to nová platforma od spoločnosti Microsoft, ktorá by mala nahradiť dnes už 12 rokov starú platformu Win32. Základom systému je .NET Framework, behové prostredie, umožňujúce beh programov napísaných v niektorom z jazykov pre tento systém (obdoba Java Runtime Environment). Ďalšiu dôležitú časť platformy tvoria triedy, napríklad na prácu so sieťovými službami, grafickým prostredím atď. Triedy nahrádzajú volania funkcií, na ktoré boli zvyknutí programátori pre Windows.

Tvorba programových binárnych modulov vo Win32, ktoré by bolo možné používať v rôznych jazykoch bola skôr obtiažna. Knižnice DLL nešpecifikovali presné dátové typy a preto vznikali nekompatibility. Programátor napríklad nemohol jednoducho používať funkcie, ktoré boli napísané v jazyku Pascal (Delphi) a ako vstupný parameter vyžadovali reťazec z jazyka C++, ktorý používa odlišné kódovanie reťazcov. Ďalší problém platformy Win32 sa označuje ako „DLL Hell“. Vzniká, keď niektoré aplikácie vyžadujú nižšiu verziu knižnice a niektoré vyššiu, pričom rozhrania týchto verzií nie sú rovnaké. Systém .NET odstraňuje tieto problémy.

V systéme .NET je zdrojový kód ľubovoľného jazyka preložený do **MSIL** (Microsoft Intermediate Language, obdoba Java Byte Code). MSIL reprezentuje medzistav v procese konverzie zdrojového kódu do binárneho kódu. Je to akýsi pseudo-assembler, ktorý sa nachádza medzi zdrojovým kódom a strojovým kódom. Pri kompilovaní nejakého programu pre .NET kompilátor preloží zdrojový kód do MSIL, ktorý je nezávislý od sady inštrukcií používaných procesorom.
Po spustení programu je MSIL preložené do strojového kódu špecifického pre konkrétny proces použitím just-in-time (JIT) kompilátora. MSIL určuje základné dátové typy, určuje ich správanie a vďaka tomu sú všetky .NET jazyky navzájom kompatibilné a ľahko prepojiteľné.

Systém .NET spravuje existenciu objektov pomocou **Garbage Collection**. Keď program uvoľní poslednú referenciu na objekt, možno uvoľniť použitú pamäť. Neudeje sa tak hneď, ale garbage collector ho odstráni niekedy v budúcnosti. Vďaka tomuto prístupu dosahujú niektoré operácie vyšší výkon ako pri bežnom binárnom kóde. Potvrdzujú to aj nezávislé testy.
Systém .NET je rovnako schopný nájsť a rozpoznať aj cyklické referencie medzi objektmi a ak už na ne neexistuje iná referencia, môže ich uvoľniť.

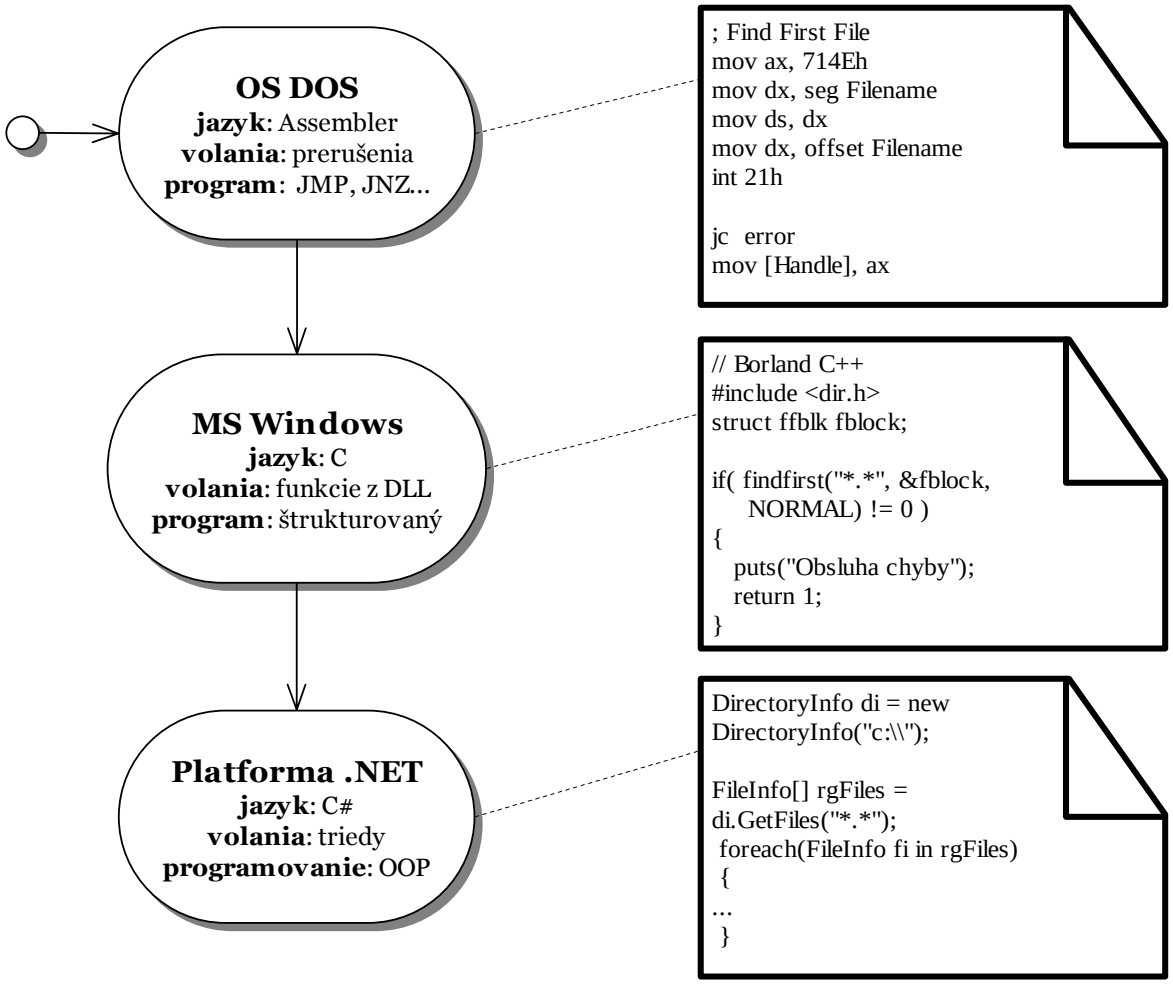
Na označenie súboru obsahujúceho MSIL kód sa používa označenie **assembly**. Môže to byť súbor s príponou .exe alebo .dll. Formát týchto súborov je však už odlišný, ako bol pri platforme Win32 a bez nainštalovaného .NET Frameworku súbor nemožno spustiť[27]. Assembly sú spravované a ukladané v assembly cache, ktorá umožňuje používanie viacerých verzií jedného súboru, dokonca súčasne z jedného programu.

Systém .NET je celou koncepciou veľmi podobný Jave. Na rozdiel od nej však Microsoft ponúka viacero jazykov[3], napríklad C#, Visual Basic .NET, Managed C++, z ktorých najviac preferuje C# (vyslovuje sa C Sharp). Zdá sa, že snahou

[3] Všetky jazyky musia spĺňať „Common Language Infrastructure“, všeobecne definované pravidlá, ako má jazyk pracovať so základnými dátovými typmi atď. Z ďalších jazykov od iných výrobcov možno spomenúť napríklad jazyk Pascal od Borlandu.

spoločnosti bude postupne „presvedčiť“ všetkých vývojárov pracujúcich na Win32, aby prešli na nový .NET, ktorý časom má nahradiť pôvodnú platformu tak, ako sa to stalo s MS-DOSom.

Nasledujúci diagram obsahuje zhrnutie vývoju platformy .NET. V kolónke „*jazyk*“ sa nachádza jazyk, ktorým možno najlepšie charakterizovať vývoj na danej platforme. Kolónka „*volania*“ opisuje, akým spôsobom platforma poskytovala programom svoje služby. Kolónka „*program*“ hovorí o spôsobe programovania, ktorý na binárnej úrovni umožňovala daná platforma. Neznamená to, že pre danú platformu sa nedalo programovať napríklad v jazyku podporujúcom OOP.



Microsoft prirovnáva prechod z programovania pre Win32 na platformu .NET prechodu z programovania pre DOS na Windows. Z veľkej časti je toto tvrdenie len

reklamou, aj keď .NET naozaj prináša veľa nových vlastností a rieši problémy, s ktorými sa často stretávali Windows programátori.

V auguste 2000 Microsoft spolu s ďalšími významnými spoločnosťami z oblasti informačných technológií (Hewlett-Packard, Intel, ...) zaslali Common Language Infrastructure (CLI) a špecifikáciu jazyka C# medzinárodnej štandardizačnej organizácii ECMA/ISO/IEC[18, 28]. Výsledkom bola prevzatie týchto špecifikácii ako štandardov[19].

Súčasne so štandardizáciou sa objavili aj ďalšie projekty súvisiace s platformou .NET. Dva z nich sa snažia o implementáciu s otvoreným zdrojovým kódom .NET Frameworku vrátane kompilátorov pre C# a Visual Basic.NET. Prvým z dvoch projektov je DotGNU Portable.NET[20], druhým a so širšou vývojárskou komunitou je projekt Mono[21]. Súčasťou Mona je aj vývoj vývojárskeho prostredia pod názvom „MonoDevelop“ a debugger.

Pri mojej implementácii som sa inšpiroval obidvoma projektami. Využil som niektoré ich časti – napríklad Jay z projektu Mono, ktorý slúži na spracovanie LR(1) gramatiky a generovanie syntaktického analyzátora rozpoznávajúceho túto gramatiku. Bližšie sa syntaktickej analýze venujem v samostatnej kapitole.

Kapitola 2
Opis jazyka

Programovací jazyk určený na výučbu programovania musí mať jednoduchú syntax. Ďalšou rovnako dôležitou vlastnosťou je „výrečnosť“ – používateľovi by malo byť jasné, čo ktorý príkaz robí. Vhodné je preferovať také štruktúry v jazyku, ktoré prezrádzajú svoju funkciu[4]. Jazyk by nemal byť citlivý na veľkosť písmen – slovo Trieda a TRIEDA by mal považovať za to isté. Pri návrhu rozšírení oproti pôvodnej verzii som dbal na spomenuté princípy.

Aká bola moja motivácia rozšíriť jazyk práve o nižšie opísané prvky?
Ako som už spomínal v úvode, detský programovací jazyk „by mal mať nízky prah a vysoký strop“. Dieťa by sa malo ľahko oboznámiť s prostredím a jazykom, naučiť sa v ňom pracovať, ale nemalo by príliš skoro naraziť na jeho hranice. Často náročným a zbytočným je prechod na iný programovací jazyk, keď si treba znova osvojovať novú syntax nového jazyka. Mojou motiváciou pre väčšinu rozšírení jazyka, ktoré som navrhol, bolo práve spomenuté tvrdenie.
Niektoré nové jazykové prvky som zaviedol kvôli tomu, aby sa študent alebo žiak hneď na začiatku vyhli bežným chybám a naučili sa používať odporúčané postupy. Napríklad kvôli správnemu ošetreniu chýb som do jazyka zaviedol výnimky.
Ďalšia časť nových jazykových prvkov bola vyžadovaná kvôli integrácii do prostredia .NET.
Pri niektorých prvkoch, ako napríklad indexéry a vlastnosti (oba sú popísané v samostatných podkapitolách), bolo mojou motíváciou na ich vloženie do Karolovho jazyka zvyšenie prehľadnosti programu.

[4] Napríklad príkaz for je v jazyku C pre použitie v malých jazykoch určených pre deti nevhodný. Forma, ktorá je používaná v jazyku Pascal je omnoho lepšia. Prezrádza, čo príkaz robí.

Na nasledujúcich stranách môže čitateľ nájsť opis jazyka od základných prvkov až po jazykové konštrukcie, ktoré umožňujú pracovať s triedami. Opis jednotlivých jazykových konštrukcií je zo svojej podstaty viac faktografickým vymenovávaním ako príjemným čítaním. Text som sa napriek tomu snažil rozšíriť o časti, ktoré čítanie uľahčia a čitateľa neunavia po prvých pár stranách.

Základné prvky jazyka

Na úvod sa budem venovať základným prvkom, ktoré jazyk obsahuje a primitívnym dátovým typom, ktoré možno používať. Neznamená to, že tieto typy musí malý programátor poznať, len považujem za správne najprv opísať tie.

1. Kľúčové slová (Keywords) sú slová, ktoré majú špeciálny význam v danom jazyku. Všetky slová sú rezervované jazykom a nemožno ich vo väčšine prípadov používať ako identifikátor. Medzi kľúčové slová patrí napríklad „opakuj“, „ak“ a „procedura“.
2. Literál je textová reprezentácia nejakej hodnoty určeného druhu. Medzi druhy literálov patrí napríklad integer, floating point, string, character, a date. Robot Karol.NET podporuje tieto literály:
    - Pravda a Nepravda sú literály typu Boolean. Typ Robot Karol označuje menom „Pravdivost“ alebo „bool“.
    - Celé čísla môžu byť v desiatkovej, šestnástkovej alebo osmičkovej sústave. Typ celočíselného literálu je určený podľa jeho hodnoty alebo podľa nasledujúceho znaku. Ak nie je žiadny určený, čísla v intervale <-2 147 483 648; 2 147 483 647> dostanú typ „CeleCislo“ (existuje aj alias “Cislo”, „Cislo32”); hodnoty mimo tohto rozsahu dostanú typ VelkeCislo (alebo VelkeCele, VelkeCeleCislo, Cislo64). Ak nestačí ani typ VelkeCeleCislo, je ohlásená chyba.
      Ďalej Robot Karol pozná MaleCislo (alias “Cislo16”, “MaleCeleCislo”) a najmenší číselný typ je MiniCislo („Cislo8“ alebo „MiniCeleCislo“).
    - Desatinné číslo je predvolene typu „VelkeDesatinneCislo“ (alias “VelkeDesatinne“, veľkosť 8 byteov). Ďalšie dátové typy sú

„DesatinneCislo“ (“Desatinne”, 4 byty) alebo „Decimalne“ (alias „MaxiDesatinneCislo“, „MaxiDesatinne“, 16 byteov)

- Literál typu „retazec“ je postupnosť nula alebo viacerých znakov v kódovaní Unicode začínajúcich s dvojitou úvodzovkou. V samotnom reťazci možno úvodzovky zapísať cez sekvenciu \”.
- Literál typu znak reprezentuje jeden znak v kódovaní Unicode. Jeho veľkosť je teda 16 bitov a dokáže bez problémov uchovať slovenské národné znaky.
- “Nic“ je špeciálny literál; nemá určený žiadny typ a je skonvertovateľný na ľubovoľný iný typ.

Premenné

Programátor často potrebuje uložiť nejaké hodnoty do pamäte počas behu programu. Premenné plnia práve túto funkciu. V jednoduchších implementáciách Robota Karola premenné neexistovali. Možnosť zapamätať si nejaký stav sa riešila napríklad rekurziou alebo počtom položených tehál v miestnosti. Nie vždy je takéto riešenie postačujúce a možnosť ukladať hodnotu aj inak výrazne rozšíri schopnosti jazyka.

Každá premenná má svoje meno a typ, ktorý určuje, aký druh dát bude v premennej uložený. Premennú je možné deklarovať nasledovne:

[typ] [názov] = [inicializačná hodnota]

pričom inicializačnú hodnotu možno vynechať. Premenné je možné deklarovať kdekoľvek v kóde. Robot Karol rieši viditeľnosť podobne, ako sú zvyknutí programátori z iných jazykov. Premenná deklarovaná v metóde bude viditeľná iba v nej, ak programátor zadeklaruje premennú vnútri nejakého bloku (napríklad vnútri vetvenia „ak“, „opakuj“, atď), mimo bloku nebude premenná viditeľná. Nasledujúci príklad sa nepodarí skompilovať, kompilátor ohlási chybu:

```
1. privatna staticka procedura Vstup ()
2.         ak pravda tak
3.            cislo Y1 = 5;
4.         *ak
5.         Vypis(Y1);
6. *procedura
```

Polia

Pole je dátová štruktúra, ktorá obsahuje niekoľko hodnôt prístupných pomocou indexu. Pole má rozmer(dimenziu) určujúcu počet indexov potrebných, aby sme sa dostali k prvku. Pre pole dĺžky N, možno používať index od 0 do N – 1 vrátane. Pole nachádza široké využitie – napríklad Karolova miestnosť môže byť reprezentovaná ako viacrozmerné pole a prístupná programátorovi na priame zásahy. Nasledujúci príklad vytvorí pole piatich celých čísiel:

```
cislo[] mojePole = nove cislo [5]
```

K prvkom možno pristupovať pomocou indexov mojePole[0] až mojePole[4]. Ďalšia možnosť, ako vytvoriť pole, je priamo zadať jeho hodnoty. V tomto prípade netreba zadávať rozmer poľa, je určený z počtu prvkov. Napríklad:

```
cislo[] mojePole = nove cislo[] {1, 3, 5, 7, 9}
```

Nasledujúci príklad vytvorí trojrozmerné pole s rozmermi 4 x 2 x 3:

```
cislo[,,] mojePole = nove cislo [4,2,3]
```

V prípade, že potrebujeme vytvoriť inicializované viacrozmerné pole, možno použiť zápis podľa nasledovných príkladov:

```
cislo[,] mojePole = nove cislo[,] {{1,2}, {3,4}, {5,6}, {7,8}}
cislo[,] mojePole = {{1,2}, {3,4}, {5,6}, {7,8}}
```

Operátory

Robot Karol poskytuje niekoľko operátorov. Jazyk ponúka predvolené funkcie pre väčšinu bežných aritmetických a logických operátorov, je ich však možné preťažiť a vytvoriť vlastnú obsluhu (bližšie v kapitole Preťažovanie operátorov). Nasledujúca tabuľka obsahuje zoznam podporovaných operátorov. Pri výbere operátorov podporovaných Karolom som hľadal čo najväčšiu množinu bežne používaných. Vždy je lepšie mať operátorov viac, ako ich funkciu doprogramovávať.

Aritmetické	+ - * / %
Logické	& | ^ ! ~ && || „azaroven“ „alebo“
Reťazce	+
Inkrementácia	++ --
Bitové posuny	<< >>
Relačné	== != < > <= >=
Priradenia	= += -= *= /= %= &= |= ^= <<= >>=
Index	[]
Pretypovanie	()
Adresovanie	* -> [] &

Pri práci s aritmetickými operátormi (+, -, *, /) môžeme dostať výsledky, ktoré sa nachádzajú mimo rozsahu zvoleného typu. Pomocou kľúčových slov „kontrolovany“ a „nekontrolovany“ možno určiť správanie v niektorých prípadoch. Bližšie sa im venujem v samostatnej kapitole.

Ak výraz obsahuje viacej operátorov, výpočet je založený na prioritách operátorov. V nasledujúcej tabuľke je sumarizácia priorít, priority klesajú z hora nadol:

Druh	**Operátory**
Základné	x.y f(x) a[x] x++ x-- novy

Unárne	+ - ! ~ ++x --x (T)x
Násobenie	* / %
Sčitovanie	+ -
Posun	<< >>
Relačné a typové testovanie	< > <= >= matyp jetypu ako
Rovnosť	== !=
Logické AND	&
Logický XOR	^
Logické OR	|
Podmienkový AND	&&
Podmienkový OR	||
Priradenia	= *= /= %= += -= <<= >>= &= ^= |=

Keď sa objavia dva operátory s rovnakou prioritou, ďalšie správanie závisí od asociativity:

1. Okrem operátorov priradenia, všetky binárne operátory sú ľavo-asociatívne, (operácie sú vykonávané zľava doprava). Napríklad výraz x + y + z sa vyhodnotí ako (x + y) + z.
2. Operátory priradenia sú pravo-asociatívne, operácie sú vykonávané sprava doľava. Napríklad výraz x = y = z sa vyhodnotí ako x = (y = z).
3. Priority a asociativitu možno meniť použitím zátvoriek.

Chovanie operátorov je podobné chovaniu operátorov z jazyka C#. Z operátorov, ktorých význam nemusí byť úplne jasný spomeniem tieto dva:

1. ~ operátor: počíta bitový komplement
2. ^ operátor: počíta funkciu XOR

Riadenie behu kódu

Riadenie behu kódu umožňuje počas behu programu rozhodnúť, ktorá časť kódu sa bude podmienečne vykonávať, prípadne opakovane vykonávať. Veľa štruktúr uvedených nižšie už existovalo aj v starších verziách. Dodržal som „hviezdičkovú" notáciu označujúcu koniec štruktúry z pôvodných verzií (napríklad vetvenie ak ukončíme slovom „*ak").

Vetvenie

Vetvenie nám umožňuje na základe vyhodnotenia podmienky vykonať rôzne časti kódu. Medzi podporované jazykové konštrukcie patrí:

1. Ak [podmienka] tak [kod] *ak
2. Ak [podmienka] tak [kod1] inak [kod2] *ak

Ďalšou možnosťou je obdoba príkazu switch z jazyka C. Tento druh vetvenia dokáže rozdeliť beh programu voľbou jednej premennej podľa viacerých hodnôt. Má nasledovnú formu:

ak pre [premenná] plati
    pripad [hodnota$_1$]: [kód$_1$]
    pripad [hodnota$_2$]: [kód$_2$]
    …
    pripad [hodnota$_n$]: [kód$_n$]
    inak [kód, vykoná sa, ak nenastal ani jeden z prípadov vyššie]
*ak

Kde význam jednotlivých položiek je nasledovný:

1. hodnota$_i$ je hodnota závislá od typu premennej v časti ak. Môže byť celočíselná alebo typu reťazec.
2. kód$_i$ je kód, ktorý sa vykoná, ak premenná nadobúda hodnotu rovnú hodnota$_i$. Beh programu nie je ukončený na konci prípadu sám, treba ho ukončiť slovom „prerus". V opačnom prípade bude vykonávanie programu pokračovať nie za najbližším *ak v ďalšom prípade. Výhodou takéhoto

prístupu je, že možno ho použiť tak, že jedna obsluha dokáže spracovať viacero rôznych prípadov. Napríklad:

```
1.  ak pre pripona plati
2.      pripad “JPG”:
3.      pripad “JPEG”:
4.          ... načítaj JPG zo súboru a zobraz ho...
5.          prerus;
6.      pripad „MP3“:
7.          ... načítaj MP3 a prehraj ju...
8.          prerus;
9.  *ak
```

3. Ak nebol nájdený ani jeden vhodný prípad a existuje voliteľná časť „inak“, obsluha sa odovzdá tejto časti.

Počet prípadov, ktoré programátor použije, nie je obmedzený. Nemôže ale nastať, že dva prípady testujú premennú na rovnakú hodnotu (kompilátor takúto chybu nájde a ohlási ju). Ak by sme takéto chovanie povolili, mohlo by to viesť k ťažko odhaliteľným chybám[5]. Jednou z odlišností od chovania príkazu switch v jazyku C je to, že ak v nejakom prípade programátor zadá nejaký kód, musí už použiť príkaz „prerus“ (respektíve „vrat“, aby sa zamedzilo voľnému prechodu z jedného prípadu do druhého). Napríklad nasledovný príklad skončí chybou:

```
1.  ak pre X plati
2.      pripad 0:
3.          Vypis("Hodnota X je 0, 1 alebo 2 (" + X + ")");
4.      pripad 1:
5.      pripad 2:
6.          Vypis ("Hodnota X je 0, 1 alebo 2 (" + X + ")");
7.          prerus;
8.      pripad 3:
```

[5] takéto chovanie je zakázané aj napríklad v jazyku Java a C# z rovnakých dôvodov.

```
 9.          Vypis ("Hodnota X: 3 (" + X + ")");
10.          prerus;
11.      inak
12.          Vypis ("Všeobecná obsluha. (" + X + ")");
13.          prerus;
14.  *ak
```

Ak by sme odstránili riadok číslo 3, program by sa podarilo skompilovať. Tento prístup teda kombinuje 2 výhody: umožní programátorovi jedným kódom obslúžiť viacero prípadov a zároveň upozorní programátora na možnú chybu – zabúdané kľúčové slovo break v jazyku C.
Z rovnakých dôvodov také isté správanie existuje aj v C#. Neumožňuje síce takú variablitu ako switch v jazyku C alebo Jave, vzhľadom na nie moc častú potrebu spomenutého správania som sa rozhodol pre „bezpečnejšiu“ aj keď „výrazovo slabšiu“ implementáciu.

Cykly

Cykly sú používané na viacnásobné opakovanie častí kódu. Medzi podporované konštrukcie patrí:

1. Kym [podmienka] rob [kod] *kym
2. Opakuj [pocet] krat [kod] *opakuj
3. Opakuj od [pociatok] do [koniec] [kod] *opakuj
4. Opakuj od [pociatok] do [koniec] pre [premenna] [kod] *opakuj
5. Opakuj pre [kazde| kazdy|kazdu] [typ] [premenna] spomedzi [pole alebo kolekcia] *opakuj

Jazyk definuje aj vyššie spomínané kľúčové slovo „prerus“. Slovo preruší vykonávanie najbližšieho cyklu alebo vetvenia, v ktorom sa nachádza. Beh pokračuje za riadkom, kde končí prerušený cyklus.
Ďalšie kľúčové slovo „pokracuj“ vynechá zvyšok najbližšieho cyklu, v ktorom sa nachádza a spustí ďalšiu iteráciu cyklu.

Bližšie sa opisu týchto štruktúr nebudem venovať, nakoľko sú obdobou podobných z väčšiny programovacích jazykov a ani ich správanie nie je odlišné.

Výnimky

Jazyk ponúka aj podporu na obsluhu výnimočných neočakávaných situácií, označovaných ako výnimky, ktoré sa môžu vyskytnúť počas vykonávania programu. Tieto udalosti sú obsluhované kódom, ktorý je za normálnych okolností vynechaný.

```
skus
  [rizikový kód]
zachyt ([deklarácia výnimky 1])
  [obsluha výnimky 1]
...
zachyt ([deklarácia výnimky n])
  [obsluha výnimky n]
nakoniec
  [ukončovací kód]
*skus
```

Kľúčové slovo „vyvolaj“ sa používa na označenie vzniku výnimky počas behu programu. Všeobecná forma príkazu je veľmi jednoduchá:

```
vyvolaj ([výnimka, trieda oddedená od System.Exception])
```

Nasledovný príkaz MalyMatematik dokáže sčitovať čísla do 100. Ak by súčet mal prekročiť 100, vyvolá výnimku:

```
1. prikaz MalyMatematik (cislo Scitanec1, cislo Scitanec2) : cislo
2.         ak Scitanec1+ Scitanec2>100 tak
3.          vyvolaj (nova
4.                 MatematickaVynimka("Neviem pocitat do viac ako 100"));
5.         *ak
6.         vrat Scitanec1+ Scitanec2;
7. *prikaz
```

Výnimky som sa rozhodol pridať do jazyka kvôli osvojeniu si dobrých programátorských techník.
Prečo je vhodné používať výnimky namiesto návratových hodnôt funkcií? V prvom rade výnimky poskytujú jednotnú formu na kontrolovanie chybových stavov. Programátor sa nemusí zaoberať tým, či príkaz vracia pravdivostné hodnoty, alebo chybný stav označuje návratová hodnota rovná -1, atď. Ďalšou výhodou je, že aj v prípade, že programátor „nezabalí“ kód do bloku skús, hlavná obsluha chybu zachytí a podá o nej správu. Pri návratových hodnotách však chybu možno veľmi ľahko ignorovať, čo môže spôsobiť zvláštne chovanie programu na úplne iných miestach, ako samotná chyba vzniká.

Kontrolovaný a nekontrolovaný kód

Kľúčové slovo „kontrolovany“ sa používa na riadenie kontrolovania pretečenia pri výpočtoch s celočíselnými premennými a operáciami. Motiváciou pre toto rozšírenie bola kompatibilita s inými jazykmi .NET. Existujú dve formy tohto príkazu:

```
  kontrolovany [kod] *kontrolovany
alebo
  kontrolovany ([vyraz])
```

kde význam je nasledovný:

1. [kod] je ľubovoľný programový kód, ktorého výrazy majú byť pri vyhodnocovaní kontrolované na pretečenie.

2. [vyraz] je výraz, ktorého hodnoty majú byť kontrolované na pretečenie. Výraz musí byť v zátvorkách.

Kontrolované na pretečenie znamená, že ak výraz pri výpočte vytvorí hodnotu, ktorá je mimo rozsahu výsledného typu, správanie môže byť nasledovné:

1. ak výraz obsahuje len konštantné hodnoty, vyvolá sa chyba počas kompilovania a program sa nepodarí skompilovať
2. inak sa hodnoty vypočítavajú počas behu a je vyvolaná výnimka.

Kontrolovanie pretečení je predvolene nastavené na „kontrolovany“. Kľúčové slovo „kontrolovany“ je preto potrebné len v prípade, že v nekontrolovanom bloku potrebujeme odrazu kontrolovaný.

Kľúčové slovo „nekontrolovany“ sa používa obdobne ako „kontrolovany“. Rovnako existujú dve formy tohto príkazu:

nekontrolovany [kod] *nekontrolovany

alebo

nekontrolovany ([vyraz])

Nasledovný príklad ukazuje, ako možno používať spomínané kľúčové slová:

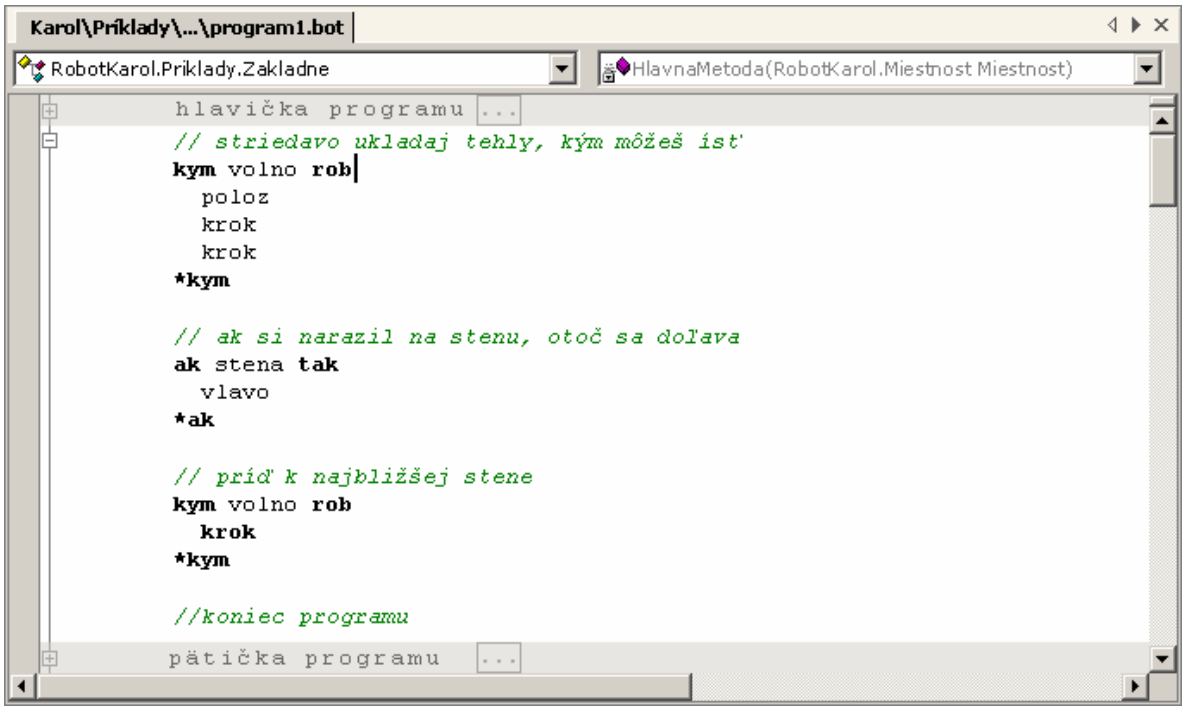
```
1.  staticke malecislo x = 32767;   // maximálna hodnota typu malecislo
2.  staticke malecislo y = 32767;
3.
4.  verejny staticky prikaz KontrolovanyVypocet () : cislo
5.      cislo z = 0;
6.      skus
7.          z =  kontrolovany((malecislo)(x + y));
8.      zachyt (System.OverflowException e)
9.          //bude vyvolaná výnimka, sem môže prísť jej obsluha
10.     *skus
11.     vrat z;
```

```
12. *prikaz
13.
14. verejny staticky prikaz NekontrolovanyVypocet() : cislo
15.     cislo z;
16.     z =  nekontrolovany((malecislo)(x + y));
17.     vrat z;
18. *prikaz
```

Ak zavoláme KontrolovanyVypocet(), výsledok tejto metódy bude číslo 0 a vyvolá sa výnimka. Ak zavoláme NekontrolovanyVypocet(), ako výsledok sa vráti -2.

Triedy

Ďalším rozšírením, ktoré som pridal do jazyka Robota Karola je rozšírenie o objektovo orientované programovanie. V jazykoch pracujúcich v systéme .NET platí zásada: *„nič, len trieda“*. Aby sa aj malý programátor mohol s Karolom hrať, bude potrebné vytvoriť špeciálny editor, ktorý umožní skryť niektoré časti programu. Môj návrh, ako by takýto editor mohol vyzerať, je na nasledovnom obrázku:



Malé tlačítko „plus“ v hornej a dolnej časti editora skrýva zložitejšie časti programu, ktoré deklarujú triedu, nejakú základnú metódu atď.

V tejto a niekoľkých nasledujúcich podkapitolách opíšem prvky jazyka podporujúce objektovo orientované programovanie. Najprv sa venujem opisu samotných tried, nasledovne rozhraniam a metódam, ďalej atribútom, idexérom a už spomenutému preťažovaniu operátorov.

Kľúčové slovo Trieda sa používa na deklarovanie triedy, jej vlastností a metód. Všeobecne zapísaná forma tohto príkazu vyzerá nasledovne:
[verejna | privatna | chranena | priatelska | interna | staticka]
[nova | prazdna | uzatvorena]
trieda [$MenoTriedy_1$]
    [potomok $MenoTriedy_2$]
    [implementuje Rozhrania]
    ... ďalšie deklarácie metód, atribútov, ...
 *trieda

Význam jednotlivých slov je viac-menej priamočiary a zhoduje sa s inými objektovo orientovanými jazykmi:

1. **„verejna“** je voliteľné slovo(nemusí byť zadané). Ak je trieda označená ako verejná, pristupovať k nej môže ľubovoľná iná trieda.
2. Ak je trieda deklarovaná ako „**interna“** (existuje alias „**priatelska**“), je prístupná všetkým triedam z assembly, v ktorom sa nachádza.
3. **„privatna“** je rovnako voliteľné slovo. Privátne môžu byť len vnorené triedy (bližší popis je v ďalšej podkapitole). Ak je trieda privátna, možno k nej pristupovať len v rámci triedy, do ktorej je vnorená.
4. K vnoreným triedam deklarovaným ako **„chranena“** možno pristupovať len z triedy, do ktorej je vnorená a z tried od nej oddedených.
5. Ak je trieda deklarovaná ako „**prazdna**“, znamená to, že nemožno vytvoriť jej inštanciu, pretože nemá dodefinované všetky metódy.

6. **Uzatvorená** trieda je trieda, od ktorej už nemožno dediť. Modifikátor „**nova**“ umožňuje prekryť členov triedy, od ktorej dedíme. Význam má len pri vnorených triedach. Príklad sa nachádza v ďalšej podkapitole.
7. MenoTriedy$_1$ je meno, pod ktorým bude trieda vystupovať.
8. Voliteľné slovo „**potomok**“ indikuje, že trieda dedí on inej triedy.
9. MenoTriedy$_2$ je meno predka vytváranej triedy.
10. Ak programátor zadá slovo „implementuje“, určí, že trieda implementuje metódy rozhrania. Trieda musí nutne implementovať všetky metódy rozhrania.
11. Rozhrania je zoznam rozhraní, ktoré trieda implementuje. Rozhrania sú oddelené čiarkou.

Poznámka:
Triedy, ktoré nemajú určený prístupový modifikátor(verejna a pod.) sú predvolene deklarované ako „priatelska“.

Vnorené triedy

Niekedy môže byť užitočnou vlastnosťou vložiť jednu triedu do vnútra inej triedy, napríklad ak ide o nejakú pomocnú triedu, ktorá sa používa výhradne v nadradenej triede. Zoberme si napríklad dvojicu tried:

```
1. Verejna trieda Parser
2.   Token[] tokeny;
3. *trieda
4.
5. Verejna trieda Token
6.    retazec Nazov;
7. *trieda
```

V tomto príklade sú obe triedy verejne dostupné, čo nie je ideálne riešenie.
S triedou Token totižto bude pracovať len trieda Parser. Je preto vhodné urobiť z nej vnorenú triedu, ktorú možno deklarovať ako privátnu. Tým ju skryjeme pred

všetkými ostatnými triedami s výnimkou triedy Parser. Triedu Token by sme síce mohli deklarovať ako internú, malo by to však dve nevýhody:

1. mohli by k nej pristupovať aj iné triedy z assembly
2. ak bude trieda vnorená, bude každému na prvý pohľad jasné, kam trieda patrí a aká trieda s ňou pracuje.

```
1. Verejna trieda Parser
2.   Token[] tokeny;
3.   Verejna trieda Token
4.       retazec Nazov;
5.   *trieda
6. *trieda
```

Triedu Token teraz žiadna iná trieda nevidí. Pracovať s ňou môže len trieda Parser. Do tried možno vnorovať aj rozhrania. Kompilátor dokáže kontrolovať správnosť prístupových modifikátorov. Ak sa napríklad programátor pokúsi spraviť chránenú inštanciu privátnej triedy, kompilátor ohlási chybu:

```
1. verejna trieda Trieda1
2.     privatna trieda Trieda2
3.         ...
4.     *trieda
5.     chranena Trieda2 inst = nova Trieda2 ();
6. *trieda
```

V predchádzajúcej kapitole som spomenul kľúčové slovo „nova“, ktoré umožňuje prekryť predchádzajúcu definíciu triedy a deklarovať ju nanovo. Ako toto kľúčové slovo funguje, možno ľahko a rýchlo pochopiť z nasledovného príkladu:

```
1.  verejna trieda ZakladnaTrieda
2.      verejna trieda Trieda1
3.          verejne cislo x = 200;
4.          verejne cislo y;
5.      *trieda
6.  *trieda
7.
8.  verejna trieda OdvodenaTrieda : ZakladnaTrieda
9.      nova verejna trieda Trieda1 // trieda skrýva predchádzajúcu deklaráciu
10.         verejne cislo x = 100;
11.         verejne cislo y;
12.         verejne cislo z;
13.     *trieda
14.
15.     verejna staticka procedura Vstup()
16.         // Vytvor inštanciu triedy deklarovanej na riadku 9.
17.         Trieda1 S1  = nova Trieda1();
18.         // Vytvor inštanciu triedy deklarovanej na riadku 2.
19.         ZakladnaTrieda.Trieda1 S2 = nova ZakladnaTrieda.Trieda1();
20
21.         Vypis(S1.x);
22.         Vypis(S2.x);
23.     *procedura
24. *trieda
```

Príklad vypíše číslo 100 a do druhého riadku 200. Ak by sme zmazali riadok číslo 10, kompilátor by na riadku 21 ohlásil chybu.

Rozhrania

Rozhranie špecifikuje, aké metódy musí trieda, ktorá ho implementuje, podporovať. Sú vo veľmi úzkom vzťahu s abstraktnými triedami. Pripomínajú abstraktnú triedu, ktorej všetky členy sú abstraktné. Pri práci s rozhraniami môžeme použiť dve funkcie:

1. Zistiť, či trieda implementuje dané rozhranie: možno tak urobiť pomocou kľúčového slova „matyp“ (alias „jetypu“).
2. Požiadať o rozhranie pomocou slova „ako“.

Rozhrania môžu opisovať metódy, atribúty, udalosti a indexéry.
Deklarácia rozhrania vyzerá nasledovne:

[nove | verejne | chranene | interne | privatne] rozhranie [meno] [: predkovia]
... ďalšie deklarácie metód, atribútov, ...
*rozhranie

Predkovia môže byť zoznam rozhraní, oddelených čiarkou, od ktorých rozhranie dedí. Všetky modifikátory majú rovnakú funkčnosť ako pri triedach.

Metódy tried

Metódy obsahujú vykonateľný kód. Robot Karol pozná dva základné druhy metód:

1. ktoré vracajú hodnotu, nazýva ich „prikaz“ (existuje alias „funkcia“)
2. ktoré len vykonávajú kód a nevracajú hodnotu, nazývajú sa „procedura“.

a dve špeciálne metódy, ktoré vyžaduje systém .NET:

3. konštruktor
4. deštruktor

Všeobecný formát vyzerá nasledovne:
[prístupové modifikátory]
[staticka | virtualna | prepisujuca | prazdna | nova]
[procedura | prikaz | konstruktor | desktruktor] [Meno metódy] ([parametre])
[: typ návratovej hodnoty v prípade, že deklarujeme príkaz]
... telo metódy ...
[*procedura | *prikaz | *konstruktor | *desktruktor]

Ďalej môžu existovať metódy s parametrami alebo bez. Ku každej metóde možno určiť niektoré vlastnosti: Karol pozná metódy:

1. **Statické** a nestatické. K statickej metóde možno pristupovať priamo, bez vytvorenia inštancie triedy. K nestatickým metódam, nazývaným aj inštančné, možno pristupovať len cez vytvorenú inštanciu.
2. Metódy môžu byť **preťažené**, čo znamená, že viacero metód môže mať rovnaké meno. Musia však mať odlišné parametre alebo ich počet. Výsledná hodnota funkcie sa nepovažuje za rozlišujúcu vlastnosť. Pri volaní metódy sa hľadá najvhodnejšia z preťažených metód podľa parametrov smerom od najľavejšie definovaného.
3. **Virtuálne** metódy sú metódy, ktoré možno prepísať pri dedení. Keď je virtuálna metóda zavolaná počas behu programu, behové prostredie .NET pohľadá najnižšie definovanú metódu v „strome potomkov“ a obsluhu odovzdá jej[6]. Predvolene je metóda nevirtuálna. Nevirtuálne metódy nie je možné prepísať. Modifikátor „virtualna“ nemožno používať s nasledovnými modifikátormi: „staticka”, „prazdna“ a „prepisujuca“.
4. Kľúčové slovo „**prepisujuci**“ (alebo „prepisujuca“) je modifikátor, ktorým možno označiť zdedenú metódu pri jej novej implementácii. Prepisujúca metóda musí mať rovnaký názov a vstupné parametre ako prepisovaná metóda. Prepisovať sa nedajú nevirtuálne a statické metódy. Prepisovaná metóda musí byť deklarovaná ako „virtualna“, „prazdna“, alebo „prepisujuca“.

   Pri prepisovaní nemožno zmeniť prístupnosť metódy (napríklad zmena „verejna” na „privatna“ nie je možná).
5. **Prázdna** metóda[7] je metóda, ktorá má deklarovaný len názov a parametre, ale žiadne telo. Inštancie nemožno vytvárať z tried, v ktorých sa nachádzajú prázdne metódy.
6. Špeciálne metódy sú **konštruktory**, ktoré umožňujú vytvorenie novej inštancie. Konštruktory nemôžu byť volané priamo programom. Zavolá ich systém .NET na žiadosť cez kľúčové slovo „novy“. Konštruktor deklarujeme pomocou slova „konstruktor“. Meno konšturktoru sa musí zhodovať s názvom triedy.

[6] Virtuálne metódy sa správajú rovnako ako v iných jazykoch, preto ich bližšie neopisujem.
[7] V iných jazykoch sa pre rovnakú funkciu používa označenie abstract.

7. Ďalším zvláštnym typom metód sú **deštruktory**. Rovnako nemôžu byť volané priamo. Ďalej ich nemožno preťažovať, teda jedna trieda môže mať maximálne jeden deštruktor. Deštruktory nemožno dediť, teda trieda nemá žiadny deštruktor, dokým ho programátor nezadeklaruje. Kedy bude vyvolaný deštruktor, to určuje systém .NET a jeho garbage collector. Po vyvolaní deštruktora systém spustí reťazové volania deštruktorov predkov. Meno deštruktora sa musí zhodovať s názvom triedy.
8. Metódam možno určiť ich **prístupové** modifikátory: „verejna“, „privatna“, „chranena“ a „interna“ (respektíve „priatelska“).

Metódy, atribúty a ďalšie prvky, nachádzajúce sa v triedach, definované ako privátne sú viditeľné len vnútri triedy. Metódy deklarované ako verejné sú viditeľné vnútri triedy, rovnako aj mimo nej. Metódy, ktoré nemajú určenú viditeľnosť, sú predvolene interné (toto neplatí pre atribúty, ktoré sú predvolene privátne). Chránené metódy sú viditeľné len vnútri triedy a pre potomkov triedy.
Na prístup k členom materskej triedy je možné použiť kľúčové slovo „predok“. Slovo umožňuje volať metódu materskej triedy, ktorá bola prepísaná inou metódou alebo určiť, ktorý konštruktor materskej triedy má byť použitý pri vytváraní inštancií odvodenej triedy.

Atribúty tried

Triedy môžu udržiavať a poskytovať svoje dáta dvoma spôsobmi:

1. členské premenné[8] udržiavajú dáta, ktoré trieda potrebuje počas svojej existencie. Členským premenným určujeme prístupovú úroveň nasledovanú typom a menom premennej. Premenné môžu byť označené ako verejné, privátne, chránené, interné alebo chránené interné.
2. vlastnosti[9] - sú metódy triedy, ku ktorým sa dá pristupovať akoby boli členskými premennými. V jazykoch, kde nie je možné vlastnosti definovať podobným spôsobom sa zvyknú používať metódy getProperty a setProperty. Riešenie, ktoré som zvolil v jazyku Karola považujem za prehľadnejšie, čo

[8] V systéme .NET označované ako field
[9] V systéme .NET označované ako property

bolo hlavným motívom, prečo som ich do jazyka pridal. Toto riešenie dokáže skryť implementačné detaily triedy a umožní pracovať s vlastnosťami triedy priamočiarejšie. Vlastnosť si možno napriek inému zápisu predstavovať ako dvojicu metód. Pre vlastnosť platí všetko čo pre metódy – môžu byť virtuálne, prepisujúce atď. V skutočnosti sú aj ako dvojica metód generované do binárneho súboru[10].

Všeobecný formát príkazu „vlastnost“ je nasledovný:

[pristupové modifikátory] vlastnost [názov] : [typ]
  citaj [kod_1] *citaj
  zapis [kod_2] *zapis
*vlastnost

Časti „citaj“ a „zapis“ nie sú povinné a možno niektorú z nich vynechať. Programový kód, nachádzajúci sa v časti „zapis“ (kod_2), má deklarovanú špeciálnu premennú s názvom „hodnota“, ktorá obsahuje hodnotu na zapísanie. Nasledovný príklad ukazuje, ako možno v jazyku Robota Karola používať vlastnosti a členské premenné:

```
1.  trieda Cas
2.    privatne velkedesatinnecislo Sekundy;
3.    verejne velkedesatinnecislo Posun;
4.
5.    verejna vlastnost Hodiny : velkedesatinnecislo
6.      citaj
7.        Vypis("Citam hodnotu vlastnosti Hodiny");
8.        vrat (Sekundy / 3600) + Posun;
9.      *citaj
10.     zapis
11.       Vypis ("Zapisujem hodnotu vlastnosti Hodiny");
```

[10] K pomenovaniu vlastnosti sa pridá „get_“ alebo „set_”. Vyplýva z toho aj obmedzenie, kompilátor nepovolí duplicitné vytvorenie metódy s rovnakou signatúrou(rovnaké meno a rovnaké vstupné parametre).

```
12.          Sekundy = hodnota * 3600;
13.       *zapis
14.    *vlastnost
15. *trieda
16.
17. verejna trieda VlastnostTest
18.     privatna staticka procedura Vstup ()
19.          Cas mojCas = novy Cas();
20.          mojCas.Hodiny = 16;
21.          mojCas.Posun = 1;
22.          Vypis("Cas v hodinach: " + mojCas.Hodiny);
23.      *procedura
24. *trieda
```

Pri atribútoch tried možno použiť kľúčové slovo „zamknuty“ (alebo „zamknuta“). Tento modifikátor znemožní zapisovanie do atribútu mimo konštruktoru a inicializačného priradenia.

Konštanty

Kľúčové slovo „konstanta“ možno použiť pri atribútoch a lokálnych premenných. Podobne ako slovo „zamknuty“ znemožní zmeny premennej. Možné použitie:

[pristupové modifikátory] konstanta [typ] [meno] = [hodnota]

Hodnota konštanty na rozdiel od atribútov deklarovaných ako „zamknuty“ musí byť určiteľná v čase kompilovania. Konštantu môžeme deklarovať aj takto:

```
verejna konstanta desatinne Pi = 3.14;
verejna konstanta desatinne Pi2 = 2 * Pi;
```

V skompilovanom kóde už bude hodnota Pi2 zapísaná priamo, nie ako výraz. Ďalší rozdiel medzi konštantou a zamknutým atribútom je, že hodnotu konštanty nemožno upravovať v konštruktore. Cyklické deklarácie hodnôt konštánt nie sú prípustné a kompilátor ich dokáže odhaliť.

Indexéry

Niekedy môže byť vhodné mať možnosť indexovať nejaký objekt rovnako, ako by to bolo pole. Dá sa to dosiahnuť tak, že pre daný objekt napíšeme takzvaný indexér, ktorý sprostredkuje rovnakú funkčnosť ako prístup do poľa. Podobne ako vlastnosť vyzerá ako jednoduchá premenná, aj keď jej čítanie a modifikácia prebieha cez prístupové funkcie, indexér vyzerá ako pole, ale indexovacie operácie prebiehajú cez prístupovú funkciu. Časté využitie nachádza napríklad pri práci s databázou, keď jeden index môže reprezentovať jeden riadok tabuľky. Ďalšie využitie môže byť napríklad štruktúra udržiavajúca zoznam robotov s možnosťou prístupu cez index:
Roboti[“Karol”].Krok();
Roboti[“Peter”].Vlavo();

Motívom na pridanie indexérov do jazyka bolo sprehľadnenie programov. Určite možno považovať za jednoduchšie vyššie uvedené dva riadky ako nasledujúce:
Roboti.VratRobotaPodlaMena(“Karol”).Krok();
Roboti.VratRobotaPodlaMena (“Peter”).Vlavo();

Všeobecná forma príkazu vyzerá nasledovne:
[novy | virtualny | zatvoreny | prepisujuci | prazdny]
[verejny | privatny | chraneny | interny]
indexer **[** [typ] [meno] **]** : [vyslednytyp]
citaj [kod_1] *citaj
zapis [kod_2] *zapis
*indexer

Nasledujúci príklad vytvorí triedu na prácu so súborom, s ktorým bude možné pracovať akoby to bol jeden veľký reťazec (aj keď v skutočnosti celý súbor v pamäti samozrejme udržiavať nebudeme). Príklad ukazuje aj použitie vlastnosti.

```
1.  verejna trieda VelkySubor
2.    Stream stream;
3.    verejny konstruktor VelkySubor(retazec MenoSuboru)
4.      stream = novy FileStream(MenoSuboru, FileMode.Open);
5.    *konstruktor
6.
7.    verejna procedura Zatvor()
8.      stream.Close();
9.      stream = nic;
10.   *procedura
11.
12.   // Indexér
13.   verejny indexer [velkecislo index] : minicislo
14.     citaj
15.       minicislo[] buffer = nove minicislo[1];
16.       stream.Seek(index, SeekOrigin.Begin);
17.       stream.Read(buffer, 0, 1);
18.       vrat buffer[0];
19.     *citaj
20.     zapis
21.       minicislo[] buffer = nove minicislo[1] {hodnota};
22.       stream.Seek(index, SeekOrigin.Begin);
23.       stream.Write(buffer, 0, 1);
24.     *zapis
25.   *indexer
26.
27.   verejna vlastnost Dlzka : velkecislo
28.     citaj
29.       vrat stream.Seek(0, SeekOrigin.End);
30.     *citaj
31.   *vlastnost
32.
33. *trieda
```

S takto vytvorenou triedou možno pracovať nasledovne:

```
1.  VelkySubor subor = novy VelkySubor("C:\\test.txt");
2.  opakuj pre I od 0 do subor. Dlzka-1
3.      Vypis( subor[I] );
4.  *opakuj
5.  subor.Zatvor();
```

Pre indexéry platí to isté, čo pre vlastnosti. V skutočnosti sú do binárneho súboru prekladané ako dvojica metód get_Item() a set_Item().

Preťažovanie operátorov

Preťažovanie operátorov umožňuje programátorovi vytvoriť vlastnú implementáciu pre operácie, kde jeden alebo obidva operandy sú užívateľom definované typy. Typickým príkladom na ukážku preťažovania operátorov je trieda pracujúca s komplexnými číslami alebo vektormi. Preťažený môže byť napríklad vektor určujúci vzdialenosť od ľavého dolného rohu Karlovej miestnosti a časť programu, ktorý by realizoval „naháňačku“ po miestnosti dvoch robotov môže vyzerať takto:

```
1.  // zisti rozdiel medzi vektormi
2.  Vektor v1 = Roboti["Karol"].Pozicia - Roboti["Peter"].Pozicia;
3.  // posuň sa smerom k Petrovi o jeden krok
4.  Roboti["Karol"].Pozicia += v1 / v1.Velkost;
```

Všeobecná forma preťažovania vyzerá nasledovne:

```
verejny staticky operator [ operator ]
    ( [ typ1 ] [ premenna1 ] [, typ2 ] [ premenna2 ]) : [typ3]
  ... implementácia...
*operator
```

kde typ_2 a $premenna_2$ sú nepovinné údaje, ktoré je nutné zadať len ak je operátor binárny. Preťažovať možno tieto operátory:

1. unárne: + - ! ~ ++ --

2. binárne: + - * / % & | ^ << >> == != > < >= <=

Kompilátor pri práci zabezpečí, že operátory, ktoré sú logicky súvisiace (!= a ==, > a <, >= a <=), musia byť definované naraz. Ak užívateľ nezadefinuje obidva operátory zo skupiny, ohlási chybu. Nasledujúci príklad ukazuje, ako možno vytvoriť triedu na prácu s komplexnými číslami v jazyku Robota Karola:

```
1.  trieda KomplexneCislo
2.      verejne cislo real;
3.      verejne cislo imag;
4.      verejny konstruktor KomplexneCislo(cislo real, cislo imag)
5.              ja.real = real;
6.              ja.imag = imag;
7.      *konstruktor
8.      verejny staticky operator + (KomplexneCislo c1, KomplexneCislo c2)
9.                                                      : KomplexneCislo
10.         vrat nove KomplexneCislo(c1.real + c2.real, c1.imag + c2.imag);
11.     *operator
12. *trieda
13.
14.     ...
15.     KomplexneCislo num1 = nove KomplexneCislo(2,3);
16.     KomplexneCislo num2 = nove KomplexneCislo(3,4);
17.     KomplexneCislo suma = num1 + num2; // hodnota (5, 7)
19.     ...
```

Vymenovanie

Vymenovanie je špeciálna forma hodnotového typu, ktorý je potomkom System.Enum a poskytuje alternatívne pomenovania pre hodnoty nižšie ležiaceho primitívneho typu(napríklad „cislo“). Typ vymenovanie má určený názov, nižšie ležiaci primitívny typ a množinu hodnôt a ich pomenovaní. Nižšie ležiaci typ musí byť jeden zo základných typov, napríklad „malecislo“, „cislo32“, alebo „cislo64“).

Všeobecná forma vymenovania vyzerá takto:

[verejne | chranene | priatelske| chranene priatelske | privatne] vymenovanie [meno] typu [nizsie leziaci typ]
  $menoclena_1$ [= inicializacnyvyraz$_1$],
  $menoclena_2$ [= inicializacnyvyraz$_2$],
  ...
  $menoclena_n$ [= inicializacnyvyraz$_n$]
*vymenovanie

Ako príklad možno uviesť:
verejne vymenovanie DniTyzdna typu cislo
  Pondelok = 1, Utorok, Streda, Stvrtok, Piatok, Sobota, Nedela
*vymenovanie

Voliteľná hodnota inicializačný výraz je hodnota, ktorú bude mať daný člen vymenovania. Môže to byť literál, konštanta ktorá už bola definovaná, iný člen vymenovania alebo ich kombinácia použitím logických a aritmetických operátorov. Nie je možné používať na inicializáciu premenné alebo volania funkcií. V prípade, že inicializacnyvyraz$_i$ nie je určený, priradená hodnota je nula (ak je to prvý člen $menoclena_1$), alebo väčšia o jedna ako hodnota bezprostredne predchádzajúceho člena.

Vymenovanie sa môže nachádzať len v deklarácii menného priestoru. Nemožno ich deklarovať vnútri metódy. Typické využitie môžu nachádzať napríklad pri deklarovaní smerov, v ktorých môže byť robot natočený:

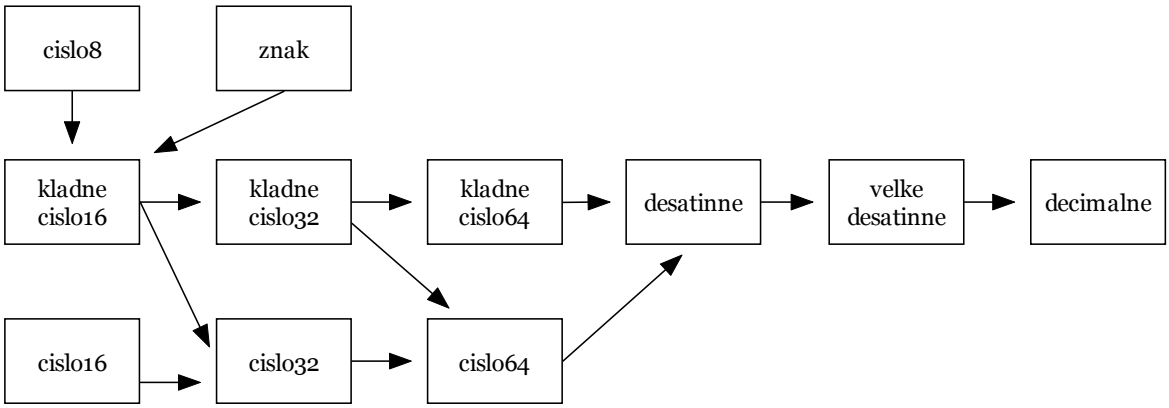
```
1. verejne vymenovanie Smer typu cislo
2.     Zapad, Sever, Vychod, Juh
3. *vymenovanie
```

Konverzie typov

V niektorých prípadoch potrebujeme previesť jeden dátový typ na iný. Typová konverzia môže byť buď implicitná alebo explicitná. Ak je jeden dátový typ konvertovaný na iný automaticky kompilátorom, hovoríme o implicitnej. Všetky automaticky robené zmeny prebehnú len v tom prípade, ak sa nestratia žiadne dáta (napríklad konverziou 16 bitového čísla na 8 bitové).

Konverzia prebehne napríklad pri vyhodnocovaní výrazov, priradeniach, volaniach metód s inými typmi parametrov atď. Podporované implicitné konverzie možno rozdeliť na:

1. **Numerické** konverzie: na obrázku sú uvedené všetky podporované konverzie pre numerické dátové typy. Smer šípky na obrázku ukazuje možnosť konverzie.



2. **Zaobaľovacie** konverzie dokážu previesť hodnotu na objekt. Napríklad zaobalenie číselnej hodnoty pozostáva z vytvorenia novej inštancie a naplnenie jej hodnoty na hodnotu pôvodnú. Lepšie možno zaobaľovanie pochopiť z príkladu:

```
1.  cislo x = 10;
2.  objekt o = x;        //Zaobalenie, vytvorí sa inštancia triedy System.Int32
3.  ak o matyp cislo tak
4.      Vypis("o obsahuje typ cislo");
    *ak
```

Explicitné konverzie sú vynútené programátorom použitím operátora ().

V nasledujúcom príklade pretypujeme desatinne číslo na celé číslo:

```
cislo x = (cislo) 26.45;
```

Ďalším typom konverzie je konverzia na reťazec. Trieda „objekt“, od ktorej v systéme .NET dedia všetky ostatné triedy implementuje metódu ToString(), ktorá vracia reťazcovú hodnotu. Pri konverzii na reťazec sa vyvolá táto metóda. Ak napríklad napíšeme:

```
retazec vystup = "Skuska:" + 12;
```

vykoná sa postupne zaobalenie hodnoty 12 do triedy System.Int32 a následné použitie jej metódy .ToString(), ktorá v tomto prípade vráti jej číselnú hodnotu.

Prvky jazyka pre podporu .NET

Menné priestory (namespaces) sú užívateľom definované priestory, v ktorých sa nachádzajú jednotlivé deklarácie tried a typov. Slúžia na organizáciu tried a ďalších typov do hierarchickej štruktúry, čo bolo jedným z dvoch dôvodov, prečo som ich pridal do Karlovho jazyka. Druhým bola kompatibilita s prostredím .NET a ostatnými jazykmi.

Väčšina základných tried systému .NET je definovaná v mennom priestore System, ako napríklad System.Object, System.Int32 a System.String. Názvu objektu predchádza bodkou oddelené meno hierarchického umiestnenia. Systém .NET používa hierarchické pomenovávane na zoskupovanie typov do logických[11] kategórií s podobnou funkcionalitou. (Menné priestory sú obdobou balíkov /packages/ v jazyku Java). Menné priestory teda môžu byť vnorené do iných priestorov, ako napríklad System.Data, ktorý obsahuje triedy na prácu s dátami, napríklad System.Data.DataSet. Robot Karol.NET poskytuje syntaktickú skratku – kľúčové slovo „pouzi“. Napríklad po pridaní nasledovného riadku na začiatok programu:

[11] podľa činnosti, nie podľa umiestnenia napríklad na disku. Opačným prístupom sú assemblies spomínané v kapitole „Systém .NET“.

```
pouzi RobotKarol
```

je možné volať metódu "Vycisti" bez zadania menného priestoru, ako ukazujú ďalšie riadky:

```
Miestnost.Vycisti // možno použiť tento riadok
RobotKarol.Miestnost.Vycisti // namiesto tohto
```

Menné priestory sa definujú pomocou príkazu priestor. Môžeme ho použiť nasledovne:

```
priestor [ meno priestoru ]
   [tu deklarované triedy patria do priestoru uvedeného vyššie]
*priestor
```

Ako meno priestoru je možné uviesť navigáciu v hierarchii oddelenú bodkou. Druhá možnosť je vnoriť do seba viacero príkazov „priestor“. Použitie ukazujú dva nasledovné príklady:

```
1.  priestor Vonkajsi
2.      priestor Vnutorny
3.          // ... menný priestor Vonkajsi.Vnutorny
4.      *priestor
5.  *priestor
6.
7.  priestor Vonkajsi.Vnutorny
8.          // ... menný priestor Vonkajsi.Vnutorny
9.  *priestor
```

Jeden assembly môže obsahovať typy, ktorých hierarchické pomenovania sú odlišné, a triedy jedného menného priestoru sa môžu nachádzať v rôznych assembly. Menný priestor je logické pomenovanie, zatiaľ čo assembly je pomenovanie fyzické – určuje názov súboru s binárnym kódom.

Kompilátoru treba pred úspešným skompilovaním uviesť názov a cestu používaných assembly.

Záver k opisu jazyka

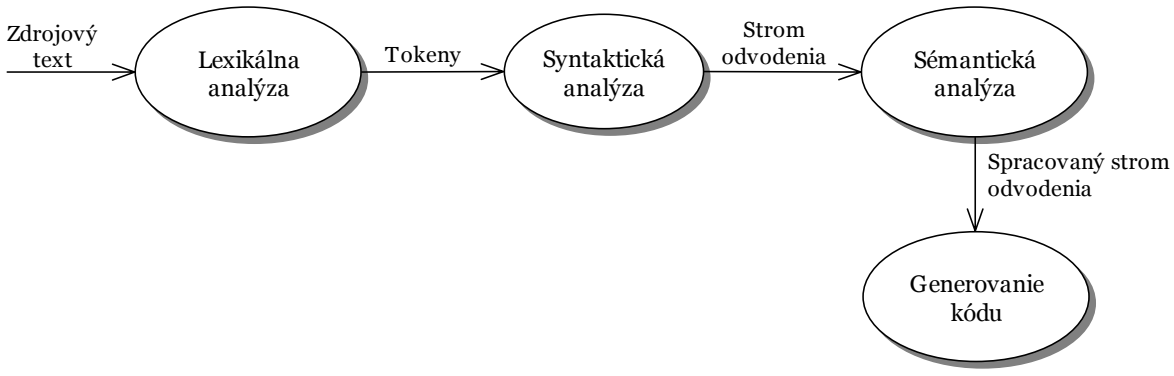
V tejto kapitole som opísal mnou navrhnutý jazyk a motívy, pre ktoré bol rozšírený o spomínané jazykové prvky. Pôvodný jazyk Robota Karola nové návrhy rozširujú o podporu pre objektovo orientované programovanie a premenné. V ďalšej kapitole opíšem, ako funguje kompilátor a moje rozhodnutia pri jeho implementácii.

Kapitola 3
Kompilátor

Táto kapitola sa venuje teoretickému a rozhodovaciemu procesu počas vývoja kompilátora. Obsahuje opis niektorých známych techník používaných pri vývoji kompilátorov a zachytáva moje rozhodnutia pre voľbu niektorých z nich. Rovnako opisuje aj niektoré časti mojej implementácie.

Systémový design

Systémový design je činnosť, pri ktorej sa problém organizuje do menších podproblémov a navrhuje základná systémová architektúra. Keďže oblasť kompilátorov je už veľmi dobre prebádaná, existuje bežne používané rozdelenie na niekoľko podsystémov (fáz), ktoré som dodržal aj ja. Najlepšie je základná činnosť a rozdelenie kompilátora viditeľná z diagramu dátových tokov (data flow diagrams[12]):



V ďalších podkapitolách detailne opisujem metódy, ktoré je možné používať na implementáciu jednotlivých fáz kompilácie a mnou vybrané metódy.

[12] Podľa technológie OMT [10].

Lexikálna analýza

V tejto kapitole sa zameriam na opis práce lexikálneho analyzátora. Je to prvá fáza kompilácie, ktorej hlavnou úlohou je prečítať vstupné znaky a vytvoriť postupnosť „tokenov“, ktoré v ďalšej fáze použije syntaktický analyzátor. Moja implementácia lexikálnej analýzy na začiatku načíta všetky zdrojové súbory a uloží si ich v pamäti. Ďalší postup opíšem detailnejšie.

Medzi dôvody, prečo je vhodné lexikálnu analýzu oddeliť od zvyšku kompilácie patrí:

1. Efektivita a rýchlosť kompilátora. Vďaka oddeleniu sa môžeme venovať len konkrétnej činnosti a využiť špecializované techniky na zrýchlenie načítavania vstupu, napríklad použitím existujúcich techník, ako je práca s bufferom.
2. Zvýšená portabilita kompilátora. Lexikálny analyzátor môže odfiltrovať rôzne odlišnosti medzi jednotlivými platformami (napríklad zakončenie riadkov na UNIXe a Windowse)
3. Jednoduchší design kompilátora. Oddelením tejto fázy sa celkový návrh zjednoduší a sprehľadní.

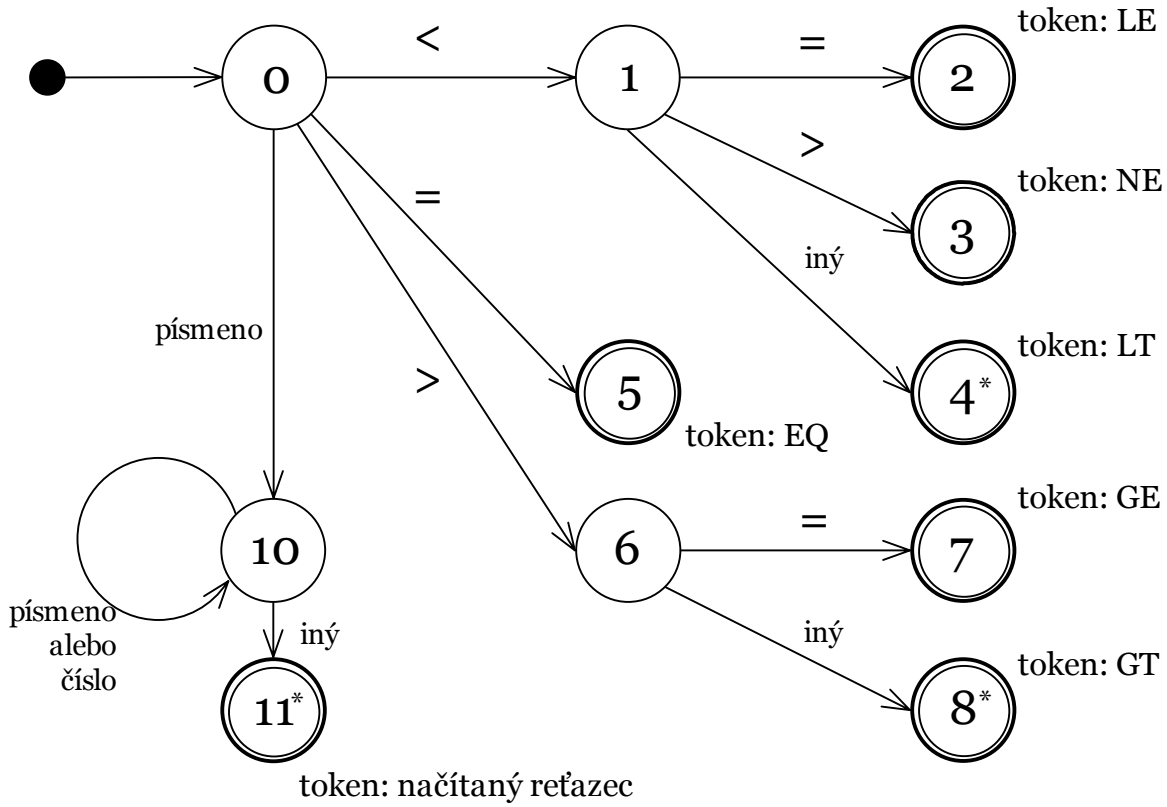
Dôležité pojmy:

1. Slovom **lexéma** budem označovať konkrétnu jazykovú jednotku, ktorú počas lexikálnej analýzy kompilátor rozpozná. Ako príklad možno uviesť: „*pouzi*“, „*opakuj*“, „<“, „>“, „*3.1415*“. Lexémy budem v ďalšom texte označovať kurzívou. Moja implementácia ignoruje veľkosť písmen. Pri zisťovaní, či ide o nejaké kľúčové slovo, je načítaný vstup najprv zväčšený na veľké písmená.
2. Slovom **token** označím druh lexémy. Ak je napríklad lexéma „+“ prípadne “-”, potom token je „binaryoperation“. Tokeny budem v ďalšom texte označovať podčiarknutým písmom.

Rozpoznávanie slov

Pred vytvorením samotného lexikálneho analyzátora opíšem prechodové diagramy, ktoré nám neskôr prácu uľahčia. Prechodový diagram znázorňuje, ako presne získame slovo (lexému) zo vstupu. Stavy v diagrame označujeme krúžkom. Sú navzájom prepojené šípkami (hranami alebo prechodmi). Pre každú hranu sú určené znaky, pre ktoré sa možno z aktívneho stavu presunúť do cieľového. Jeden stav je počiatočný. Niektoré stavy označíme ako akceptačné(na obrázku sú znázornené dvoma sústredenými kruhmi). Činnosť vyzerá takto:

1. Načítaj znak zo vstupu
2. Nájdi hranu, ktorá má na sebe napísaný práve načítaný znak
3. Presuň sa do nového stavu po vybranej hrane
4. Ak prechod na načítaný znak neexistuje, vyhlás chybu
5. Pokračuj od bodu 1



Lexikálna analýza bude používať dva ukazovatele na čítaný text[13]. Jeden sa bude nachádzať vždy na začiatku čítaného slova, pričom druhý (niekedy nazývaný aj „look ahead") bude samotné slovo čítať. Ak sme dočítali slovo, posunie sa na jeho koniec aj prvý ukazovateľ. Prečo to robíme takto? Na obrázku s prechodovým diagramom sú stavy s číslami 4, 8 a 11 označené hviezdičkou. Mohlo by nastať, že napríklad v stave 1 načítame nejaký znak rôzny od = a >. V takomto prípade sa musíme vrátiť o jeden znak späť – a to je presne tam, kde stojí prvý ukazovateľ. V prípade 11 môžeme načítať napríklad znak >, ten však už nepatrí k načítavanému slovu a preto sa takisto potrebujeme vrátiť. Vo veľa prípadoch si vystačíme s pozeraním dopredu len na jeden znak – v mojom jazyku ale občas potrebujeme väčší "look ahead". Napríklad pre slovo „*trieda" vieme určiť, či je to naozaj kľúčové slovo až po prečítaní znaku za posledným písmenom „a". Ak nasledujú za ním ešte ďalšie písmená alebo číslice, pôjde pravdepodobne o násobenie premennou, ktorej názov začína na prefix „trieda".

Lexikálny analyzátor môže používať viacero prechodových diagramov. Každý diagram môže spracúvať niekoľko tokenov. Ľahko si možno predstaviť jeden čítajúci reálne čísla a jeden operátory (<, <=, <>, ...). Aj v tomto prípade môžeme využiť ukazovatele. Ak počas práce s jedným prechodovým diagramom zlyháme, stačí vrátiť „look ahead" ukazovateľ na pozíciu prvého a pokračovať s ďalším diagramom, dokým nenájdeme diagram, ktorý token rozpozná. V prípade, keď vhodný diagram neexistuje, znamená to, že slovo nie je z nášho jazyka.

Implementácia prechodových diagramov

Existuje niekoľko metód na prevod prechodových diagramov na kód. Opíšem metódu, pri ktorej pre každý stav vytvoríme obslužnú rutinu. Ak zo stavu vychádzajú nejaké hrany, potom načítame jeden znak(posunieme druhý ukazovateľ) a podľa neho vyberieme hranu, ktorou budeme pokračovať. Teda budeme pokračovať v kóde, ktorý obsluhuje nový stav. Ak nenájdeme vhodnú hranu a stav, v ktorom sa nachádzame, nie je akceptačný, vrátime druhý ukazovateľ

[13] v teórii formálnych jazykov je to upravený konečný automat s dvomi hlavami. Jazyk, ktorý lexikálna analýza rozpoznáva je regulárny. Pre väčšinu jazykov nám to v lexikálnej analýze stačí.

na začiatok slova a pokračujeme rovnako len s ďalším diagramom. Ak už nie je s akým diagramom pokračovať, kompilátor ohlási chybu.

```
1.  verejny staticky prikaz Diagram1 () : cislo
2.      kym pravda rob // cyklus je ukončený volaním "vrat"
3.          ak pre Stav plati
4.              pripad 0:
5.                  la = Vstup.LookAhead (); // posúva druhý ukazovateľ
6.                  ak pre la plati
7.                      pripad "<": Stav = 1; prerus;
8.                      pripad "=": Stav = 5; prerus;
9.                      pripad ">": Stav = 6; prerus;
10.                     pripad "a"..."z": Stav = 10; prerus;
11.                     inak Vstup.LANavrat(); vrat -1;
12.                 *ak;  prerus;
13.             pripad 1:
14.                 la = Vstup.LookAhead ();
15.                 ak pre la plati
16.                     pripad "=": Stav = 2; prerus;
17.                     pripad ">": Stav = 3; prerus;
18.                     inak: Stav = 4; prerus;
19.                 *ak
20.                 prerus;
21.             ... // obsluha ďalších stavov
22.         *ak
23.     *kym
24. *prikaz
```

Bufferovanie vstupu

Pri spracovávaní vstupného zdrojového kódu sa často potrebujeme vrátiť naspäť – doteraz som spomínal riešenie pomocou look ahead ukazovateľa. Problém nastáva, ak celý zdrojový kód nenačítame naraz do pamäti a potrebujeme si udržiavať len jeho časť. Veľmi rozšírené riešenie používa techniku dvoch bufferov (presnejšie

jedného buffera rozdeleného na dve časti). Najprv naplníme prvú časť dátami jedným systémovým čítaním z disku. Keď sa look ahead ukazovateľ má presunúť z prvej časti do druhej, dočítame do nej ďalšiu časť vstupu. Ak sa look ahead dostane na koniec druhej časti bufferu, do prvej dočítame ďalšiu časť vstupu a ukazovateľ posunieme na jeho začiatok. Tento postup naráža len na jeden problém – ak je dĺžka lexémy väčšia, ako je veľkosť buffera. Našťastie, toto nie je príliš časté a v mojom návrhu jazyka sa takýto problém nevyskytuje (nenachádzajú sa tu štruktúry, ktoré by na rozpoznanie potrebovali veľmi „dlhý“ look ahead a dĺžka slov je obmedzená).

Tento postup možno urýchliť. Stačí si všimnúť, že pri každom posune robíme dva testy (“*sme na konci prvej časti?*” a “*sme na konci druhej časti?*”). Stačí zaviesť nový špeciálny znak, o ktorom vieme, že sa na vstupe nevyskytne(pravdepodobne to bude znak EOF) a umiestniť ho na konce obidvoch častí. Vďaka tomu stačí vždy testovať, či sa nenachádzame na špeciálnom znaku a iba v prípade, že áno, spraviť ďalšie testy a správne sa presunúť. Ukážkový kód vyzerá takto:

```
1.  ak Vstup.LookAhead () == EOF tak
2.      ak Vstup.LAPozicia == Vstup.Buffer.Velkost / 2 tak
3.          Vstup.Buffer.NacitajDruhuCast;
4.          Vstup.LookAhead ()
5.      inak
6.          ak Vstup.LAPozicia == Vstup.Buffer.Velkost / 2 tak
7.              Vstup.Buffer.NacitajPrvuCast;
8.              Vstup.LookAhead.Pozicia = 0;
9.          *ak
10.     *ak
11. *ak
```

Uvedenú metódu používam pri lexikálnej analýze v mojej implementácii, samozrejme s drobnými zmenami. Na čítanie zdrojových súborov som implementoval špeciálnu triedu SeekableStreamReader schopnú „look-aheadu“.

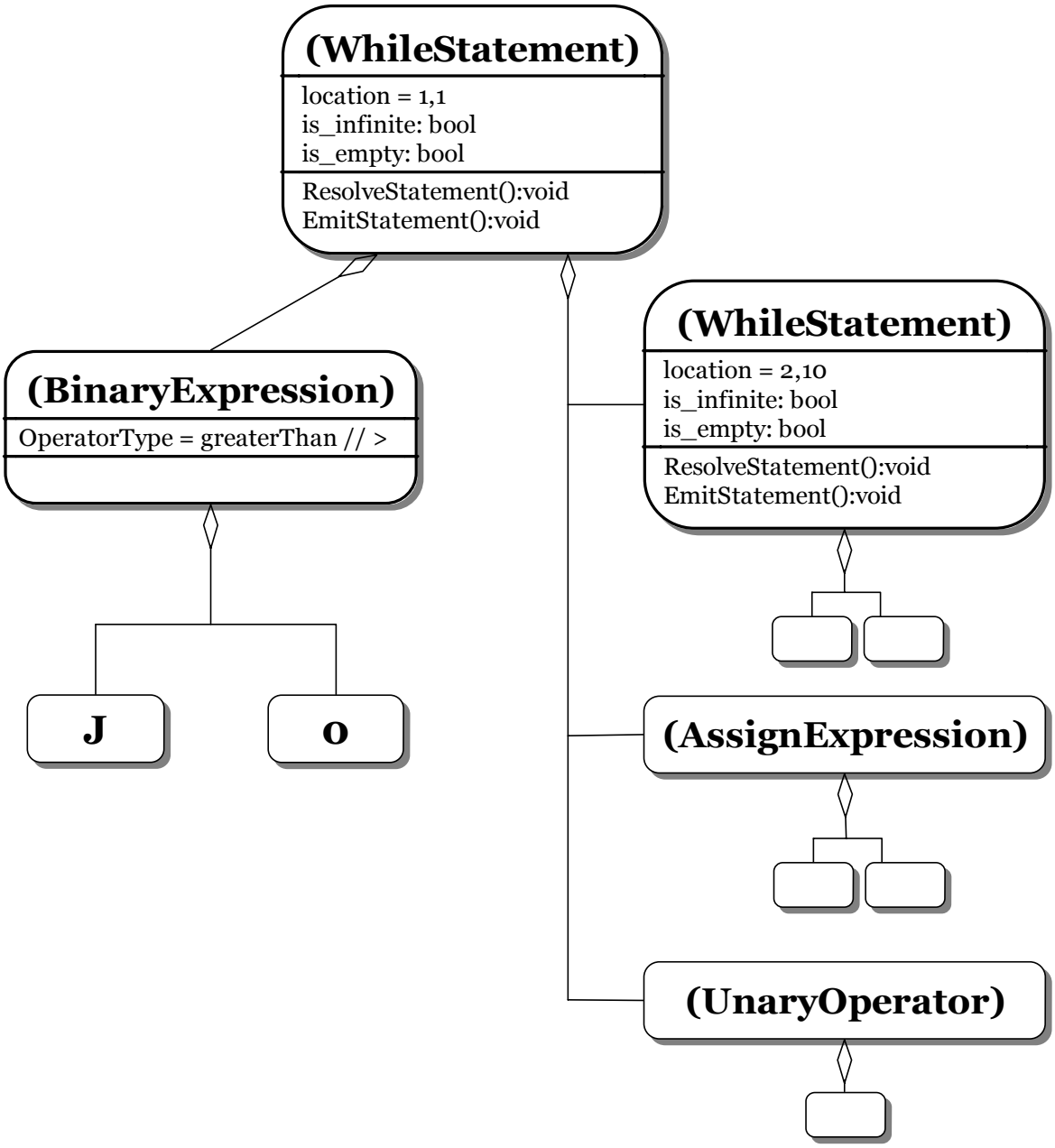
Syntaktická analýza

Syntaktická analýza na vstupe dostáva postupnosť tokenov z lexikálnej analýzy a vytvára strom odvodenia. Kompilátor na uloženie informácií o spracovanom vstupe používa triedy, ktorých mená reprezentujú jazykové konštrukcie, napríklad vetvenie „ak“ je uložené v triede pomenovanej „IfStatement“. Informácie o triede sú uložené v triede pomenovanej „ClassStatement“. Pri kompilovaní sa počas syntaktickej analýzy vytvára strom založený na týchto triedach. Ako príklad uvediem nasledovný útržok z programu:

```
1. kym j>0 rob
2.         kym k<j rob
3.             Napis( "*" );
4.             k = k + 1;
5.         *kym
6.         k = 0;
7.        j--;
8. *kym
```

Jemu zodpovedajúci strom tried má v koreni triedu WhileStatement, ktorá má referencie na ďalšie triedy: jedna z nich obsahuje informácie o parametroch (podmienke) cyklu a ďalšie obsahujú telo cyklu. Telo cyklu tvoria tri triedy, prvá obsahuje informácie o ďalšom cykle, druhá o priradení (AssignExpression) a tretia o dekrementácii (UnaryOperator).

Nižšie nachádzajúci sa diagram je diagram podľa objektového modelu OMT. Obdĺžniky so zaokrúhlenými vrcholmi reprezentujú inštancie tried. Pod názvom v zátvorkách uvádzam niektoré atribúty ktoré používam a prípadne metódy. Spojenia medzi inštanciami sú typu „agregácia“.



Vo všeobecnosti metódy používané na syntaktickú analýzu možno rozdeliť do troch skupín:

1. univerzálna: patrí sem napríklad CYK(nezávisle objavený Cockom a Youngerom v roku 1967 a Kasamim v roku 1965) alebo Erleyho algoritmus. Metódy v tejto skupine dokážu spracovať veľkú množinu jazykov, sú však

pomalé a náročné na výpočtovú silu. Pre môj jazyk nie sú potrebné a preto sa nimi nebudem zaoberať.

2. zhora-nadol (top-down)
3. zdola-nahor (bottom-up)

Posledné dve metódy spracúvajú vstup pri vytváraní stromu zľava doprava, pričom top-down vytvára syntaktický strom od hora smerom k listom a bottom-up z dola smerom ku koreňu. Obe pracujú pomerne rýchlo a dokážu rozpoznať len niektoré jazyky – napríklad generované LL alebo LR gramatikami. Oba druhy gramatík sú však dostatočne „výrečné", aby nám pri návrhoch programovacích jazykov stačili.

Syntaktickú analýzu zhora nadol som sa rozhodol nepoužiť. Jedným z dôvodov bola absencia nástrojov umožňujúcich automatické generovanie pre C#, druhým bolo to, že som už mal skúsenosti s použitím syntaktickej analýzy zdola nahor. Ďalším dôvodom bolo, že sa mi zdalo jednoduchšie vytvoriť a napísať gramatiku, ako ručne vytvárať parser na analýzu zhora nadol.

Prístup zdola-nahor k syntaktickej analýze sa niekedy označuje aj ako „shift-reduce"[14]. Metódy založené na tomto prístupe sa snažia vytvoriť strom odvodenia smerom od listov ku koreňu (k počiatočnému symbolu gramatiky). Celý proces si možno predstaviť ako postupné redukovanie vstupného reťazca na počiatočný symbol gramatiky spätným nahradzovaním pravidiel (hľadaním pravej strany a jej prepísaním za ľavú, tento krok sa nazýva reduce; krok keď sa posúvame ďalej na vstupe nazývame shift a prečítané znaky ukladáme do zásobníka). Existuje niekoľko metód pracujúcich na opísanom princípe (SLR[15], Canonical LR, ...).Dnes je v praxi najviac používaná metóda LALR (Look Ahead LR). LALR vytvára menšie parsovacie tabuľky ako Canonical LR a zároveň dokáže rozpoznať niektoré jazyky,

[14] Do slovenčiny býva tento termín niekedy prekladaný ako posunovo redukčná schéma

[15] L v skratke znamená left-to-right, čo označuje smer čítania vstupu a R znamená rightmost derivation, teda najpravejšie možné odvodenie (prioritne používame pravidlá na neterminály nachádzajúce sa vpravo vetnej formy). S označuje Simple.

ktoré SLR nedokáže. LALR metódu používam pri vytváraní syntaktickej analýzy aj ja.

V mojom prípade je syntaktická analýza implementovaná pomocou upraveného generátora Yacc pre jazyk Java[22], s ďalšími úpravami pre jazyk C#. Jazykovému generátoru som nadefinoval gramatika jazyka a zodpovedajúce programové akcie. Uvádzam malú časť z gramatiky rozpoznávajúcej jazyk Karola, programový kód som odstránil:

```
1.  statement_iterations
2.          : while_statement
3.          | repeat_statement // for
4.          | foreach_statement
5.          ;
6.  while_statement
7.          : WHILE boolean_expression DO
8.                  statement_list_optional END_WHILE
9.          ;
10. try_statement
11.         : TRY opt_statement_list opt_try_catch opt_try_finally END_TRY
12.         ;
13. opt_try_finally
14.         : /* prazdne */
15.         | try_finally
16.         ;
17. try_finally
18.         : FINALLY opt_statement_list
19. opt_try_catch
21.         : /* prazdne */
22.         | catch_clauses
23.         ;
24. catch_clauses
25.         : catch_clause
26.         | catch_clauses catch_clause
```

```
27.         ;
28. catch_clause
29.         : CATCH opt_catch_args opt_statement_list ;
30. opt_catch_args
31.         : /* prazdne */
32.         | catch_args
33.         ;
34. catch_args
35.         : OPEN_PARENS type opt_identifier CLOSE_PARENS
36.         ;
```

Z uvedenej časti gramatiky vidno štruktúru cyklu „kym“ a príkazu „skus“. Celú gramatiku možno nájsť na priloženom CD.

Ošetrenie chýb

Ak kompilátor nájde počas svojej práce chybu – môže ju ohlásiť a skončiť. V praxi sa viac používajú metódy, keď sa kompilátor pokúsi z chyby dostať a pokračovať ďalej. Cieľom je, aby kvôli jednej chybe nemusel ísť celý proces odznova a aby mal programátor informáciu naraz o viacerých chybách. Existujú 4 základné stratégie:

1. panický mód: Ak narazí syntaktická analýza na chybu, zahadzuje prichádzajúce tokeny dovtedy, kým nenájde nejaký, ktorý je označený ako „bezpečný“ (označované aj ako synchronizačný). Až po tomto tokene sa pokúsi pokračovať v spracovávaní ďalej. Príkladom bezpečného tokenu môže byť *bodkočiarka* a „*}*” v jazyku C alebo *end* v Pascale.
2. oprava konštrukcií jazyka: Ak nájdeme chybu, pokúsime sa ju opraviť. Častým príkladom môže byť doplnenie bodkočiarky. Metódu nepoužívam, pretože opravený výsledok často môže zmeniť význam programu v porovnaní s tým, čo mal na mysli programátor.
3. chybové pravidlá: V tomto prípade sa doplnia do gramatiky jazyka nové pravidlá, každé pre nejakú známu chybu. Napríklad v jazyku C vždy po if nasleduje zátvorka. Môžeme vytvoriť pravidlo, ktoré v opačnom prípade

ohlási chybu „*očakávaná ľavá zátvorka*“. Chybové pravidlá som sa snažil doplniť do pravidiel, ktoré som považoval za „najrizikovejšie“.

4. globálna oprava: Metódy založené na tejto stratégii využívajú špeciálne algoritmy, ktoré dokážu vrátiť pre chybný vstupný text výsledok, v ktorom bolo urobených minimum úprav(doplnenie znaku, ...) a ktorý je gramatikou generovateľný. Tieto algoritmy sú však veľmi náročné na výpočtovú silu a preto sa v praxi nepoužívajú. Zároveň si treba uvedomiť, že program, ktorý dá ako výstup opravný algoritmus nemusí byť to, čo myslel autor. Z tohto dôvodu som metódu nepoužil.

Správa typov a strom dedičností pre rozhrania a triedy

Po vykonaní syntaktickej analýzy mám zdrojový text prevedený do stromovej štruktúry. Možno teda pristúpiť k ďalšej časti.
Správa typov načíta všetky programátorom spomenuté a použité „assemblies“. Štruktúra, ktorú na udržanie potrebných informácií používam je CLR systém behového prostredia .NET.
Počas ďalšieho generovania kódu a sémantickej analýzy vkladám do tej istej štruktúry aj nové typy pomocou volania System.Reflection.Emit.
Vďaka tomu možno použiť rovnaké rozhranie na získavanie informácií o typoch, ktoré boli natiahnuté z assemblies ako aj o typoch deklarovaných v zdrojovom kóde.

Po tom, ako sa ukončí vytváranie stromu, kompilátor spracuje a určí strom dedičností pre rozhrania. Toto spraví rekurzívne. Počas činnosti zachytí rekurzívne (zacyklené, teda nepovolené) definície rozhraní. Po spracovaní a určení stromu pre rozhrania pokračuje rovnako s triedami. Triedy môžu mať zadaného predka alebo implicitne dediť od triedy System.Object. V tomto kroku kompilátor overí niektoré chyby – napríklad či implementovanie rozhraní je správne (toto možno overiť vďaka tomu, že už poznáme všetky rozhrania). V tomto bode teda máme pomocou volania System.Reflection.Emit vytvorené tieto jazykové prvky:

1. rozhrania
2. triedy

Sú tiež zaregistrované do správcu typov (všetko zatiaľ len ako prázdne triedy, ktoré nič neimplementujú).

Generovanie prvkov jazyka

Pre vyššie spomenuté jazykové prvky (rozhrania, triedy) nájdeme v strome odvodenia ich metódy a vlastnosti, ktoré vložíme cez System.Reflection.Emit k ich triedam. Negenerujeme ešte ich kód (to ešte nie je v tomto štádiu možné – na samotné generovanie kódu budeme už potrebovať tieto informácie kompletne spracované). Po tomto štádiu teda viem o každej triede, rozhraní a o ich metódach.

Po vložení definícií tried a rozhraní možno pristúpiť ku generovaniu kódu. Pomocou System.Reflection možno zisťovať informácie o typoch – aj o našich práve vytvorených aj o typoch z iných assemblies. Môžeme teda pristúpiť k samotnému generovaniu MSIL. Platforma .NET obsahuje menný priestor „System.Reflection.Emit“, v ktorom sa nachádzajú triedy umožňujúce kompilátorom generovať jazyk MSIL a prípadne ho uložiť na disk[23].

Vytvorenie jednoduchého EXE súboru ukážem na nasledujúcom príklade. Zároveň má možnosť čitateľ prezrieť si aj MSIL inštrukcie [24], [25], [26]. Chcem vytvoriť program, ktorý na obrazovku trikrát vypíše „Ahoj Robot Karol!“ do samostatného riadku, pričom každý výpis bude očíslovaný hodnotou od 0 do 2.
Riadky 1. až 7. v nasledujúcom výpise vytvoria potrebné triedy z System.Reflection.Emit. Trieda assembly reprezentuje jeden binárny súbor, trieda „module“ zase modul a „method“ definuje telo metódy. Na riadku 8. označím túto metódu ako vstupný bod pre celý assembly – sem bude odovzdané riadenie po spustení programu. Na riadku 9. si vypýtam referenciu na triedu ILGenerator, ktorá je zodpovedná za generovanie kódu. Na ďalších troch riadkoch vytvorím lokálnu premennú typu „cislo“, na ktorú sa môžem odkazovať pomocou referencie „iteratorVariable“. Volania na riadkoch 15. a 17. zoberú potrebné volania externých metód:

1. System.Console.WriteLine(retazec) – metóda vypíše reťazec na štandardný výstup.
2. System.String.Concat(objekt, objekt) – metóda spája dva reťazce[16].

Ďalšia časť programu už generuje inštrukcie MSIL. Vymenovanie OpCodes obsahuje zoznam všetkých inštrukcií, ktoré systém podporuje. Napríklad na riadku 24. sa nachádza skok na návestie. Tu si treba všimnúť, že trieda ILGenerator nám dokáže vytvoriť skok na ešte neurčené návestie. Nemusíme sa starať o spätné dopĺňanie/zaplátanie (backpatching), pretože túto funkciu za nás vyrieši systém .NET.

```
1.  AssemblyBuilder assembly = curDomain.DefineDynamicAssembly ( ... );
2.  ModuleBuilder module = assembly.DefineDynamicModule ( ... );
3.  MethodBuilder method = module.DefineGlobalMethod (
4.                      "HelloWorld",
5.                      MethodAttributes.Public | MethodAttributes.Static,
6.                      vrattyp (prazdny),
7.                      nic);
8.  assembly.SetEntryPoint (method, PEFileKinds.ConsoleApplication);
9.  ILGenerator mainIL = method.GetILGenerator ();
10.
11. LocalBuilder iteratorVariable = mainIL.DeclareLocal (vrattyp (cislo));
12. Label endPoint = mainIL.DefineLabel ();
13. Label startPoint = mainIL.DefineLabel ();
14.
15. MethodInfo writeMethod = vrattyp (Console).GetMethod("WriteLine", novy Type []
16.                     { vrattyp (retazec)});
17. MethodInfo concatMethod = vrattyp (String).GetMethod("Concat", novy Type []
18.                     { vrattyp (objekt), vrattyp (objekt)});
19.
21.
22. mainIL.Emit (OpCodes.Ldc_I4, 0);        // nastavenie iteračnej premennej na 0
23. mainIL.Emit (OpCodes.Stloc, 0);
24. mainIL.Emit (OpCodes.Br_S, endPoint);  // nepodmienený skok na návestie
```

[16] V MSIL kóde neexistujú inštrukcie na spájanie reťazcov. Operátor +, používaný v jazyku Robota Karola je v skutočnosti preťaženým operátorom.

```
25.  mainIL.MarkLabel (startPoint);                    // označ nové návestie
26.  mainIL.Emit (OpCodes.Ldloc, 0);
27.  mainIL.Emit (OpCodes.Box, vrattyp (cislo));
28.  mainIL.Emit (OpCodes.Ldstr, ": Ahoj Robot Karol!");
29.
30.  mainIL.Emit (OpCodes.Call, concatMethod);    // spojenie dvoch objektov
31.  mainIL.Emit (OpCodes.Call, writeMethod);      // vypisanie vystupu
32.
33.  mainIL.Emit (OpCodes.Ldloc, 0);
34.  mainIL.Emit (OpCodes.Ldc_I4, 1);              // inkrementuj o jedna
35.  mainIL.Emit (OpCodes.Add);                    // inkrementovanie
36.  mainIL.Emit (OpCodes.Stloc, 0);
37.
38.  mainIL.MarkLabel (endPoint);
39.  mainIL.Emit (OpCodes.Ldloc, 0);
40.  mainIL.Emit (OpCodes.Ldc_I4, 3);              // porovnanie, štvorbytové
41.  mainIL.Emit (OpCodes.Blt_S,startPoint);       // skoč na začiatok cyklu ak
42.
43.  mainIL.Emit(OpCodes.Ret);                     // koniec metódy
44.  module.CreateGlobalFunctions ();
45.
46.  assembly.Save (outputFileName);              // ulož EXE súbor
```

Takto je vlastne do súboru vygenerovaný cyklus „opakuj“ pre nejakú číselnú premennú, s krokom jedna, od 0 do 2. Každá z tried[17] reprezentujúcich jazykový prvok (napríklad trieda WhileStatement, IfStatement, ForStatement, ...) má svoju vlastnú implementáciu metódy EmitStatement, ktorá vytvára podobný kód ako v uvedenom príklade.

Generovanie teda začína vytvorením triedy assembly a zavolaním metódy EmitStatement na najvrchnejší uzol stromu. Ten postupne volá metódy EmitStatement tried, ktoré agreguje. Takto dostaneme celý skompilovaný kód a uložíme ho do súboru. Obrovské zjednodušenie nám poskytuje .NET. Kým normálne ešte potrebujeme backpatching na doplnenie správnych skokov do ešte

[17] Triedy sú spomínané v úvode kapitoly venovanej syntaktickej analýze

nevygenerovaného kódu, v tomto prípade sa o túto funkciu postará trieda ILGenerator.

Optimalizácia

Kompilátor v súčasnej dobe používa niekoľko jednoduchých optimalizačných techník: vyhodnocovanie konštánt (teda ak užívateľ zadá hodnotu ako 10*5, kompilátor výraz vyhodnotí a ďalej pracuje len s číslom 50). Ďalej je kompilátor schopný vynechať „mŕtvy“ nepoužívaný kód.

O odstraňovanie mŕtveho kódu sa stará samostatná trieda „FlowBranchingChecker“ a jej potomkovia (v závislosti od kontrolovaného kódu, napríklad FlowBranchingCheckerLoop, FlowBranchingCheckerBlock). Trieda dokáže zistiť, či je daný kód nedosiahnuteľný a či v prípade nutnosti vracia hodnotu. Rozlišuje 3 stavy: vracia hodnotu vždy, vracia hodnotu niekedy, nevracia hodnotu. Podľa toho kompilátor buď preruší kompilovanie, ohlási varovanie alebo bez chyby pokračuje ďalej.

Optimalizácia by sa dala rozšíriť o niektoré z metód kukátkovej optimalizácie:

1. Detekcia a odstránenie „flow-of-control“, kedy sa skok na ďalší skok nahradí jednou inštrukciou, ktorá riadenie odovzdá rovno na cieľové návestie.
2. Viacnásobné ukladanie a načítavanie z toho istého miesta v pamäti
3. Ďalším možným rozšírením by bola optimalizácia aritmetických výrazov, napríklad odstránenie násobenia číslom 1 a sčitovania s nulou.

Z hľadiska Robota Karola a môjho cieľu považujem implementáciu týchto metód za zbytočnú. Omnoho dôležitejšie pre jeho použiteľnosť bude napríklad vytvorenie kvalitného grafického prostredia.

Kapitola 4
Záver

Cieľom mojej diplomovej práce bolo navrhnúť a naprogramovať nový programovací jazyk, ktorý by sa svojimi rozšíreniami vyrovnal dnes používaným jazykom. Všetky spomínané jazykové konštrukcie sa mi podarilo naprogramovať. Zdrojový kód má približne 25 000 riadkov.

Na záver by som chcel konštatovať, že touto prácou som tému Karola ani zďaleka nevyčerpal. Jej rozsiahlosť je taká veľká, že by sa o nej dali napísať ďalšie práce.

Pre samotného Karola je dôležité aj vytvorenie 3D modelu miestnosti, ktoré by bolo na dnešnej úrovni také, aby bolo príťažlivé napríklad aj v porovnaní s počítačovými hrami. Ďalšou dôležitou vlastnosťou je kvalitné užívateľské prostredie. V práci som navrhol výzor editoru schopného skrývať niektoré časti programu. Prostredie by malo byť aj „didakticky“ vhodné, čo je však práca skôr pre študentov iných odborov, najmä učiteľských.

Súčasťou mojej práce je aj niekoľko príkladov a programov napísaných v jazyku, ktorý som navrhol. Programy som používal počas vývoja na testovanie a neobjavil som žiadne nedostatky alebo chyby v programe. Programy možno nájsť na priloženom disku CD, ich popis možno nájsť aj v kapitole Prílohy.

Kapitola 5
Referencie

[1] Brusilovsky, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A., Miller, P. (1997) Mini-languages: A Way to Learn Programming Principles. Education and Information Technologies 2, strany. 65-83.

[2] Playground: an object-oriented simulation system with agent rules for children of all ages, http://portal.acm.org/citation.cfm?id=74891

[3] Pattis, Richard E., Jim Roberts, and Mark Stehlik. Karel the Robot: A Gentle Introduction to The Art of Programming. New York: Wiley, 1995. ISBN: 0471597252

[4] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools

[5] Antony L. Hosking., Compilers: Principles and Practice. Addison Wesley, Reading Mass. 1986

[6] RNDr. Ján Šturc, CSc. Kompilátory, poznámky k prednáškam

[7] A. V. Aho, J. D. Ullman, Compiler Design. Addison Wesley. 1977

[8] C. N. Fisher, R. J. Leblanc, Crafting A Compiler. Cummings Publishing Company, Inc, 1988

[9] Jiří Dobeš. Prekladače. Skriptá z MFF Univerzita Karlova

[10] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, Object-oriented modeling and design, Prentice-Hall, Inc., 1991

[11] Michal Zeman, Diplomová práca Karel 2010, Fakulta matematiky, fyziky a informatiky, 2004

[12] Ľuba Gašparovičová, Jozef Hvorecký, Kamaráti robota Karla. Bratislava: Mladé letá, 1991

[13] Ulli Freiberger, http://www.schule.bayern.de/karol/

[14] Byron Weber Becker, Teaching CS1 with Karel the Robot in Java, University of Waterloo,

[15] Joseph Bergin, Mark Stehlik, Jim Roberts, Richard Pattis: http://csis.pace.edu/%7Ebergin/KarelJava2ed/Karel++JavaEdition.html

[16] Christoph Bockisch, http://www.cbockisch.de/projects/karelj/

[17] Karel 3D pre Windows, Diplomová práca, Košút, Univerzita Komenského, 1997, Bratislava

[18] Common Language Infrastructure Standards, http://msdn.microsoft.com/net/ecma/

[19] ISO/IEC, Common Language Infrastructure http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?csNUMBER=36769

[20] DotGNU Project, http://www.dotgnu.org/

[21] Project Mono, http://www.mono-project.com/

[22] Projekt Jay, http://www.informatik.uni-osnabrueck.de/alumni/bernd/jay/

[23] MSDN Library, Emitting Dynamic Assemblies, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconemittingdynamicassemblies.asp

[24] CodeGuru; MSIL Tutorial; http://www.codeguru.com/Csharp/.NET/net_general/il/article.php/c4635/

[25] John Robbins; MSDN Magazine, May 2001, Microsoft Journal for Developers; ILDASM is Your New Best Friend

[26] MSDN Library, .NET Framework Class Library, OpCodes, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemreflectionemitopcodesmemberstopic.asp

[27] Eamon O'Tuathail; The Portable Executable (PE) Image Format with CLI Extensions, http://www.clipcode.biz/stream/IL_BINARY_FORMAT.pdf

[28] ECMA TC39/TG3 Document, Common Language Infrastructure (CLI) Partition III, CIL Instruction Set

Kapitola 6
Prílohy

K práci je priložené CD so zdrojovými kódmi a skompilovaným projektom.
V adresári Test možno nájsť ukážkové programy a zároveň aj dávkové súbory, ktorými možno príklady skompilovať. Pred kompilovaním ich treba skopírovať na disk s právami na zápis a mať nainštalovaný Microsoft .NET Framework. Ten je voľne stiahnuteľný zo stránky Microsoftu, prípadne ho možno nainštalovať z CD.
Príklady sú rozdelené do viacerých podadresárov:

1. V adresári Language_Features možno nájsť krátke programy, ktoré ukazujú jednotlivé vlastnosti jazyka. Časť z týchto príkladov bola uvádzaná v kapitole Opis jazyka.

2. V adresári Ook možno nájsť dva jednoduché kompilátory napísané v jazyku Robota Karola a niekoľko príkladov k nim.

3. V adresári Reversi možno nájsť jednoduchú hru.

4. V adresári Karol je jednoduchá trieda, ktorá dokáže simulovať Karlové správanie (krok, vľavo atď)

V adresári Dokumenty sa nachádza elektronická verzia tejto práce a dodatok o činnosti syntaktickej analýzy metódami bottom-up a top-down, ktorý som z dôvodu dĺžky práce z nej odstránil.