PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

DIPLOMOVÁ PRÁCE

Interpret objektově orientovaného programovacího jazyka

2006 Radovan Šimek

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval samostatně.

15. srpna 2006 Radovan Šimek

Anotace

Cílem této diplomové práce bylo navrhnout a implementovat interpret objektově orientovaného programovacího jazyka. Objektově orientovaný přístup a jazyky přinášejí, především díky dědičnosti a polymorfismu, spoustu zajímavých vyjadřovacích schopností. Jejich interprety a překladače proto obsahují poměrně velké množství algoritmů zajišťujících syntaktickou a sémantickou analýzu a překlad těchto prvků. Interpret je navržen též pomocí objektově orientovaných technologií a implementován v jazyce C# 2.0 pro platformu .NET 2.0. Samotný interpretovaný jazyk, nazvaný MiniC#, vychází z jazyka C# 1.2, je ovšem omezen na základní koncepty objektově orientovaného přístupu.

Děkuji RNDr. Arnoštu Večerkovi za rady, připomínky a odbornou literaturu, které mi poskytl při vedení mé diplomové práce.

Obsah

1. Úvod

Téma, jímž se zabývá tato diplomová práce, zasahuje do dvou oblastí informatiky. Jsou jimi jednak *kompilátory a interprety* (obecně *překladače*) a jednak *objektově orientovaný přístup* tvorby softwaru. Konstrukce překladačů masivně využívá poznatků *teorie formálních jazyků a automatů*. Součástí objektově orientovaného přístupu je *objektově orientované paradigma programování*, které je dnes nejpoužívanějším *programovacím paradigmatem* (stylem). Důležité poznatky z těchto oblastí informatiky pro tuto diplomovou práci jsou uvedeny v následujících podkapitolách.

1.1. Formální jazyky, gramatiky a automaty

Teorie formálních jazyků, gramatik a automatů poskytuje prostředky pro popis a automatické (algoritmické) rozpoznání vět napsaných v určitém jazyce. Rozvoj této teorie přinesla potřeba konstrukce překladačů pro programovací jazyky. Aby mohl být program ve zdrojovém jazyce překladačem převeden do cílového jazyka (např. strojového kódu), je potřeba nejprve zdrojový text rozpoznat a rozčlenit na základní části (syntaktická analýza). Poté následuje zpracování významu takto rozčleněného textu (sémantická analýza) a generování textu v cílovém jazyce.

Formální jazyky

Definice 1.1. Konečná neprázdná množina Σ symbolů z univerza U se nazývá *abeceda*.

Definice 1.2. Konečná posloupnost α symbolů z abecedy Σ se nazývá *řetězec* nad abecedou Σ. Počet symbolů v řetězci α udává *délku řetězce*, kterou značíme $|\alpha|$. Řetězec s délkou 0 se nazývá *prázdný řetězec* a obvykle se označuje ε.

Definice 1.3. *Zřetězením* (konkatenací) řetězců $\alpha = a_1 a_2 \dots a_m$ a $\beta = b_1 b_2 \dots b_n$ je řetězec $\alpha\beta = a_1 a_2 \dots a_m b_1 b_2 \dots b_n$.

Pro délku konkatenace $\alpha\beta$ dvou řetězců α a β platí $|\alpha\beta| = |\alpha| + |\beta|$. Dále pro konkatenaci libovolného řetězce α s prázdným řetězcem ε platí $\alpha\varepsilon = \alpha\varepsilon = \alpha$.

Definice 1.4. Nechť Σ je abeceda. Množinu všech neprázdných řetězců sestavených ze symbolů abecedy Σ nazýváme *pozitivní uzávěr* množiny Σ a značíme Σ^+.

Definice 1.5. Nechť Σ je abeceda. Množinu všech řetězců sestavených ze symbolů abecedy Σ nazýváme *uzávěr* množiny Σ a značíme Σ^*.

Uzávěr abecedy Σ tedy na rozdíl od pozitivního uzávěru téže abecedy obsahuje i prázdný řetězec ε. Délka řetězců v pozitivním uzávěru ani v uzávěru abecedy není nijak shora omezena, tedy pozitivní uzávěr i uzávěr jsou vždy nekonečné, ale spočetné množiny.

Definice 1.6. Nechť Σ je abeceda, pak libovolná podmnožina uzávěru Σ^* se nazývá *formální jazyk* L nad abecedou Σ. Tedy $L \subseteq \Sigma^*$.

Gramatiky

Uvedená definice formálního jazyka je příliš obecná, definuje totiž jazyk pouze pomocí výčtu řetězců, které obsahuje. Dále formální jazyk je libovolná podmnožina uzávěru abecedy, tedy může být nekonečný. Zde se ukazuje, že definice nekonečného jazyka pomocí (nekonečného) výčtu řetězců, které obsahuje, prakticky není možná. Dále, tato definice formálního jazyka nic neříká o vnitřní struktuře jeho řetězců, ani o tom, jak určit, jestli daný řetězec do jazyka patří nebo ne. Tyto problémy odstraňují gramatiky, které je konečnými a strukturovanými popisy (i nekonečných) formálních jazyků.

Definice 1.7. *Gramatika* G je čtveřice (N, T, P, S), kde

- N je abeceda tzv. neterminálních symbolů,
- T je abeceda tzv. terminálních symbolů, Množiny terminálních a neterminálních symbolů jsou disjunktní ($N \cap T = \emptyset$),
- P je konečná množina pravidel tvaru $\alpha \to \beta$, kde $\alpha \in (N \cup T)^* N (N \cup T)^*$, $\beta \in (N \cup T)^*$,
- S je tzv. startovní symbol gramatiky.

Gramatické pravidlo $\alpha \to \beta$ tzv. *přepisuje* řetězec α na řetězec β. V řetězci α se dle definice musí vyskytovat alespoň jeden neterminální symbol, řetězec β může být libovolný řetězec terminálních a neterminálních symbolů gramatiky.

Neterminální symboly slouží k tzv. odvození vět jazyka a terminální symboly tvoří věty jazyka, který gramatika popisuje.

Definice 1.8. Nechť je dána gramatika $G = (N, T, P, S)$, o řetězci $\tau = \mu\beta\omega$ řekneme, že je *přímou derivací* řetězce $\sigma = \mu\alpha\omega$ ($\mu,\ \omega \in (N \cup T)^*$), jestliže $\alpha \to \beta$ je pravidlo gramatiky G, tedy jestliže pravidlo $\alpha \to \beta$ přepisuje některou část řetězce σ, čímž vznikne řetězec τ. Přímou derivaci řetězce σ na řetězec τ zapisujeme $\sigma \Rightarrow \tau$.

Definice 1.9. Nechť je dána gramatika $G = (N, T, P, S)$, o řetězci τ řekneme, že je *derivací* řetězce σ, jestliže existuje konečný počet derivací $\sigma \Rightarrow \sigma_1, \sigma_1 \Rightarrow \sigma_2, \ldots, \sigma_{k-1} \Rightarrow \sigma_k$, kde $\sigma_k = \tau$ a $k \geq 1$. Derivaci označujeme $\sigma \Rightarrow^+ \tau$.

Často také píšeme $\sigma \Rightarrow^* \tau$, jestliže $\sigma \Rightarrow^+ \tau$ nebo $\sigma = \tau$.

Definice 1.10. Nechť je dána gramatika $G = (N, T, P, S)$, pak řetězec τ, jež je derivací startovního symbolu gramatiky S, tj. $S \Rightarrow^* \tau$ nazveme *větná forma* gramatiky, a řetězec jež je derivací startovního symbolu gramatiky S a obsahuje jen terminální symboly, tj. $S \Rightarrow^+ \tau$ a $\tau \in T^*$, nazveme *věta* gramatiky.

Pomocí výčtu vět gramatiky, definujeme formální jazyk, který gramatika generuje.

Definice 1.11. Nechť je dána gramatika $G = (N, T, P, S)$, pak množinu všech vět, jež lze v gramatice odvodit, tj. $L(G) = \{\tau |\ S \Rightarrow^+ \tau, \tau \in T^*\}$, nazveme *formální jazyk L generovaný gramatikou G*.

Noam Chomsky gramatiky rozdělil podle složitosti a tvaru gramatických pravidel na 4 typy. Jednotlivé typy se označují buďto čísly 0 až 3 nebo názvem typu.

Definice 1.12. Chomského klasifikace gramatik:

- *Gramatiky bez omezení* (typ 0) – tvar pravidel není nijak omezen.
- *Kontextově závislé gramatiky* (typ 1) – pravidla gramatiky $\alpha \to \beta$ musí splňovat vztah $|\alpha| \leq |\beta|$. Dále gramatika může obsahovat pravidlo $S \to \varepsilon$ za předpokladu, že startovní symbol S není na pravé straně žádného pravidla.
- *Bezkontextové gramatiky* (typ 2) – pravidla mají tvar $A \to \alpha$, kde $A \in N$, $\alpha \in (N \cup T)^*$.
- *Regulární gramatiky* (typ 3) – pravidla mohou mít tvar $A \to a$ nebo $A \to aB$, kde $A, B \in N,\ a \in T$.

Podle definic jednotlivých tříd gramatik vidíme, že každá kontextově závislá gramatika je zároveň gramatikou bez omezení a každá regulární gramatika je bezkontextovou gramatikou. Mezi třídami bezkontextových a kontextově závislých gramatik takový vztah neplatí. Lze ovšem dokázat, že ke každé bezkontextové gramatice lze sestrojit jinou bezkontextovou gramatiku, která je zároveň kontextově závislou gramatikou a přitom generuje stejný jazyk.

Definice 1.13. Třídy jazyků, generované gramatikami z příslušných tříd gramatik, se nazývají třídami *jazyků bez omezení* ($\mathcal{L}0$), *kontextově závislých jazyků* ($\mathcal{L}1$), *bezkontextových jazyků* ($\mathcal{L}2$) a *regulárních jazyků* ($\mathcal{L}3$).

Na základě uvedených definic a poznámek vyplývá následující vztah mezi třídami jazyků podle Chomského hierarchie gramatik:

$$\mathcal{L}0 \supseteq \mathcal{L}1 \supseteq \mathcal{L}2 \supseteq \mathcal{L}3.$$

Gramatiky jsou jen popisem jazyků. V praxi je ovšem potřeba s jazyky pracovat a řešit dva základní problémy:

- Určit, zda daný řetězec terminálních symbolů patří do jazyka generovaného gramatikou.
- Určit, jakým postupem, tj. jakou posloupností pravidel, byla věta jazyka odvozena.

Tyto problémy řeší tzv. automaty. Jelikož automaty řeší dané problémy algoritmicky, požadujeme, aby byly dostatečně efektivní (nejčastěji z hlediska časové složitosti). Pro gramatiky typu 0 a 1 vhodné automaty nejsou známy, proto se v praxi výlučně používají gramatiky typu 2 a 3, i když generují mnohem užší třídy jazyků. Tyto třídy jsou ovšem z dnešních hledisek dostačující.

Regulární jazyky

Regulární gramatiky jsou velmi jednoduché jazyky, jež je možné rozpoznávat pomocí poměrně jednoduchých, tzv. konečných automatů.

Definice 1.14. *Konečný deterministický automat* A je pětice (T, Q, δ, q_0, F), kde

- T je konečná množina vstupních symbolů,
- Q je konečná množina stavů,
- δ je přechodová funkce tvaru $\delta : Q \times T \to Q$,
- q_0 je počáteční stav automatu,

- F je množina koncových stavů, tedy $F \subseteq Q$.

Následují algoritmus popisuje činnost automatu, jehož úkolem je určit, zda předložený řetězec symbolů patří do množiny řetězců, které automat rozpoznává.

Algoritmus 1.1. (Činnost konečného deterministického automatu)

1. *Počáteční podmínky:*

   - *Automat se nachází v počátečním stavu q_0,*
   - *na vstupu automatu je řetězec $\omega = w_1 w_2 \cdots w_n \in T^*$, tzv. vstupní slovo.*

2. *Průběžný krok:*

   Automat se nachází ve stavu $q_i \in Q$ a na vstupu je doposud nezpracovaná část vstupního slova $w_k w_{k+1} \cdots w_n$.

   Automat odebere ze vstupu nejlevější znak w_k a nový stav automatu se určí pomocí přechodové funkce $\delta(q_i, w_k)$.

   - *Pokud přechodová funkce δ není pro argumenty q_i, w_k definována, činnost automatu končí a automat vstupní slovo ω nepřijal (nerozpoznal).*
   - *Jinak automat přejde do stavu $q_j = \delta(q_i, w_k)$. Je-li vstup prázdný, pak algoritmus pokračuje krokem 3, jinak krokem 2.*

3. *Koncový krok:*

   Automat je ve stavu q_i a celé vstupní slovo bylo zpracováno.

   - *Jestliže stav q_i je koncový, tj. $q_i \in F$, pak automat vstupní slovo rozpoznal.*
   - *Jestliže stav q_i není koncový, tj. $q_i \notin F$, pak automat vstupní slovo nerozpoznal.*

Definice 1.15. Vstupní slovo $\omega = w_1 w_2 \cdots w_n \in T^*$ je rozpoznáno konečným deterministickým automatem A, jestliže existuje posloupnost stavů $q_0, q_{i_1}, q_{i_2}, \cdots, q_{i_m}$ taková, že $q_{i_k} = \delta(q_{i_{k-1}}, w_k)$ a $q_{i_m} \in F$.

Definice 1.16. Množina slov $\omega \in T^*$, která jsou rozpoznávána konečným deterministickým automatem A, tvoří jazyk $L(A)$ rozpoznávaný automatem A.

Dále následují dvě tvrzení, která dávají do souvislosti regulární gramatiky a konečné deterministické automaty. Důkazy tvrzení jsou konstrukčního charakteru a je možno je najít v literatuře (např. [1]).

Věta 1.1. *Ke každé regulární gramatice G lze sestavit deterministický konečný automat A, který rozpoznává jazyk generovaný gramatikou G, tj. $L(A) = L(G)$.*

Věta 1.2. *Ke každému konečnému automatu A lze sestavit gramatiku G, která generuje jazyk rozpoznávaný automatem A, tj. $L(G) = L(A)$.*

Důkaz prvního tvrzení vlastně popisuje algoritmus sestavení konečného deterministického automatu z regulární gramatiky:

Algoritmus 1.2. (Konstrukce konečného deterministického automatu)
Vstupem algoritmu je regulární gramatika $G = (N, T, P, S)$ a výstupem konečný deterministický automat $A = (T, Q, \delta, q_0, F)$. Stavy automatu budou tvořeny podmnožinami množiny $N \cup \{f\}$, kde $f \notin N$ je nový symbol, který slouží jako příznak koncového stavu.

1. *Nejprve se nastaví počáteční stav $q_0 = \{S\}$, množina stavů obsahuje jen tento stav, tj. $Q = \{q_0\}$. Funkce δ není zatím nikde definována.*
2. *Pro každý stav $q \in Q$, který nebyl doposud zpracován a pro každý terminální symbol $t \in T$ se provede:*
   (a) *Vypočítá se nový stav $q' = \{U |\ X \in q, X \to tU \in P\} \cup \{f |\ X \in q, X \to t \in P\}$*
   (b) *Je-li $q \neq \emptyset$, pak se q' přidá do množiny stavů (pokud tam již není), tj. $Q = Q \cup \{q'\}$ a dále se definuje přechod $\delta(q, t) = q'$.*
3. *Koncové stavy automatu budou ty stavy, které obsahují symbol f, tj.*

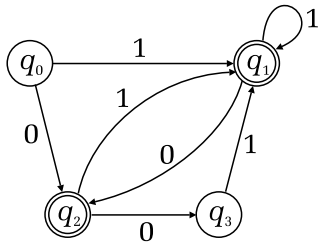
$$F = \{q |\ q \in Q,\ f \in q\}.$$

Konečné automaty se nejčastěji reprezentují pomocí orientovaných grafů. Stavy automatu odpovídají uzlům grafu a hrany představují přechody automatu. Tedy každá hodnota přechodové funkce $q_j = (q_i, x)$ je u deterministického konečného automatu zobrazena jako hrana z uzlu q_i do uzlu q_j ohodnocená symbolem x. Uzly odpovídající koncovým stavům automatu jsou zobrazeny zvýrazněně, například dvojitou čarou.

▷ Příklad 1.1. Nechť $A = (T, Q, \delta, q_0, F)$ je konečný deterministický automat, $T = \{0, 1\}$, $Q = \{q_0, q_1, q_2, q_3\}$, $F = \{q_1, q_2\}$ a přechodová funkce δ je definována takto:

$\delta(q_0, 0) = q_2 \quad \delta(q_0, 1) = q_1$
$\delta(q_1, 0) = q_2 \quad \delta(q_1, 1) = q_1$
$\delta(q_2, 0) = q_3 \quad \delta(q_2, 1) = q_1$
$\delta(q_3, 1) = q_1$

Grafem automatu pak je:



□

Regulární jazyky jsou z hlediska popisu a použití velmi jednoduché. Nicméně proto vymezují jen poměrně úzkou třídu jazyků. Ukázalo se, že i některé velmi jednoduché jazyky nejsou regulární. Například jazyk, jehož věty obsahují stejný počet nul a jedniček, není regulární. Z toho vyplývá, že regulární jazyky nepostačují pro popis běžných částí programovacích jazyků, jako jsou výrazy se závorkami, kdy výrazy musí mít stejný počet levých a pravých závorek. To podle uvedeného nelze popsat regulární gramatikou.

Bezkontextové jazyky

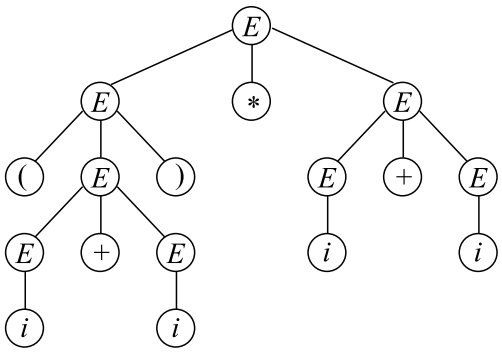
Pro popis programovacích jazyků (a dalších formálních jazyků, např. značkovacích (XML, HTML), jazyků pro zápis dokumentů (PostScript, TEX atd.)) se nejčastěji používají bezkontextové jazyky. Bezkontextové gramatiky totiž umožňují popis běžně používaných jazykových konstrukcí a jsou pro ně známy dostatečně efektivní automaty.

Definice 1.17. Nechť je dána gramatika $G = (N, T, P, S)$, věta ω jazyka $L(G)$, který gramatika generuje a nechť $\{\psi_i\}_{i=0}^{n}$ je posloupnost větných forem, jež odvozuje větu ω, tj. $\psi_0 = S$, $\psi_i \Rightarrow \psi_{i+1}$ pro $i = 0, \ldots, n-1$ je přímou derivací a $\psi_n = \omega$. Derivačním stromem věty ω v gramatice G nazveme takový strom, který splňuje následující:

- Kořenem stromu je uzel označený startovním symbolem gramatiky S, tedy jediným symbolem větné formy ψ_0.
- Vznikla-li větná forma ψ_{i+1} z větné formy ψ_i přepisem neterminálního symbolu podle některého pravidla gramatiky G, tedy $\psi_i = \mu A \nu$ a $\psi_{i+1} = \mu\alpha\nu$, $A \to \alpha \in P$, pak následníky uzlu označeného tímto neterminálním symbolem A jsou uzly označené symboly pravé strany α příslušného pravidla.

Z tohoto popisu vyplývá, že uzly stromu jsou označeny terminálními a neterminálními symboly gramatiky, listové uzly jsou vždy označeny terminálními a nelistové vždy neterminálními symboly.

▷ Příklad 1.2. Nechť $G = (N, T, P, E)$ je bezkontextová gramatika, která popisuje jednoduché aritmetické výrazy s operacemi násobení a sčítání. Množina $N = \{E\}$, $T = \{+, *, (,), i\}$, $P = \{E \to E + E,\ E \to E * E,\ E \to (E)\ , E \to i\}$. Terminální symbol i reprezentuje jednoduché operandy – čísla nebo proměnné. Derivačním stromem věty $(i + i) * i + i$ v gramatice G je např. strom:



□

Derivační strom je vhodnou reprezentací odvození věty jazyka v gramatice, neboť přehledně zobrazuje strukturu věty pomocí terminálních a neterminálních symbolů gramatiky. V případě, že příslušný automat rozpozná větu jazyka, jeho výstupem bývá i derivační strom této věty.

Bezkontextové jazyky jsou rozpoznávány tzv. zásobníkovými automaty. Podobně jako u konečných automatů existují deterministické i nedeterministické zásobníkové automaty, podle toho, jestli je následující krok činnosti automatu určen jednoznačně nebo ne.

Definice 1.18. *Nedeterministický zásobníkový automat* je sedmice

$$A = (T, Q, Z, \delta, q_0, z_0, F), \text{ kde}$$

- T je konečná množina vstupních symbolů,
- Q je konečná množina stavů,
- Z je konečná množina zásobníkových symbolů,
- δ je přechodová funkce, $\delta : Q \times (T \cup \{\varepsilon\}) \times Z^* \to 2^{Q \times Z^*}$, jednotlivé argumenty představují

  Q – aktuální stav automatu,

  T – odebraný symbol ze vstupního řetězce,

  Z^* – řetězec zásobníkových symbolů odebíraný ze zásobníku,

  $2^{Q \times Z^*}$ – Q je stav, do kterého automat v daném kroku přejde, Z^* je řetězec zásobníkových symbolů, který je v daném kroku přidán na zásobník.
- q_0 je počáteční stav, $q_0 \in Q$,
- z_0 je počáteční symbol na zásobníku, $z_0 \in Z$.

Následující algoritmus popisuje činnost nedeterministického zásobníkového automatu.

Algoritmus 1.3. (Činnost nedeterministického zásobníkového automatu)

1. *Počáteční podmínky:*

   Automat se nachází ve stavu q_0, na zásobníku je jediný symbol z_0, na vstupu je nezpracované slovo $\omega = w_1 w_2 \cdots w_n \in T^*$.

2. *Průběžný krok:*

   Aktuální stav automatu je q_i, na zásobníku je m zásobníkových symbolů $z_{i_0}, z_{i_1}, \ldots, z_{i_m}$, na vstupu je dosud nezpracovaná část vstupního slova $w_k w_{k+1} \cdots w_n$. Přechod (q_s, μ) k provedení je vybrán z množiny $\bigcup \delta(q_i, x, \chi)$, kde x je w_k nebo ε a $\chi \in Z^*$, tedy ze vstupu může být odebrán symbol w_k a ze zásobníku je odebrán řetězec zásobníkových symbolů χ a následně automat přejde do nového stavu q_s a do zásobníku jsou vloženy zásobníkové symboly z řetězce χ.

3. *Koncový krok:*

   Automat rozpozná vstupní slovo, jestliže ho celé zpracuje (vstup je na konci prázdný) a skončí v koncovém stavu.

Uvedená varianta nedeterministického zásobníkového automatu pracuje s celým obsahem zásobníku. Vedle toho existuje i varianta, která při každém kroku ze zásobníku odebírá právě jeden zásobníkový symbol. Přechodová funkce má v tomto případě tvar

- δ je přechodová funkce $Q \times (T \cup \{\varepsilon\}) \times Z \to 2^{Q \times Z^*}$.

Další používanou variantou zásobníkových automatů je automat rozpoznávající vstupní větu vyprázdněním zásobníku. Algoritmus činnosti se v tomto případě liší od výše uvedeného v bodu 3:

3'. Automat rozpozná vstupní slovo, jestliže ho celé zpracuje (vstup je na konci prázdný) a zásobník je prázdný.

Lze dokázat, že všechny uvedené varianty nedeterministických zásobníkových automatů rozpoznávají stejnou třídu jazyků. Důkazy těchto tvrzení lze nalézt v uvedené literatuře.

Následující věty ukazují vztah mezi bezkontextovými gramatikami a zásobníkovými automaty. Důkazy těchto vět jsou konstrukčního charakteru a opět je možné je nalézt v uvedené literatuře.

Věta 1.3. *Ke každé bezkontextové gramatice G lze sestrojit nedeterministický zásobníkový automat A, který rozpoznává jazyk generovaný gramatikou G, tj. $L(A) = L(G)$.*

Věta 1.4. *Ke každému nedeterministickému zásobníkovému automatu A lze sestavit gramatiku G, která generuje jazyk rozpoznávaný automatem A, tj. $L(G) = L(A)$.*

Důkaz prvního tvrzení zároveň dává návod na sestavení automatů z gramatik. Důkaz je založen na konstruování přechodové funkce automatu pomocí pravidel gramatiky tak, aby při činnosti automatu byl prováděn průchod derivačním stromem předloženého řetězce. Podaří-li se vytvořit korektní derivační strom a řetězec je celý zpracován, automat slovo rozpoznal. Průchod stromem je možný shora dolů nebo zdola nahoru. Oba způsoby je možno implementovat s využitím zásobníku.

V obou případech se používá varianta, jež rozpoznává větu pomocí vyprázdnění zásobníku. Navíc nijak nejsou využívány stavy automatu, tudíž automat se nachází stále v jednom stavu. Zásobníkovými symboly jsou terminální a neterminální symboly gramatiky G, počátečním zásobníkovým symbolem je startovní symbol S gramatiky. Automat je z pohledu gramatiky definován takto: $A = (T, \{q\}, T \cup N, \delta, q, S, \emptyset)$. Přechodová funkce δ je definována následovně:

- Pro průchod shora dolů: Přechody (operace) jsou dvou typů:
  - Expanze – je proveditelná, pokud je na vrcholu zásobníku neterminální symbol. Pro každý neterminální symbol $A \in N$ a pro všech m pravidel tvaru $A \to \alpha_i$, $i \in \{1, \cdots, m\}$ přechodová funkce obsahuje přechod $\delta(q, \varepsilon, A) = \{(q, \alpha_1), (q, \alpha_2), \ldots, (q, \alpha_m)\}$.
  - Srovnání – je proveditelná pokud je na vrcholu zásobníku terminální symbol. Pro každý terminální symbol $t \in T$ přechodová funkce obsahuje přechod $\delta(q, t, t) = \{(q, \varepsilon)\}$.

  Věta je rozpoznána, je-li celá zpracována a zásobník je na konci prázdný.

- Pro průchod zdola nahoru: Přechody (operace) jsou opět dvou typů:
  - Přesun – operace přesune symbol ze vstupu do zásobníku, tedy pro každý terminální symbol $t \in T$ přechodová funkce obsahuje přechod $\delta(q, t, \varepsilon) = \{(q, t)\}$
  - Redukce – operace nahradí v zásobníku řetězec symbolů, jež odpovídá pravé straně některého pravidla, jeho levou stranou. Nechť $\alpha_1, \ldots, \alpha_m$ jsou řetězce symbolů, jež odpovídají alespoň jedné pravé straně některého pravidla gramatiky G. Pro každou pravou stranu α_i přechodová funkce obsahuje přechod $\delta(q, \varepsilon, \alpha_i) = \{(q, A_1), \ldots, (q, A_k)\}$, kde $A_1 \to \alpha_i, \ldots, A_k \to \alpha_i$ jsou všechna pravidla gramatiky G s pravou stranou α_i.

Počáteční a koncová situace zásobníku se poněkud liší od dříve popsaných variant automatů:

- Na počátku je zásobník prázdný
- Věta je rozpoznána, pokud je vstup celý zpracován a na zásobníku je pouze startovní symbol S gramatiky G.

U konečných automatů platí, že pro každý nedeterministický automat lze sestavit deterministický konečný automat, jenž rozpoznává stejný jazyk. Obdobné tvrzení se u zásobníkových automatů nepodařilo prokázat, protože existují bezkontextové gramatiky, pro něž deterministický automat nebyl sestaven. Konstrukce deterministických zásobníkových automatů je totiž známa jen pro určité podtřídy bezkontextových gramatik.

Při konstrukci automatů se používají tzv. množiny First_k a Follow_k. Téměř výhradně se používají automaty, při jejichž konstrukci se používají jen množiny First_1 a Follow_1, jež se také označují jako First a Follow.

Definice 1.19. Nechť je dána gramatika $G = (N, T, P, S)$. Množina First je definována předpisem

$$\text{First}(\alpha) = \{t \mid \alpha \Rightarrow^* t\mu,\ t \in T\} \cup \{\varepsilon \mid \alpha \Rightarrow^* \varepsilon\},$$

kde $\alpha \in (N \cup T)^+$.

Argumentem množiny First je řetězec symbolů α, výsledná množina obsahuje terminální symboly, které se vyskytují na počátcích větných forem, které lze získat derivací řetězce α, nebo případně obsahuje prázdný řetězec ε, pokud derivací řetězce α lze získat prázdný řetězec.

Definice 1.20. Nechť je dána gramatika $G = (N, T, P, S)$. Množina Follow je definována předpisem

$$\text{Follow}(A) = \{t \mid S \Rightarrow^* \mu A \nu,\ \nu \neq \varepsilon,\ t \in \text{First}(\nu)\} \cup \{\varepsilon \mid S \Rightarrow^* \mu A\},$$

kde $A \in N$, $\mu, \nu \in (N \cup T)^*$.

Argumentem množiny Follow je neterminální symbol A, výsledná množina obsahuje terminální symboly, jež se mohou vyskytnout v některé větné formě za symbolem A, případně obsahuje prázdný řetězec ε, pokud se symbol A může vyskytnout v některé větné formě na konci.

Množiny First a Follow lze vypočítat následujícími algoritmy.

Algoritmus 1.4. (Výpočet množiny First(α)**)**
Vstupem algoritmu je gramatika $G = (N, T, P, S)$ a řetězec $\alpha \in (N \cup T)^*$. Algoritmus používá pomocnou množinu F, v níž jsou ukládány řetězce nebo pravidla s vyznačením aktuální pozice pomocí symbolu „.“ (tečka).

1. *Množina F obsahuje pouze řetězec $\alpha = a_1 a_2 \cdots a_m$ s tečkou na začátku, tj. $F = \{\ .a_1 a_2 \cdots a_m\ \}$*
2. *Dokud se množina F mění, pak:*
   (a) *Pro každý neterminální symbol A, pro každý prvek množiny F, ve kterém je bezprostředně za tečkou neterminální symbol A, se přidají do množiny F všechna pravidla z P se symbolem A na levé straně s označením pozice na začátku pravé strany, tj.*

$$F = F \cup \{A \to .\beta \mid B \to \mu.A\nu \in F,\ A \to \beta \in P\}.$$

(b) *Pro každé pravidlo* $B \to \gamma. \in F$ *s označením pozice na konci pravé strany, pro každé pravidlo* $A \to \mu.B\nu \in F$ *s označením pozice před* B *se do* F *přidá nové pravidlo* $A \to \mu B.\nu$ *s označením pozice za* B*, tj.*

$$F = F \cup \{A \to \mu B.\nu \mid B \to \gamma. \in F,\ A \to \mu.B\nu \in F\}.$$

3. *Množina* $\mathrm{First}(\alpha)$ *je tvořena terminálními symboly* t*, jež se vyskytují v některém pravidle z* F *za označením pozice tečkou, případně obsahuje prázdný řetězec* ε*, pokud má některý prvek z* F *označení tečkou až na konci, tj.*

$$\mathrm{First}(\alpha) = \{t \mid B \to \mu.t\nu \in F,\ t \in T\} \cup \{\varepsilon \mid B \to \gamma. \in F\}.$$

Algoritmus 1.5. (Výpočet množiny $\mathrm{Follow}(A)$**)**
Vstupem algoritmu je gramatika $G = (N, T, P, S)$ *a neterminální symbol* $A \in N$*. Algoritmus používá pomocnou množinu* H*, v níž uchovává již zpracované neterminální symboly kvůli zamezení opakování stejného výpočtu v důsledku rekurze. Dále* Y *označuje aktuálně zpracovávaný neterminální symbol.*

1. *Množina* H *je prázdná,* $Y = A$*.*
2. *Pokud byl symbol* Y *již zpracován* ($Y \in H$)*, výpočet pro aktuální* Y *končí.*
3. *Pokud symbol* Y *ještě nebyl zpracován:*
   - Y *je vloženo do* H*, tj.* $H = H \cup \{Y\}$*,*
   - $\mathrm{Follow}(Y)$ *je ze začátku prázdná,*
   - *je-li* $Y = S$*, do* $\mathrm{Follow}(Y)$ *je přidán prázdný řetězec, tj.*

$$\mathrm{Follow}(Y) = \mathrm{Follow}(Y) \cup \{\varepsilon\}.$$

4. *Pro všechna pravidla tvaru* $X \to \mu Y\nu \in P$*:*
   - *Je-li* $\nu \neq \varepsilon$*, položí se* $G = \mathrm{First}(\nu)$ *a* $\mathrm{Follow}(Y) = \mathrm{Follow}(Y) \cup (G \setminus \{\varepsilon\})$*, je-li* $\varepsilon \in G$*, pak se rekurzivně od kroku 2 vypočítá množina* Follow *pro* $Y = X$ *a její prvky se přidají se do* $\mathrm{Follow}(Y)$*, tj.*

$$\mathrm{Follow}(Y) = \mathrm{Follow}(Y) \cup \mathrm{Follow}(Y/X).$$

   - *Pokud* $\nu = \varepsilon$*, pouze se rekurzivně od kroku 2 vypočítá množina* Follow *pro* $Y = X$ *a její prvky se přidají do* $\mathrm{Follow}(Y)$*, tj.*

$$\mathrm{Follow}(Y) = \mathrm{Follow}(Y) \cup \mathrm{Follow}(Y/X).$$

Gramatiky $LL(k)$

$LL(k)$ je třídou bezkontextových gramatik, jež umožňují konstrukci deterministických zásobníkových automatů pracujících na principu průchodu derivačním stromem směrem shora dolů.

Označení $LL(k)$ značí, že při analýze věty automatem se postupuje zleva doprava, pro průchod derivačním stromem se používá jen derivace nejlevějšího neterminálního symbolu větné formy a pro deterministický chod automatu je potřeba znát nejméně k symbolů ze vstupu.

Pomocí množin First_k a Follow_k se definují $LL(k)$ gramatiky.

Definice 1.21. Bezkontextová gramatika je $LL(k)$ gramatikou, jestliže každá dvě její pravidla se stejným symbolem na levé straně $A \to \alpha$ a $A \to \beta$ splňují následující podmínku:

$$\bigcap_{\omega \in \mathrm{Follow}_k(A)} (\mathrm{First}_k(\alpha\omega) \cap \mathrm{First}_k(\beta\omega)) = \emptyset.$$

Zkráceně zapisujeme

$$\mathrm{First}_k(\alpha \mathrm{Follow}_k(A)) \cap \mathrm{First}_k(\beta \mathrm{Follow}_k(A)) = \emptyset.$$

Množina $\mathrm{First}_k(\alpha \mathrm{Follow}_k(A))$ se nazývá množina First − Follow pravidla $A \to \alpha$.

Podmínka pro $LL(1)$ gramatiky zajišťuje determiničnost sestaveného automatu – při operaci expanze neterminálního symbolu A je prvními k symboly na vstupu jednoznačně určeno, podle kterého pravidla bude expanze provedena.

Gramatiky $LR(k)$

$LR(k)$ je třídou bezkontextových gramatik, jež umožňují konstrukci deterministických zásobníkových automatů pracujících na principu průchodu derivačním stromem směrem zdola nahoru.

Označení $LR(k)$ značí, že při analýze věty automatem se postupuje zleva doprava, pro průchod derivačním stromem se používá jen derivace nejpravějšího neterminálního symbolu větné formy a pro deterministický chod automatu je potřeba znát nejméně k symbolů ze vstupu.

Téměř vždy se v praxi používají $LR(1)$ gramatiky, neboť platí, že pro každou $LR(k)$ gramatiku, kde $k > 1$ existuje $LR(1)$ gramatika, jež generuje stejný jazyk.

Při konstrukci deterministických automatů pro analýzu zdola nahoru je potřeba vyřešit několik technických problémů. Na rozdíl od analýzy shora dolů nelze jednoznačně rozhodnout podle stavu zásobníku, na němž jsou jen terminální a neterminální symboly, zda provést přesun nebo redukci. Dalším problémem je rozhodnutí o náhradě řetězce zásobníkových symbolů levou stranou některého pravidla, neboť je potřeba zásobník prohledávat do hloubky.

Tyto problémy se řeší zavedením speciálních zásobníkových symbolů, jež obsahují informace o fázi přesunu pravých stran některých pravidel a automat pracuje jen s vrcholem zásobníku. Je-li přesun dokončen, může dojít k redukci. Dále v sobě zásobníkové symboly nesou informace o tom, při kterých symbolech na vstupu proběhne redukce a při ostatních přesun.

Dále je potřeba zajistit, aby rozpoznání řetězce automatem, coby věty jazyka, bylo dobře zjistitelné. Proto se často při konstrukci automatu používá tzv. rozšířená gramatika. Tato má pro startovní symbol jediné pravidlo ve tvaru $S \to A$, $A \in N$, čímž je velice snadné určit, kdy je analýza ukončena, neboť jde o analýzu zdola nahoru a toto pravidlo je v derivačním stromu zcela nahoře a tudíž reprezentuje závěrečnou redukci při analýze.

Doplnit existující gramatiku na rozšířenou je snadné: K dané gramatice $G = (N, T, P, S)$ se přidá nový startovní symbol S' a pravidlo $S' \to S$, tedy vznikne gramatika $G' = (N \cup \{S'\}, T, P \cup \{S' \to S\}, S')$.

Klíčovou úlohou při konstrukci automatu z gramatiky je tedy zavedení vhodných zásobníkových symbolů.

Samotné gramatiky $LR(1)$ na jedné straně pokrývají značně rozsáhlou třídu jazyků, na druhé straně jejich automaty jsou již poměrně složité, neboť obsahují značný počet zásobníkových symbolů. Vedle samotných gramatik $LR(1)$ existují ještě další tři typy gramatiky pro deterministickou analýzu směrem zdola nahoru. Jsou to gramatiky $LALR(1)$, $SLR(1)$ a $LR(0)$. Pokrývají sice užší třídy jazyků, ale na druhé straně konstrukce automatu je pro ně jednodušší. Platí zde, že čím je gramatika obecnější, tím složitější je konstrukce automatu. Konstrukce automatů je u všech typů $LR(1)$ gramatik v podstatě stejná, liší se v některých případech strukturou zásobníkových symbolů a způsobem určení redukcí.

V následující části je uvedena konstrukce automatu pro gramatiku $LALR(1)$, jenž je v praxi nejpoužívanější, neboť je prováděno slučování zásobníkových symbolů oproti automatu $LR(1)$ na počet jako u automatů $LR(0)$ a $SLR(1)$, avšak jen nepatrně zužuje třídu rozpoznávaných jazyků.

Konstrukce zásobníkového automatu pro $LALR(1)$ gramatiku

Definice 1.22. Nechť $G = (N, T, P, S)$ je rozšířená gramatika. Výrazy tvaru $[A \to \alpha.\beta; \omega]$, kde $A \to \alpha\beta \in P$, $\omega \subseteq T \cup \{\varepsilon\}$ nazveme *položkou pravidla* gramatiky G.

- První část se nazývá *jádro položky.*
- Druhá část je *množina prediktivních symbolů.*

Neprázdná množina položek pravidel gramatiky se nazývá *zásobníkový symbol.* Jádrem zásobníkového symbolu je sjednocení jader jeho položek, podobně u prediktivní množiny.

Zásobníkové symboly se skládají z položek pravidel gramatiky. Každá z nich reprezentuje odvozovací pravidlo s vyznačením fáze přesunu. Zásobníkový symbol sestává z více položek právě proto, že může reprezentovat několik přesunů zároveň. Dále prediktivní množiny symbolů určují, které symboly mohou být na vstupu pro dokončení redukce.

Je-li jádro některé položky ve tvaru $A \to \alpha.B\beta$, kde B je neterminální symbol, pak reprezentuje situaci, že do zásobníku má být vložen neterminální symbol B. Ten se do zásobníku může dostat jedině redukcí. To znamená, že nejprve pravá strana některého pravidla se symbolem B na levé straně musí být přesunuta do zásobníku a poté redukována. Z tohoto důvodu je potřeba zásobníkový symbol obohatit o položky pravidel s levou stranou B v počáteční fázi přesunu pomocí operace uzávěr.

Definice 1.23. Nechť $G = (N, T, P, S)$ je rozšířená gramatika a M je množina položek pravidel gramatiky. Dále nechť $\{U_i\}_{i=0}^{\infty}$ je posloupnost množin, taková, že:

1. $U_0 = M$
2. $U_{i+1} = U_i \cup \{[B \to .\beta; \kappa(\gamma, \omega)] \mid [A \to \alpha.B\gamma; \omega] \in U_i, B \to \beta \in P\}$, kde nová množina prediktivních symbolů $\kappa(\gamma, \omega)$ je

$$\kappa(\gamma, \omega) = \bigcup_{\forall t \in \omega} \mathrm{First}(\gamma t).$$

Množina U_j taková, že $U_j = U_{j+1}$, se nazývá *uzávěr* množiny M a značí se $U(M)$. Dle Dirichletova principu takové j existuje a tedy uzávěr $U(M)$ lze nalézt.

Nyní již vlastní postup konstrukce automatu.

Algoritmus 1.6. (Konstrukce automatu pro gramatiku $LALR(1)$)
Vstupem je rozšířená gramatika $G = (N', T, P', S')$. Výstupem bude zásobníkový automat s množinou zásobníkových symbolů Z a tabulkou automatu. Operace přesunu jsou v průběhu algoritmu zaznamenávány pomocí funkce goto : $Z \times N \cup T \to Z$.

1. *Z pravidla se startovním symbolem S' na levé straně se zkonstruuje počáteční zásobníkový symbol $z_0 = U(\{[S' \to .S; \{\varepsilon\}]\})$, $Z = \{z_0\}$.*

2. *Pro každý doposud neuvažovaný zásobníkový symbol $z \in Z$ a pro každý symbol $a \in N \cup T$ se provedou následující kroky:*

   (a) *Sestaví se nový zásobníkový symbol*

   $$z_a = U(\{[A \to \alpha a.\beta; \omega] |\ [A \to \alpha.a\beta; \omega] \in z\}),$$

   který reprezentuje zásobníkový symbol vzniklý přesunem symbolu a do zásobníku při symbolu z na vrcholu zásobníku.

   (b) *Pokud je $z_a = \emptyset$, pak algoritmus pokračuje krokem 2.*

   (c) *Pokud z_a dosud není v množině Z a jádro z_a je různé od jader všech ostatních symbolů v Z, pak $Z = Z \cup \{z_a\}$, $goto(z, a) = z_a$ a algoritmus pokračuje krokem 2.*

   (d) *Pokud z_a dosud není v množině Z, ale jádro z_a se shoduje s jádrem zásobníkového symbolu z', pak prediktivní množiny položek pravidel v zásobníkovém symbolu z' jsou obohaceny o symboly z prediktivní množiny příslušné položky gramatiky v zásobníkovém symbolu z_a, tj.*

   $$\forall\ [A \to \alpha.\beta; \omega] \in z' :\ \omega = \omega \cup \psi,\ [A \to \alpha.\beta; \psi] \in z_a.$$

   Protože se změnily prediktivní množiny v symbolu z', je potřeba upravit prediktivní množiny v těch zásobníkových symbolech, jež vznikly použitím položek pravidel ze zásobníkového symbolu z' a to i rekurzivně. Vyjde se od symbolů z'', jež přímo vznikly použitím položek ze symbolu z', tj. $goto(z', x) = z''$ a pokračuje se rekurzivně dále. U položek ze z'', jež vznikly pouze posunem pozice z položky symbolu z' (v kroku 2. (a) tohoto algoritmu) je prediktivní množina nahrazena prediktivní množinou výchozí položky, u položek ze z'', jež vznikly uzávěrem, je množina prediktivních symbolů přepočítána obdobně jako ve funkci uzávěru.
   Dále se definuje $goto(z, a) = z'$ a algoritmus pokračuje krokem 2.

3. *Po ukončení kroku 2. je možno sestavit tabulku automatu. Její řádky jsou označeny zásobníkovými symboly, sloupce odpovídají terminálním symbolům, prázdnému řetězci a neterminálním symbolům. Každý přesun $goto(z_j, x) = z_k$ je do tabulky zaznamenán na pozici v řádku odpovídajícímu symbolu z_j a sloupci odpovídajícímu symbolu x vložením zásobníkového symbolu z_k.*

4. *Nyní je možno do tabulky zaznamenat operace redukce. Pro každý zásobníkový symbol z_k a každou jeho redukční položku $[A \to \alpha.; \omega]$ se pro každý symbol nebo prázdný řetězec x z ω na pozice v tabulce na řádku zásobníkového symbolu z_k a sloupci prvku x vloží pravidlo $A \to \alpha$.*

5. *Na pozici v tabulce na řádku zásobníkového symbolu, který obsahuje redukční položku* $[S' \to S.;\varepsilon]$*, a sloupci prázdného řetězce* ε *se vloží symbol „stop“, jenž značí provedení poslední redukce a ukončení činnosti automatu.*

Objekty na jednotlivých pozicích v tabulce automatu se obvykle kódují pomocí celých čísel. Prázdné pozice např. číslem 0, pozice se zásobníkovými symboly jejich kladnými indexy v množině Z a redukční pravidla jejich čísly v rámci množiny P. Čísla redukčních pravidel bývají odlišena od čísel zásobníkových symbolů, například tím, že jsou opačná (tedy záporná). Symbol zastavení automatu „stop“ se může kódovat např. číslem mimo rozsah jak čísel zásobníkových symbolů, tak pravidel.

Nyní zbývá popsat činnost samotného automatu. Ta vychází z již popsané činnosti obecného nedeterministického automatu pro analýzu zdola nahoru. Ovšem nedeterminističnost je odstraněna záznamy v tabulce automatu.

Algoritmus 1.7. (Činnost automatu pro $LALR(1)$ **gramatiky)**

1. *Počáteční podmínky:*

   - *V zásobníku je vložen zásobníkový symbol* z_0*.*
   - *Na vstupu je řetězec terminálních symbolů, jež má automat analyzovat.*

2. *Průběžný krok:*

   Na vrcholu zásobníku je zásobníkový symbol z_i *a první symbol na vstupu je* x *nebo prázdný řetězec* ε*.*

   V tabulce se vyhledá pozice na řádku odpovídající symbolu na vrcholu zásobníku a v sloupci odpovídající symbolu na vstupu x*. Mohou nastat 4 případy:*

   - *Na dané pozici v tabulce je zásobníkový symbol* z_k*. Značí to, že následující operací je přesun:*
     - *Symbol* x *odebereme ze vstupu.*
     - *Na vrchol zásobníku přidáme zásobníkový symbol* z_k*.*
     - *Automat pokračuje krokem 2.*
   - *Na dané pozici je redukční pravidlo* $A \to x_1 x_2 \cdots x_m$*, což značí, že následující operací je redukce:*
     - *Ze zásobníku se odebere* m *zásobníkových symbolů.*
     - *Následně se na vrchol přidá zásobníkový symbol odpovídající symbolu* A *na levé straně pravidla. Ten je v tabulce na řádku, jenž přísluší současnému symbolu na vrcholu zásobníku a v sloupci, jenž odpovídá neterminálnímu symbolu* A*.*
     - *Automat pokračuje krokem 2.*
   - *Daná pozice v tabulce je prázdná. Tím činnost automatu končí s tím, že řetězec na vstupu není větou jazyka.*
   - *Na dané pozici je symbol zastavení automatu „stop“. Jestliže je vstup prázdný, automat rozpoznal větu jazyka, jinak (pokud na vstupu nějaká část vstupního řetězce zbyla) vstupní řetězec není větou daného jazyka. Činnost automatu v obou případech končí.*

Pokud se vyskytne při sestavování automatu v tabulce na jedné pozici více objektů, dochází ke konfliktu a automat není deterministický. Znamená to, že gramatika není LALR(1). V automatu se mohou vyskytovat dva typy konfliktů.

- Konflikt přesun-redukce: Ten nastává, je-li na některé pozici v tabulce automatu zároveň přesun a jedna nebo více redukcí. Tento konflikt je nejčastěji způsoben tím, že gramatika je nejednoznačná, tedy pro některé věty jazyka existuje více derivačních stromů. Naštěstí tyto konflikty lze poměrně snadno upravit transformací pravidel gramatiky, nebo rozhodnout o tom, která operace bude provedena „ručně“. Tedy například pomocí priority a asociativity operátorů v jazyce.
- Konflikt redukce-redukce. Při něm jsou na některé pozici v tabulce automatu dvě nebo více redukcí. Tyto konflikty by v dobře napsané gramatice neměly nastávat. Tyto konflikty je potřeba řešit úpravou pravidel gramatiky.

1.2. Objektově orientovaný přístup

Objektově orientovaný přístup (OOP) k tvorbě softwaru je jedním a v současné době nejpoužívanějším přístupem k tvorbě softwaru. Jeho hlavním principem při tvorbě softwarového systému je dekompozice systému na jakési moduly, tzv. objekty, jež zároveň obsahují data i funkcionalitu.

Zastaralý strukturální přístup

Dřívější přístupy tvorby systému, například strukturální přístup, se zdají z dnešního pohledu být již zastaralé. Ve strukturálním přístupu je softwarový systém také rozdělen na moduly, ovšem dvou druhů - funkce a data.

Data – daty se rozumí veškeré proměnné základních typů – celé číslo, znak, řetězec, číslo v plovoucí čárce, dále pak strukturovaná data – pole, záznamy atd. Základní vlastností dat je, že obsahují informace.

Funkce – funkcemi rozumíme všechny prvky výkonného charakteru. Funkce jsou identifikovatelné posloupnosti činností a jejich charakteristikou je, že mění obsah dat. Funkce pracují s daty dvěma směry, buď informace z dat čtou nebo informace do dat zapisují.

Důležité je, že toto dělení je úplné, tedy veškerá data i funkce se chovají globálně. Při návrhu softwarového systému strukturálním přístupem je nejprve celý systém modelován jedinou funkcí. Dále jsou činnosti této funkce rozdělovány do menších funkcí, jež jsou vykonávány v rámci nadřazené funkce atd. Data existují všechna „na jedné hromadě“ a každá funkce ví, se kterými daty má pracovat.

Tento přístup má jeden nepříjemný důsledek - realita, jež je softwarovým systémem modelována, je sama o sobě objektově orientovaná a to vede k nutnosti převádět reálné objekty na zcela jiné a oddělené prvky, kterými jsou data a funkce.

Tímto převodem se ztrácejí přirozené vlastnosti reálných (tedy objektových) systémů, jako je zodpovědnost objektů za provádění činností, nebo zapouzdřenost a skrývání informací. Tím se zdají být systémy tvořené pomocí strukturálního přístupu méně přehledné, méně čitelné, a tím i méně rozšiřitelné a znovupoužitelné, což jsou základní požadavky na kvalitní softwarový systém.

Objektově orientovaný přístup se tedy snaží reálné objekty se všemi jejich aspekty modelovat pomocí softwarových objektů, jež mají mnohem více společných charakteristik s reálnými objekty než s daty a funkcemi.

Objekt a třída

Základní vlastností všech prostředí (jazyk, technologie atd.), která podporují objektově orientovaný přístup, je možnost vytvářet objekty a dále s nimi pracovat.

Objektem v objektově orientovaném přístupu rozumíme v softwarovém systému uzavřenou strukturu, která má tyto vlastnosti:

1. Obsahuje *vnitřní paměť*, ve které uchovává svůj vnitřní stav. Tato paměť se skládá z tzv. *atributů*. Důležité je (narozdíl od dat v strukturálním přístupu), že tato vnitřní paměť není přímo z vnějšku objektu přístupná a je čistě soukromou záležitostí objektu.

2. Obsahuje *metody* objektu, což jsou funkce nebo procedury, obecně posloupností činností, které vykonávají činnost nad vnitřní pamětí objektu a jen nad ní (samozřejmě s využitím argumentů metod, jež pochází z vnějšku objektu). Metody jsou opět z vnějšku objektu naprosto neviditelné a nelze je volat přímo. Vnitřní metody a paměť jsou v rámci téhož objektu plně viditelné. Metody jsou jediným prostředkem jak manipulovat s vnitřními daty objektu.

3. Obsahuje *mechanismus* pro příjem a zpracování tzv. *zpráv* z vnějšku. Součástí mechanismu je tzv. *protokol zpráv*, což je v podstatě převodník zpráva → metoda objektu. V protokolu je vždy k jedné zprávě přiřazena jediná metoda. Zpracování zprávy probíhá tak, že objekt nalezne v protokolu zpráv přijatou zprávu a k ní přiřazenou metodu a tuto metodu zavolá s parametry (argumenty) obsaženými ve zprávě. Po vykonání metody předá zprávě výstupní parametry metody. Vstupními a výstupními parametry zpráv jsou opět objekty. Jedinou možností, jak spolupracovat s objektem z vnějšku, je poslat mu zprávu. Stav a chování objektu tedy můžeme pozorovat jedině sledováním, jak objekt reaguje na zprávy.

4. Objekt může obsahovat *další objekty*, jimž může v průběhu vykonávání svých metod zasílat zprávy a tak řídit jejich činnost. Tímto vznikají sekvence zasílání zpráv od objektu k objektu, které tvoří tok činnosti programu.

Tyto vlastnosti objektů jsou axiomatické a základní. Ostatní vlastnosti a pojmy, jako například dědičnost nebo polymorfismus (viz dále) jsou odvoditelné z těchto základních vlastností.

V některých programovacích jazycích, které jsou objektově orientované, se s objekty pracuje poněkud odlišně. Především tyto jazyky používají konstrukce jako „volání metod objektu“ z jeho vnějšku a např. dovolují (i když to z hlediska návrhu není správné) přistupovat k atributům (fields) objektu z vnějšku. Navíc se zde nijak (explicitně) nevyskytuje protokol zpráv. To je způsobeno především syntaxí jazyků a způsobem provádění kódu, které jsou převzaty z procedurálních jazyků, neboť jsou dostatečně efektivní i ověřené časem. Přímé volání metod nebo přístup k atributům objektů jsou jen zkratkami pro zasílání zpráv různých typů.

V definici objektu se nikde nevyskytuje známý pojem třída a přitom bylo uvedeno, že daná definice je plně postačující. Objekt totiž může vzniknout i bez třídy. Pokud vzniká objekt, je potřeba deklarovat pouze jeho atributy, metody, protokol zpráv a z jakých dalších objektů je

složen. Objekt může korektně existovat i bez třídy. Tento způsob definice objektů „jeden po druhém“ je možný, má však jednu nevýhodu – při definici dalšího objektu naprosto stejných vlastností vzniká redundance.

Z tohoto důvodu, aby se definice nemusela opakovat, se zavádí nový objekt – *třída*. Je to takový objekt, který napomáhá vzniknout, podobně jako forma nebo šablona, novým objektům stejných vlastností. Třída obvykle reaguje na zprávu typu „NewObject“ a výstupním parametrem této zprávy je nový objekt.

Některé objektově orientované jazyky opět z historických důvodů zavádějí třídu jinak než OOP. Pojem třída je zde chápán jako typ proměnné. Smysl „forma na objekty“ je však zachován.

Objekty jež vznikly pomocí některé třídy se nazývají *instance* této třídy a třída se chová jako skupina „ekvivalentních“ objektů, tedy objektů se stejnými vlastnostmi, tedy stejně jako pojem třída v algebře.

Zavedením tříd v OOP vzniká nový druh abstrakce, neboť každá třída definuje vlastnosti pro všechny svoje instance, které již existují, nebo které mohou teprve vzniknout. Třída sdílí definice svých instancí, nikoliv však jejich informační obsah.

Vztahy mezi objekty

V definici objektu je uvedeno, že objekt může obsahovat další objekty a jim zasílat zprávy. Jiným způsobem by objekty mezi sebou nemohly komunikovat. Termínem „obsahovat“ ovšem není myšleno, že objekt obsahuje další objekty podobně jako velká krabice může obsahovat menší, ty ještě menší atd., neboť všechny objekty existují fyzicky vždy samostatně. Toto obsahování je myšleno spíše logicky a nejčastěji bývá realizováno pomocí *reference na objekt*. Referencí je v podstatě myšlena jakási cesta, kudy posílat zprávy obsaženému objektu.

Některé objektově orientované jazyky nerozlišují mezi atributy, jež v podstatě jsou proměnnými skalárních datových typů (čísla, znaky, logické hodnoty atd.) a obsaženými objekty. Oboje je realizováno pomocí datových položek, hodnota reference je odkaz (pointer) na objekt, tedy adresa místa (v paměti počítače), kde se objekt nachází.

Obsažené objekty mohou vystupovat vůči obsahujícímu objektu v různých vztazích:

Asociace a agregace

Asociace je vztah mezi dvěma objekty, kdy jeden objekt využívá služeb druhého objektu tím, že mu posílá zprávy a z jejich výstupních parametrů získává potřebné informace. Často je tento vztah popisován jako klient-server (odběratel-dodavatel). Objekt, který má poskytovat služby je většinou do klientského objektu dosazen z vnějšku pouze za účelem vykonávání služeb. Asociace též může být obousměrná a to v případě, kdy oba objekty obsahují (referenci na) ten druhý objekt.

Speciálním typem asociace je agregace. Vnitřní objekt je logicky chápán jako součást nadřazeného objektu. Nadřazený objekt opět od vnitřního objektu požaduje služby, ale ten (např. ještě s jinými objekty) logicky tvoří stav nadřazeného objektu.

Důležité je také to, kdo by měl být zodpovědný (ať už přímo nebo nepřímo) za rušení asociovaných a agregovaných objektů. Objekt by nikdy neměl rušit asociované objekty, neboť mohou existovat i mimo tuto vazbu a zcela jiné objekty také mohou využívat jejich služeb. Naopak objekt by měl být zodpovědný za rušení svých agregovaných objektů.

Dědičnost

Dědičnost je vztahem mezi objekty, které jsou třídami. Lze tedy zkráceně (a v řeči těch prostředí kde třída je spíše typ proměnné) uvést, že dědičnost je vztah mezi třídami. Často nastává situace, kdy se při návrhu vyskytnou objekty z různých tříd, tak, že instance jedné třídy mají stejné vlastnosti jako instance jiné třídy a ještě nějaké přidávají. Projevuje se to tím, že reagují na stejné zprávy jako objekty té první z tříd. Z hlediska znovupoužitelnosti je vhodné tyto vlastnosti v druhé třídě znovu nedefinovat, ale použít.

Dědičností vzniká mezi třídami tzv. vztah předek–potomek, neboli potomek dědí z předka. Vztah dědičnosti mezi třídami je charakterizován takto:

- Instance předka mohou po zavedení tohoto vztahu nadále nezměněně existovat.
- Instance potomka obsahuje instanci předka, kterou ovšem nelze z vnějšku objektu nijak získat.
- Instance potomka reagují na všechny zprávy jako instance předka, ovšem zprávy, které samy nemapují na vykonávání svých metod, posílají instanci předka.
- Všude tam, kde je možno použít instanci předka, je možno použít i instanci potomka – tzv. princip zastupitelnosti předka potomkem.
- Vztah dědičnosti je tranzitivní. Dědí-li třída B z třídy A a třída C dědí z třídy B, pak také třída C dědí z třídy A.

Dědičností lze dosáhnout dvou efektů:

- Rozšíření předka – instance potomka definují nové atributy, metody, přijímané zprávy a obsažené objekty.
- Specializace předka – instance potomka reagují na zprávy, které přijímá i předek, vyvoláním jiné metody a to buď s využitím tzv. *brzké* nebo *pozdní* vazby. Tímto způsobem vzniká jev, který se nazývá *polymorfismus*, při němž dva různé objekty reagují na rozdílně na stejnou zprávu.

Při pozdní vazbě, ať už je zpráva vyslána odkudkoliv, příslušná metoda k vykonávání se hledá vždy od objektu, který je v dané posloupnosti objektů ve smyslu předek-potomek nejvýše. V případě nenalezení v jeho protokolu je teprve zpráva poslána své instanci předka. Při brzké vazbě se metoda pro vykonání hledá vždy od objektu, jemuž zpráva přišla, pokud není nalezena v jeho protokolu, je poslána své instanci předka.

Některé objektově orientované jazyky dědičnost vůbec nezavádějí, neboť není základní vlastností objektů. Jiné na ní naopak silně staví. Ve většině objektově orientovaných prostředích se při polymorfismu využívá pouze pozdní vazby, v některých je možno způsob vyřizování zprávy ke zprávě určit.

Třída nemusí mít pouze maximálně jednoho předka. Předků může být více a tím vyvstávají další problémy. Především, které instanci z instancí předků přeposílat nevyřízenou zprávu nejprve a kterým následně a v jakém pořadí. Z tohoto důvodu se vícenásobná dědičnost příliš nepoužívá ani v prostředích, která to umožňují.

Vlastnosti – properties

Zajímavým prvkem, který se začal objevovat v objektově orientovaných prostředích jsou tzv. properties (vlastnosti).

Některé zprávy mají zvláštní charakter a to z hlediska pohledu na daný objekt. Uvedené zprávy lze reprezentovat tak, jako by objekt vykazoval specifické vlastnosti (angl. properties).

Tyto zprávy jsou zvláštní pouze tím, že vypadají z vnějšku objektu jako čtení nebo nastavování těchto vlastností objektů. Zavedením vlastností do objektově orientovaného prostředí je možno zprávy týkající se čtení a nastavování jedné specifické vlastnosti spojit logicky v celek. Zpráva, jež vlastnost nastavuje, se nazývá *setter* a zprávě, která vlastnost čte, se říká *getter*.

Property jsou vhodné především pro zpřehlednění zápisu, ať už v programovacím jazyku nebo jinde, se zachováním zapouzdřenosti objektu, neboť navíc zavádějí efektivnější a přehlednější syntaxi pro zasílání těchto zvláštních zpráv.

1.3. Programovací jazyky a paradigmata

Programovací jazyky a struktura programu

Programovací jazyky původně vznikly pouze jako pomůcky pro urychlení psaní počítačového kódu. Brzy se však ukázalo, že takové chování není dostačující. Důvodem bylo, že počítačové programy byly stále komplexnější a tím pádem obsahovaly stále větší množství těžko odhalitelných programátorských chyb. Vznikl tak problém *spolehlivosti* programů. Původně se předpokládalo, že programátorské chyby jsou podobné těm pravopisným. Tato analogie byla brzy vyvrácena neboť chyby se dále vyskytovaly i v *pravopisně* správně napsaných programech. Za takových okolností se začalo přemýšlet, jak i tyto chyby odstraňovat.

Jako první možnost se nabízelo zdokonalené testování. Problém byl v tom, že velké programy, které se mohou nacházet v milionech stavů, není jednoduché úplně otestovat v relativně krátké době.

Následně bylo zjištěno, že testováním lze dosáhnout korektnosti programu pouze tehdy, pokud se zaměříme na jeho *vnitřní strukturu*. Do popředí zájmu se tedy dostala vnitřní struktura programu, která je samozřejmě úzce spjata s použitým programovacím jazykem.

Kvalitu programů určují v podstatě dva typy faktorů *vnitřní* a *vnější*. Vnější kvality jsou viditelné uživateli programu, vnitřní jen počítačovým odborníkům.

Mezi hlavní vnější faktory kvality programu se obvykle řadí:

- **Správnost** – schopnost programu přesně vykonávat svou úlohu, která byla dána v požadavcích zadavatele.
- **Robustnost** – schopnost programu pracovat i v abnormálních podmínkách.
- **Rozšiřitelnost** – snadnost, s jakou mohou být existující programy přizpůsobeny novým podmínkám.
- **Rychlost**, s jakou je program schopen plnit svou úlohu.
- **Efektivita** využívání tzv. zdrojů, tedy strojového času, paměti atd.

Následně bylo ukázáno, že vnějších faktorů kvality je možno dosáhnout pouze existencí vnitřních faktorů, tedy tehdy, bude-li mít program jasně definovanou vnitřní strukturu, podléhající určitému *stylu*.

Tento styl vnitřní struktury programů se nazývá *programovací paradigma*. A právě programovací jazyky nutí programátory vytvářet programy, jež odpovídají některému programovacímu paradigmatu.

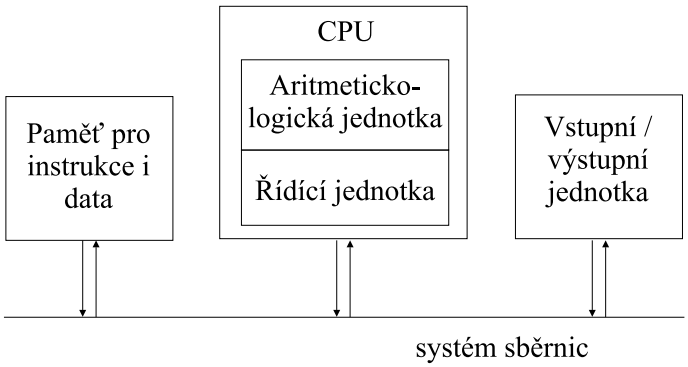
Co je přesně myšleno programovacím jazykem a programem?

Definice 1.24. *Programovací jazyk* je formální jazyk, jehož věty se nazývají *programy*. Kromě *syntaxe*, která popisuje vnitřní strukturu programů a je dána generující gramatikou, obsahuje programovací jazyk také *sémantiku*, která přiřazuje význam jednotlivým částem programů a tak význam i celým programům.

Program chápeme ze syntaktického hlediska jako strukturovanou větu programovacího jazyka. Ze sémantického hlediska jako množinu činností, která společně řeší určitou úlohu. Převod syntaktických částí programu na činnosti programu je dán právě sémantikou.

Jazyky nižší úrovně

Podoba programování a programovacích jazyků je dána základním principem práce počítače. Princip práce dnešních počítačů vychází z principu práce *von Neumannova stroje*. Von Neumannův stroj je teoretickým strojem a princip jeho práce se nazývá *von Neumannův princip*. Základní charakteristikou tohoto principu je, že data i instrukce programu sdílejí tutéž paměť.



Obrázek 1. Schéma von Neumannova stroje

Stroj se skládá z řídící jednotky, aritmetické jednotky, jednotky vstupů a výstupů a z paměti. V paměti jsou uloženy jednak instrukce programu, jednak data. Stroj pracuje tak, že instrukce jsou postupně brány z paměti a vykonávány, výsledky jsou ukládány zpět do paměti. Řízení výpočtu probíhá pomocí instrukcí skoku, kdy po provedení této instrukce pokračuje program vykováváním instrukce nacházející se na paměťové buňce zmíněné v instrukci skoku.

Program, který řídí tento stroj, se nazývá *strojový kód* a používají ho i dnešní počítače. Strojový kód je vlastně posloupnost bytů, která je srozumitelná danému stroji a je poměrně nesrozumitelný a nečitelný pro člověka.

Pro usnadnění psaní programů ve strojovém kódu byl vyvinut tzv. *jazyk symbolických adres*, nepřesně nazýván assembler. Jazyk symbolických adres umožnil nahradit bytové kódy

pomocí mnemotechnických zkratek a jednotlivá paměťová místa označit symboly. Program v jazyku symbolických adres nemůže být strojem přímo proveden, musí být nejprve přeložen do strojového kódu. Překlad v podstatě probíhá přímočarým nahrazením zkratek a symbolů jejich číselnými ekvivalenty a je tedy velmi jednoduchý. Programy jsou tedy sestavovány (asemblovány) překladačem, zvaným *assembler*.

Strojový kód a jazyk symbolických adres se dnes řadí mezi jazyky nižší úrovně a někdy ani nejsou považovány za „pravé“ programovací jazyky. Od programovacích jazyků se očekává vyšší forma abstrakce.

Jazyky vyšší úrovně

Tvorba programů v jazyku symbolických adres je velmi nákladná a pomalá. Proto během let používání assembleru neustávala snaha o automatizaci vytváření programů pro assembler, tedy tvorbu programovacích jazyků vyšší úrovně. Zprvu ovšem převažoval názor, že dobré programy lze vytvořit jen v jazycích nižší úrovně, neboť jsou přímo „ušity“ pro daný stroj.

Prvním úspěchem v této oblasti byl jazyk FORTRAN pro překlad matematických výrazů. Následovaly další a další programovací jazyky a vznikají dodnes. Triumfem vyšších programovacích jazyků bylo, když byl jazyk C použit k vytvoření operačního systému UNIX. Operační systémy, jakožto základní programy ovládající hardware, byly totiž doposud považovány za doménu použití jazyků nižší úrovně.

Programy napsané ve vyšších programovacích jazycích opět samozřejmě nelze přímo provádět strojem. Programy je tedy nutno před spuštěním transformovat do strojového kódu – přeložit. Programy, které překládají programy z jednoho jazyka do druhého, se nazývají *překladače*. Překladačům se věnují části 1.4. a 3.

Vztah programu, programovacího jazyka a stroje

Dá se říci, že program, programovací jazyk a stroj (strojový jazyk) představují tři odlišitelné vrstvy, kde každá vrstva využívá služeb vrstvy podřízené. Programovací jazyk využívá instrukcí a jiných možností jazyka stroje ke tvorbě svých nových konstrukcí a ty nabízí programům k použití. Ze služeb, které programovací jazyky poskytují programům je možno uvést:

- Výpočetní model, tzn. překrytí výpočetního modelu stroje jiným výpočetním modelem.
- Datové typy a operace podporované strojem.
- Podpora abstrakce – možnost zavádět nové typy a operace a s nimi pracovat stejně jako s typy a operacemi stroje.
- Kontrola správnosti – mnoho programátorských chyb lze odhalit již při překladu.

Přehled základních programovacích paradigmat

Tato část obsahuje výčet základních programovacích paradigmat, tedy ověřených programovacích stylů, které vedou k tvorbě kvalitního softwaru.

Procedurální paradigma

Procedurální paradigma bývá též nazýváno imperativní nebo klasické. Toto paradigma se asi nejméně odlišuje od strojového jazyka, neboť důležitou roli zde hrají *příkazy*, především přiřazovací příkaz. Průběh výpočtu je dán sekvencí po sobě následujících příkazů a program je logicky dělen do modulů tzv. *procedur*.

Toto paradigma používají například programovací jazyky Fortran, C, nebo Pascal. Použití je univerzální.

Funkcionální paradigma

Ve funkcionálním paradigmatu je program členěn do modulů – *funkcí*. Průběh výpočtu je založen na postupném *aplikování* funkcí na výsledky aplikace jiných funkcí. Toto paradigma nepoužívá přiřazovací příkaz a typické pro něj je použití rekurze.

Typickými programovacími jazyky pro toto paradigma jsou Lisp a jeho dialekty např. Scheme. Použití je v oblasti umělé inteligence, v počítačové grafice, výuce a výzkumu.

Objektově orientované paradigma

Základními moduly programu jsou útvary zvané *objekty*. Objektově orientovanému přístupu se věnuje kapitola 1.2. Tyto objekty modelují objekty reálného světa. Každý objekt má svůj *stav* a *metody* pro jeho změnu. Běh programu je dán zasíláním *zpráv* mezi objekty, objekty reagují na příjem zprávy provedením určené metody, v jejímž průběhu může objekt poslat zprávu jinému objektu.

Ke starším jazykům tohoto paradigmatu patří Simula nebo SmallTalk, k moderním jazyk C++ a jazyky pro platformu .NET, např. C# a Visual Basic. Ty jsou ovšem odvozeny od procedurálních jazyků. K čistě objektovým jazykům patří Eiffel. Původní použití paradigmatu bylo v oblasti simulací a umělé inteligence, dnes je již univerzální.

Logické paradigma

Program je tvořen množinou faktů a pravidel – tzv. *klauzulí*. Průběh výpočtu je založen na hledání *důkazu* nebo vyvrácení předloženého tvrzení pomocí množiny klauzulí. Programy obsahují specifické rysy odlišné od ostatních paradigmat, např. to, že je dáno, co se má počítat, a nikoliv jak a kam uložit mezivýsledky. Programovacím jazykem používající toto paradigma je Prolog a jeho dialekty. Použití je zatím pouze v oblasti umělé inteligence.

Paralelní paradigma

Programy mohou být dekomponovány na úlohy, které mohou běžet paralelně. Klíčovými pojmy jsou *procesy* a jejich *komunikace* a *synchronizace*.

Jazyky určené pro toto paradigma jsou například jazyky SR a MPD. Použití je zatím jen ve výzkumu, neboť jazyky jiných paradigmat také mohou procesy a jejich synchronizaci a komunikaci používat s použitím svých konstrukcí.

Jednotlivá paradigmata se samozřejmě vyvíjejí. Často dochází k tomu, že jedno paradigma přebírá prvky jiných paradigmat. Například příkazy z imperativního paradigmatu pronikají téměř do všech ostatních. Objektově orientované paradigma je dnes komerčně nejpoužívanější a proto také do sebe přetahuje prvky z jiných paradigmat – paralelního paradigmatu a dokonce z funkcionálního paradigmatu, jako například lambda výrazy a funkce vyššího řádu.

Struktura a prvky programovacích jazyků

Pro dosažení kvality programů je nutné, aby programy měli jistou přesně definovanou vnitřní strukturu. Tato struktura je dána syntaxí jazyka a podléhá danému paradigmatu.

Všechny vyšší programovací jazyky poskytují programům tři nástroje:

1. Primitivní (atomární) prvky reprezentující jednak data a základní akce a dále pomocné prvky.
2. Různé mechanismy pro sestavování složitějších prvků z jednodušších.
3. Mechanismy pro pojmenovávání složitějších prvků a tím možnost s nimi pracovat stejně jako s některými primitivními prvky.

Atomy jazyka

Atomy (lexikální symboly) jsou z pohledu jazyka základní, nedělitelné prvky. V textovém zápisu programu jsou ovšem složeny z posloupností textových znaků.

Atomy jazyka se obvykle rozdělují do několika kategorií:

- Identifikátory – používají se k pojmenování některých entit a umožňují tak abstrakci těchto entit. Uvedením tohoto jména jinde v programu je možné danou entitu použít.
- Klíčová slova – obvykle vypadají jako identifikátory, ale mají v programovacím jazyku speciální význam a jako identifikátor je nelze použít.
- Literály – složí pro zápis konstantních hodnot různých typů přímo v programu, např. čísel, řetězců logických hodnot apod.
- Operátory, znaménka – se používají pro logické spojování nebo naopak oddělování částí programu, jsou jimi například aritmetické operátory, různé druhy závorek, tečka, čárka, středník apod.

Jak již bylo uvedeno, programovací jazyky obsahují mechanismy pro sestavování složitějších prvků z jednodušších. Jaké druhy složených prvků programovací jazyk obsahuje, je pravě dáno použitým programovacím paradigmatem.

Výrazy

Složenými prvky programů, které se vyskytují téměř ve všech paradigmatech jsou *výrazy*. Výrazy jsou tvořeny buď samostatnými atomy, například literály nebo identifikátory, ale častěji z operátoru a jeho operandů, jimiž jsou další výrazy, tzv. podvýrazy. Operátory výrazů jsou nejrůznějších typů, některé jsou obvyklé ve většině jazyků, jiné jsou jejich specialitou. Nejběžnějšími výrazy jsou samozřejmě aritmetické výrazy sčítání, odečítání, násobení a podobně, dále relační operátory rovno, větší, menší a podobně. Výrazy obvykle slouží pro výpočet různých hodnot.

Příkazy

Dalšími složenými prvky programů, které se vyskytují téměř ve všech paradigmatech jsou *příkazy*. Příkazy většinou slouží pro řízení chodu výpočtu.

Nejčastějším příkazem je tzv. podmiňovací příkaz (if). Je složen ze dvou až tří částí – podmínky, prvního příkazu a případně druhého příkazu. Příkaz je proveden tak, že je vyhodnocena podmínka, pokud je pravdivá provede se první příkaz. Pokud je podmínka nepravdivá, pak se provede druhý příkaz (pokud je uveden).

Dalším nečastějším typem příkazů jsou cykly. Úkolem cyklů je opakovat vykonávání daných příkazů. Opakování cyklu bývá zastaveno nesplněním určité podmínky. Cykly bývají různých typů – s podmínkou na začátku, na konci, s řídící proměnnou apod.

Některá paradigmata příkazy nepoužívají, jiná jsou na nich naopak založena. Ve funkcionálním paradigmatu jsou např. místo cyklů používá rekurze.

Naopak v imperativním paradigmatu jsou příkazy silně podporovány a důležitou roli hraje přiřazovací příkaz. Jeho provedením se uložena hodnota na paměťové místo označené např. jménem (proměnnou).

Moduly

Programovací paradigmata většinou určují, aby prvním cílem strukturalizace programu bylo rozdělení rozsáhlého problému na jisté menší, snadno pochopitelné a otestovatelné části. Tyto části se obecně nazývají *moduly*.

Je vhodné, aby moduly měly několik základních rysů:

- Moduly musí být jasně rozpoznatelné již ze zápisu programu v programovacím jazyce. Musí být evidentní, kde modul začíná a kde končí. Tento úsek nebo úseky programu se nazývají *deklarace* modulů.
- Modul by měl vykonávat jednu definovanou, případně více jasně definovaných úloh.
- Modul musí mít tzv. explicitní rozhranní, tj. z definice modulu musí být jasné, jakou úlohu vykonává a jaké jsou předpoklady pro toto vykonání.
- Modul musí být relativně samostatný, pokud komunikuje s jinými moduly, mělo by jich být co nejméně.
- Modul musí mít právo na jisté soukromí, tedy ne všechny informace, které modul používá, musí být viditelné z vnějšku modulu, atd.

Funkce

Jedním z nejběžnějších a nejpoužívanějších modulů, vyskytujících se téměř ve všech paradigmatech, je funkce.

Funkce jakožto moduly se do informatiky dostaly jistě z matematiky. Matematická funkce je jisté zobrazení přiřazující každému prvku z definiční množiny jeden prvek z množiny hodnot. Matematická funkce nemusí říkat nic o tom, jak prvek z množiny hodnot k prvku z definičního oboru získat. Informatika chápe funkce jako algoritmus. Navíc funkce v informatice splňuje definici modulu.

Funkce jsou v různých paradigmatech nazývány různě – procedura, podprogram, rutina, metoda – ale vždy se jedná o stejný typ modulu.

Část programu, která obsahuje zápis funkce, se nazývá *deklarace* funkce a obsahuje typicky několik částí: jméno funkce, seznam formálních parametrů a tělo funkce.

Použití funkce v rámci některého výrazu programu nazýváme *volání* nebo *aplikace* funkce. Parametry, uvedené při volání funkce se nazývají *skutečné* (aktuální parametry), neboli *argumenty*.

Provedení volání funkce probíhá takto:

1. Nejprve jsou provedeny výrazy, jež představují aktuální parametry,
2. výsledky těchto výrazů jsou přiřazeny do formálních parametrů,
3. provede se tělo funkce, které používá v podobě formálních parametrů aktuální parametry,
4. výsledek provedení těla funkce (pokud výsledek je) se stává výsledkem volání funkce.

Každé volání funkce si v paměti počítače vytváří svoji vlastní kopii formálních (aktuálních) parametrů. Pokud by funkce volala nějakou jinou funkci, parametry obou funkcí budou existovat v paměti počítače zároveň a nezávisle na sobě, i když budou mít stejná jména. Speciálně to platí pro, tzv. *rekurzivní* volání, kdy funkce ve svém těle aktivuje sama sebe.

Je důležité striktně rozlišovat mezi funkcemi (jejich deklaracemi) a jejich voláními. Funkce existuje v programu jedinkrát a trvale, kdežto volání funkce se může v programu vyskytovat vícekrát a rámci provádění programu je volání mnohokrát iniciováno, proběhne a skončí.

Žádaný efekt vlastní kopie parametrů každého volání funkce většinou úzce souvisí s rozložením programu v paměti počítače. Konkrétně tento efekt bývá realizován pomocí aktivačního zásobníku. Ten roste a zase ubývá podle toho, jak probíhá vykonávání programu. Každé vyvolání funkce přidá jeden záznam na zásobník. Po ukončení daného vyvolání je tento záznam odebrán. Vyvolá-li funkce ve svém těle jinou funkci, dojde k tomu, že další záznam je vložen nad záznam volající funkce. Jakmile vyvolání funkce skončí, řízení programu se vrací bezprostředně za místo, odkud byla funkce volána. Tyto tzv. aktivační záznamy obsahují především kopie parametrů volaných funkcí.

Vedlejší efekt funkcí

V čistém funkcionálním paradigmatu vyvoláním funkce dojde v těle funkce k výpočtu návratové hodnoty pomocí hodnot aktuálních parametrů, dalších proměnných a konstant. Pro tyto funkce, stejně jako pro matematické funkce, není přípustné, aby funkce modifikovala jakékoliv hodnoty vně funkce.

Pro funkce v jiných paradigmatech je přípustné, aby modifikovaly hodnoty vně funkce, tento jev se nazývá *vedlejší efekt* funkce. Tato situace může jít tak daleko, že funkce má pouze vedlejší efekt a nemá žádnou návratovou hodnotu. Takové funkce se obvykle nazývají *procedury*.

Proměnné a rozsah jejich platnosti

Proměnnou obecně nazýváme místo v paměti, jež je označeno nějakým jménem (symbolem) a hodnota uložená na tomto místě v paměti se v průběhu vykonávání programu může měnit. Důležité je, že k proměnným se v programu přistupuje pomocí jejich jmen. Proměnné obecně mohou být deklarovány vně i uvnitř různých modulů programů. Deklarací proměnné se rozumí především uvedení jejího jména a dalších informací v programu.

Základním poznatkem je, že pokud je proměnná deklarována uvnitř modulu, je v něm přístupná, tedy je možné její jméno používat v nejrůznějších výrazech programovacího jazyka, uvedených uvnitř modulu. Taková proměnná se nazývá *lokální* danému modulu. Některé proměnné definované vně modulu, mohou být uvnitř modulu taktéž přístupné. Které to mohou být, se již liší jazyk od jazyka. Proměnné, které jsou přístupné ve všech modulech, se někdy nazývají *globální* proměnné.

Taktéž může nastat situace, že proměnná deklarovaná vně modulu, které nic nebrání v tom, aby mohla být v modulu přístupná, je zde nepřístupná. Tato situace nastává, když některá lokální proměnná modulu má stejné jméno jako tato vnější proměnná. V tomto případě platí, že lokální proměnná „zastíní“ vnější proměnnou a ta se tím stává nepřístupná.

Rozlišujeme v postatě dvě možnosti, jak může programovací jazyk definovat pojem „vně“ modulu:

1. Vnějšek modulu je chápán *lexikálně*, tedy tak, jak jsou jednotlivé moduly do sebe vnořovány v zápisu programu. Vnějšek modulu je jednoznačně určen již před spuštěním programu.

2. Vnějšek modulu je chápán *dynamicky*, tedy je určen těmi aktivačními záznamy modulů, z kterých byl daný modul aktivován. Vnějšek modulu je tak dán až v průběhu vykonávání programu.

V prvním případě daný jazyk podporuje *statický (lexikální) rozsah platnosti proměnných*, ve druhém případě *dynamický rozsah platnosti proměnných*. Moderní programovací jazyky preferují statický rozsah platnosti, dynamický rozsah platnosti proměnných používá například jazyk Lisp.

Typy a typové systémy programovacích jazyků.

V programech se vyskytují různé druhy prvků – deklarace modulů, příkazy, výrazy apod. Výrazy jsou tvořeny z operací a z operandů, jimiž mohou být další složené výrazy nebo jednoduché výrazy – konstanty, proměnné apod. Každá konstanta, proměnná nebo složený výraz může nabývat jen určitých hodnot a lze s nimi provádět jen určité operace. V této souvislosti se hovoří o (datovém) *typu* výrazu. Typ výrazu určuje, jakých hodnot může nabývat a jaké operace na něj mohou být aplikovány.

Většina programovacích jazyků obsahuje typ často zvaný `Integer` (nebo podobně), který reprezentuje celá čísla v daném rozsahu a operace s nimi. Výrazy s typem `Integer` mohou nabývat hodnot např. `10`, `0`, `-654` atd., ale nemohou např. nabývat hodnot `-1.728`, `"abcd"` nebo `false`, neboť tyto výrazy nejsou celými čísly. Déle je možné výrazy typu `Integer` např. sčítat nebo odečítat, ale není je možné převádět např. na malá písmena nebo konkatenovat.

Druhy datových typů

Datové typy lze kategorizovat podle dvou kritérií: podle toho, na jaké úrovni jsou definovány a podle jejich struktury.

Podle toho, na jaké úrovni jsou typy definovány, rozeznávají se :

- typy na úrovni stroje – tyto typy přímo podporuje stroj a provádí s nimi základní operace, jež jsou integrovány přímo v elektronických obvodech stroje. Těmito typy mohou být např. typy `Byte`, `Integer32`, `Float32`.

- typy na úrovni jazyka – tyto typy simuluje programovací jazyk pomocí strojových typů, algoritmy příslušných operací jsou součástí jazyka. Patří sem např. typy `Boolean`, `String`, `Array`.
- typy na úrovni programu – tyto typy definuje programátor v programu a proto jsou různé program od programu, časté jsou např. `List`, `Person` apod.

Podle struktury dělíme typy na:

- jednoduché – jsou dále nedělitelné, tedy atomární. Např. `Integer`, `Char`, `Boolean`.
- strukturované – jsou tvořeny několika jednoduchými datovými typy. Tyto typy musí definovat operace pro přístup k jednotlivým svým položkám (tzv. *selektory*). Strukturované typy lze ještě členit v podstatě na dva druhy:
  - Pole – obsahují více položek stejného typu. K těmto položkám se přistupuje uvedením jednoho nebo více celočíselných indexů.
  - Záznam (struktura) – obsahují více položek různých typů. Záznamy odpovídají kartézskému součinu více typů. K položkám se nejčastěji přistupuje pomocí selektoru `.` (tečka) a uvedením identifikátoru položky.

V některých programovacích jazycích může být hodnotou výrazu i funkce. Funkce je pak nutné považovat za další rovnocenný datový typ.

Typové systémy jazyků

Soubor pravidel, která přiřazují výrazům jejich typ, se nazývá *typový systém* jazyka. Typ výrazu se určuje na základě typů operandů a uvedené operace. U atomárních výrazů je možné typ získat například z deklarace proměnné, u konstant je určen přímo kompilátorem. K některým výrazům typ nelze najít a tedy nemohou být provedeny. Tato situace nastává, když na žádné úrovni není definována uvedená operace pro dané typy operandů. Proces přiřazení datového typu výrazů se též nazývá *typová kontrola*. Typová kontrola má velký význam v programování, neboť umožňuje odhalit značné množství programátorských chyb.

Programovací jazyky používají obecně dva druhy typových systémů:

- *Silný typový systém* umožňuje uvádět jen bezpečné výrazy. Bezpečné výrazy jsou takové výrazy, jejichž provedení za chodu programu nemůže způsobit typovou chybu.
- *Slabý typový systém* je typový systém, který není silný.

Existují v zásadě dva přístupy, kdy provádět typovou kontrolu.

- Provádí-li jazyk typovou kontrolu již *při překladu*, je nutné již v této době přesně znát typy všech nekonstantních částí výrazů – proměnných, návratové typy funkcí apod. Typový systém požaduje *explicitní* oznámení typu nekonstantních prvků ještě dříve, než jsou použity ve výrazu. Explicitní uvedení typu se nejčastěji vyskytuje v deklaracích proměnných. Tento přístup znemožňuje, aby se typ proměnné nebo funkce za běhu měnil.

- Druhou možností je provádět typovou kontrolu až *při vykonání* výrazu. Tento způsob je pružnější, nevyžaduje deklarace typů proměnných a funkcí a tudíž jejich typy se mohou měnit za běhu programu.

Oba systémy mají své výhody i nevýhody. Programy, u kterých se typové kontrola provádí již při překladu jsou obvykle rychlejší a mají menší paměťové nároky. Navíc programy jsou přehlednější a lépe se ladí. Naopak tento systém bývá často méně pružný a nastávají situace, kdy je potřeba program zapsat složitěji, než se zdá z lidského hlediska.

Silné typové systémy obvykle používají programovací jazyky pro komerční, robustní, programy, slabé typové systémy jsou pružnější, proto se často používají v jazycích pro vědu a výzkum, navíc zde tolik nezáleží na rychlosti programu.

1.4. Překladače a interprety

Společně s tvorbou nových a nových programovacích jazyků také vznikla nová disciplína v informatice: konstrukce *překladačů* – speciálních programů, které transformují programy z jednoho jazyka do druhého. Nejčastěji se jedná o převod z některého vyššího programovacího jazyka do jazyka stroje. Zároveň se vytvořily dva přístupy k překladu programů: přístup *kompilační* a *interpretační*. Překladače se proto dělí na dvě skupiny:

- **Kompilátory** (klasické překladače) nejdříve přeloží celý program. Poté nejčastěji uloží program v cílovém jazyku (nejčastěji strojovém kódu) do vnější paměti (do souboru). Překlad celého programu tedy předchází jeho spuštění. Následně může být program spuštěn načtením z vnější paměti. V případě, že při překladu dojde k chybám cílový program vůbec není vytvořen.

- **Interprety** překládají program souběžně s během programu podle potřeby, cílový program jako celek nevzniká, nebo je uložen jen v operační paměti. Výhodou je, že se program může za běhu sám modifikovat, nevýhodou je např. to, že program běží pomaleji a ke svému chodu vždy potřebuje interpret. Jako interprety se také označují překladače, které sice nejdříve přeloží celý program, ale neukládají jej do vnější paměti, takže program může být spuštěn jedině ihned po překladu dokud je v operační paměti.

Je zajímavé, že některé jazyky jsou svou povahou spíše kompilační a jiné interpretační, ovšem i toto je spíše otázkou technické realizace.

Překladače se skládají z několika fází, která postupně analyzují program v dané formě a vytvářejí novou (další) formu programu. První formou programu je samotný zdrojový program jako posloupnost znaků, dalšími formami jsou rozličné tzv. interní formy programu a poslední formou je program zapsaný v cílovém jazyce, kterým často bývá přímo strojový kód.

Překladače obvykle obsahují přinejmenším čtyři fáze překladu:

1. **Lexikální analýza** zpracovává řetězec znaků zdrojového programu a rozděluje je do tzv. lexikálních symbolů (atomů), které jsou nejzákladnějšími prvky (terminálními symboly) programovacího jazyka, pro nějž je překladač určen. Přitom lexikální analyzátor vynechává znaky vstupního programu, které netvoří atomy, protože slouží k formátování zdrojového textu a oddělení atomů (mezery a pod.), nebo tvoří např. komentáře.

2. **Syntaktická analýza** zpracovává posloupnost atomů jazyka a seskupuje je do další formy jazyka, kterou nejčastěji bývá upravený syntaktický (derivační) strom programu.

Dále se při syntaktické analýze vytváří datové struktury, které uchovávají informace o identifikátorech apod., tzv. tabulky symbolů.

3. **Sémantická analýza** kontroluje správnost programu v interní formě vytvořené syntaktickou analýzou, případně ji doplňuje o další informace. Správnost se kontroluje z hlediska sémantických pravidel zdrojového jazyka, k čemuž se využívá také tabulky symbolů. Nejčastěji se kontroluje správné použití identifikátorů, kontrola typů výrazů apod.

4. **Generování kódu** transformuje program v interní formě do cílového jazyka. Tato část bývá poměrně důležitá hlavně u kompilačních překladačů, které překládají přímo do strojového kódu daného stroje.

Všechny fáze překladu (snad kromě generování kódu) mohou produkovat chybová hlášení, neboť zdrojový program neodpovídá správnému programu v daném programovacím jazyce. Výskyty převážné většiny chyb ukončují překlad neúspěchem.

Velká část programovacích jazyků umožňuje, aby všechny čtyři fáze překlady mohly být provedeny jedním průchodem zdrojovým programem. Takový typ překladu bývá nazýván *syntaxí řízený překlad*. Veškeré zpracování programu probíhá při vytváření interní formy programu syntaktickým analyzátorem.

Některé jazyky, např. pokud nevyžadují tzv. předsunuté deklarace, je ovšem potřeba překládat ve dvou a více průchodech: první průchod obsahuje lexikální a syntaktickou analýzu a produkuje interní formu programu a další pomocné struktury, druhý průchod prochází již interní formu programu a obsahuje sémantickou analýzu a generování kódu. Generování kódu je v některých případech potřeba provádět v dalším samostatné průchodu.

Některé jazyky také umožňují provádět část sémantické analýzy v prvním průchodu.

2. Specifikace jazyka MiniC#

Jazyk MiniC# (čteno mini sí šárp) vznikl v souvislosti s touto diplomovou prací. Lze říci, že jazyk MiniC# je podmnožinou jazyky C#. Jazyk C# je moderní objektově orientovaný, typově bezpečný programovací jazyk, jež patří do rodiny jazyků vycházejících z jazyka C. Jazyk C# vznikl jako hlavní programovací jazyk pro platformu .NET společnosti Microsoft v roce 2001.

Z jazyka C# ve verzi 1.2 jsou do jazyka MiniC# vybrány všechny prvky potřebné pro objektově orientovaný jazyk a další konstrukty, které jsou nejpoužívanější a užitečné.

Zjednodušeně (a tedy pro znalé jazyka C#) by se dalo říci, že MiniC# obsahuje vše z jazyka C# kromě:

- členění typů do jmenných prostorů namespace,
- deklarací uživatelských hodnotových typů,
- deklarací vnořených typů,
- deklarací a tudíž i používání rozhranní,
- deklarací a tudíž i používání delegátů,
- používání uživatelských atributů,
- deklarací indexerů a přetížených operátorů,
- systému výjimek,
- deklarací a používání tzv. zubatých (jagged) polí, tedy polí, jejichž prvky jsou pole stejných dimenzí, ale potažmo různých rozsahů,
- některých méně obvyklých výrazů a příkazů, konkrétně `goto`, `typeof`, `checked`, `unchecked`, `lock`, `using`,
- používání identifikátorů se jmény shodnými s klíčovými slovy (tedy v C# s předponou `@`).

Dále MiniC# neobsahuje ty prvky, které v důsledku vynechání výše uvedených prvků nemají smysl, například neobsahuje modifikátory typů, nebo `extern` modifikátor pro funkční členy tříd.

2.1. Charakteristiky jazyka

Objekty

Jazyk MiniC# vykazuje čistě objektově orientované rysy. Není možné používat například globální funkce nebo proměnné. Plně podporuje dědičnost a polymorfismus. Všechny entity v paměti běžícího programu jsou *objekty*. Co se týče tříd, terminologie plně neodpovídá objektově orientovanému přístupu. Objekty jsou v MiniC# (z historických důvodů) instancemi *typů*. *Třídou* je nazýván uživatelsky definovaný tzv. odkazový typ (viz sekce 2.2.).

Typová bezpečnost

MiniC# je typově bezpečný jazyk. Znamená to, že nedovoluje uvést výraz daného typu na místě, kde je požadován výraz jiného typu. Naštěstí jazyk poskytuje řadu implicitních konverzí pro základní typy a také unifikovaný typový systém (viz sekce 2.2.). Typová bezpečnost umožňuje psát bezpečnější a robustnější programy a téměř veškeré typové kontroly probíhají jen při kompilaci programu.

Proměnné a jejich hodnoty

Proměnné představují místa v paměti označená symboly. Každá proměnná má přiřazený svůj datový typ. Díky typové bezpečnosti nelze do proměnné přiřadit hodnotu jiného typu. V jazyku MiniC# existuje několik kategorií proměnných: statické datové položky, instanční datové položky, prvky polí, hodnotové parametry, odkazové parametry, výstupní parametry a lokální proměnné.

V jazyce MiniC# je vyžadováno, aby proměnná byla definitivně přiřazena předtím, než je použita. Statické a instanční datové položky, prvky polí a hodnotové a odkazové parametry jsou vždy počátečně přiřazeny a mají svoji počáteční hodnotu. U ostatních typů kompilátor analyzuje možné větve výpočtu a v místě použití hodnoty proměnné kontroluje, zda byla přiřazena v každé možné větvi programu vedoucího do tohoto místa.

Hodnotami proměnných hodnotových typů jsou přímo instance objektů. Hodnotami proměnných odkazových typů jsou odkazy (pointery, reference) na objekty uložené na paměťové haldě programu, nebo hodnota `null`, která představuje nepřítomnost instance.

Proměnným, které jsou považovány za počátečně přiřazené, je při jejich alokaci automaticky přiřazena počáteční hodnota. Pro odkazové typy je to hodnota `null`, pro hodnotové typy jsou jimi hodnoty `0`, `0.0`, `false`, `'\0'` (podle typu), tedy hodnoty, jež odpovídají vyplnění paměti objektu bitově nulami.

Automatická správa paměti

Jazyk MiniC# používá automatickou správu paměti, která oprošťuje autory programů od manuálního alokování a uvolňování paměti zabírané objekty. Automatickou správu paměti zajišťuje tzv. garbage collector (sběrač smetí).

Životní cyklus objektu z hlediska správce paměti je následující:

1. V okamžiku vzniku objektu mu správce paměti přidělí paměť, zavolá se konstruktor a objekt se považuje za živý.
2. Jestliže k objektu nelze přistoupit žádným možným pokračování běhu programu, objekt je považován za nepoužitelný a je tedy vhodný ke zrušení.
3. Po nějakém čase (nejčastěji v případě potřeby paměti pro nový objekt) je paměť přidělená objektu, který je určen ke zrušení, uvolněna.

2.2. Typový systém

Jazyk MiniC# používá unifikovaný typový systém. Znamená to, že typy tvoří hierarchickou strukturu s jediným kořenem tvořeným typem `object`. Vztahy mezi typy jsou dány vztahem dědičnosti. Jestliže jeden typ dědí z jiného typu, znamená to, že přebírá některé jeho

tzv. členy. Některé členy mohou být potomkovi skryty. MiniC# podporuje jen jednonásobnou (jednoduchou) dědičnost. Znamená to, že každý typ (kromě typu `object`) má jen jednu přímou základní třídu. Typ `object` nemá žádnou základní třídu.

Typy se dělí do dvou kategorií: hodnotové typy a odkazové typy, podle toho, jak se s jejich instancemi zachází v paměti. Hodnotami hodnotových typů jsou přímo objekty, kdežto hodnotami odkazových typů jsou vždy reference na objekt uložený na haldě programu.

Hodnotové typy jsou dále děleny na jednoduché typy a na výčtové typy. Odkazové typy se dělí na třídy a pole. Pouze výčtové typy a třídy jsou uživatelsky definovatelné.

Jednoduché hodnotové typy

Jazyka MiniC# obsahuje několik předdefinovaných hodnotových typů, nazývaných jednoduché hodnotové typy. Tyto typy jsou identifikovány pomocí rezervovaných slov jazyka. Jednoduché hodnotové typy jazyka MiniC# jsou tyto:

`bool`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`, `decimal`

Jednoduché typy jsou běžnými typy, proto obsahují jednak zděděné, jednak vlastní členy. Navíc jazyk s nimi dovoluje pracovat i odlišně než s uživatelem definovanými typy:

- Hodnoty jednoduchých typů je možno zapisovat pomocí literálů.
- Výrazy obsahující pouze literály, konstantní výrazy, je možno spočítat již při překladu.

Následuje popis jednotlivých jednoduchých typů:

- Typ `byte` reprezentuje booleovské logické hodnoty `true` a `false`. Neexistují žádné konverze mezi tímto typem a číselnými typy.
- Typ `byte` reprezentuje 8-bitová celá čísla bez znaménka z rozsahu 0 až 255.
- Typ `short` reprezentuje 16-bitová celá čísla se znaménkem z rozsahu -32768 až 32767.
- Typ `int` reprezentuje 32-bitová celá čísla se znaménkem z rozsahu -2147483648 až 2147483647.
- Typ `long` reprezentuje 64-bitová celá čísla se znaménkem z rozsahu -9223372036854775808 až 9223372036854775807.
- Typ `char` reprezentuje 16-bitová celá čísla bez znaménka z rozsahu 0 až 65535. Hodnoty odpovídají hodnotám znaků Unicode.
- Typ `float` reprezentuje 32-bitová čísla v plovoucí čárce dle standartu IEEE 754. Reprezentuje reálná čísla v rozsahu 1.5×10^{-45} až 3.4×10^{38}, s přesností na 7 míst.
- Typ `double` reprezentuje 64-bitová čísla v plovoucí čárce dle standartu IEEE 754. Reprezentuje reálná čísla v rozsahu 5.0×10^{-324} až 1.7×10^{308}, s přesností na 15 míst.
- Typ `decimal` reprezentuje 128-bitová reálná čísla. Je vhodný pro finanční operace. Rozsah typu je 1.0×10^{-28} až 7.9×10^{28} s přesností na 28 míst.

Všechny výše uvedené typy dědí přímo z typu `object`.

Výčtové typy

Výčtové typy jsou samostatnými typy s pojmenovanými konstantami. Všechny výčtové typy mají svůj tzv. podložní (underlying) typ, kterým je typ `int`. Všechny výčtové typy dědí z typu `Enum`. Množina hodnot výčtového typu je stejná jako množina hodnot typu `int`. Hodnoty výčtového typu *nejsou* omezeny jen na hodnoty pojmenovaných konstant.

Odkazové typy

Hodnoty odkazových typů jsou reference na instance těchto typů, nazývaných objekty. Speciální hodnota `null` je kompatibilní se všemi odkazovými typy a reprezentuje nepřítomnost instance.

Třídy

Třídy umožňují deklarovat struktury obsahující datové členy (datové položky a konstanty) a funkční členy (konstruktory, metody, vlastnosti). Třídy plně podporují dědičnost.

Jazyk MiniC# obsahuje některé předdefinované třídy:

- Třída `object` představuje nejzákladnější typ všech ostatních typů.
- Třída `string` reprezentuje řetězce znaků Unicode. Hodnoty typu `string` mohou být zapsány pomocí literálů.
- Třída `Enum` reprezentuje základní třídu pro všechny výčtové typy, obsahuje pro ně některé společné členy.
- Třída `Array` reprezentuje základní třídu pro všechna pole, obsahuje pro ně některé společné členy.

Pole

Pole jsou datové struktury, které obsahují daní počet proměnných přístupných pomocí indexů. Jazyk MiniC# podporuje jedno i vícedimenzionální pole. Všechna pole dědí implicitně z třídy `Array`, která deklaruje některé společné vlastnosti. Pole nelze uživatelsky deklarovat, jazyk je implicitně deklaruje při použití. Pole se chovají jako jiné objekty, jazyk pro ně navíc nabízí speciální syntaxi pro přístup k jejich prvkům.

2.3. Lexikální struktura programu

Program v jazyku MiniC# se skládá z jednoho nebo více zdrojových souborů. Zdrojový soubor se skládá z posloupnosti znaků kódování Unicode. Zdrojový soubor obvykle odpovídá souboru v souborovém systému, ale tato příslušnost není vyžadována.

Řetězec znaků zdrojového souboru musí odpovídat dané lexikální struktuře. Ta je definována jako posloupnost lexikálních elementů jazyka. Lexikální elementy jazyka MiniC# jsou následující: konce řádků (line terminators), mezery (white space), komentáře (comments) a lexikální symboly (atomy, tokens).

Lexikální zpracování zdrojového souboru probíhá redukování posloupnosti znaků na posloupnost lexikálních symbolů, ostatní lexikální elementy slouží k oddělení lexikálních symbolů jazyka. Tato posloupnost lexikálních symbolů je vstupem pro syntaktickou analýzu.

Jestliže určitá posloupnost znaků zdrojového souboru odpovídá více lexikálním elementům, lexikální analyzátor vždy formuje nejdelší možný element.

Komentáře

Komentáře jsou v MiniC# dvou forem. Jednořádkový komentář začíná `//` a pokračuje až do konce řádku.

Ohraničený komentář začíná dvojicí znaků `/*` a končí dvojicí znaků `*/`. Ohraničené komentáře mohou zabírat více řádku, ale nelze je do sebe vnořovat.

Lexikální symboly – atomy

V jazyku MiniC# existuje několik druhů lexikálních symbolů, které tvoří základní atomy jazyka.

• Identifikátory

Identifikátor začíná znakem `_` (podtržítko) nebo písmenem. Na dalších místech mohou být číslice, písmena a některé další znaky, které bývají povoleny v identifikátorech, např. `_`. Identifikátor nemusí obsahovat jen znaky z anglické klávesnice, ale může obsahovat i národní znaky. Příklady identifikátorů jsou např. `i`, `j1`, `_temp` nebo `součet_12`.

• Klíčová slova

Klíčové slovo je sekvence znaků Unicode, které vypadá jako identifikátor, ale má speciální využití v jazyce a nemůže být použito jako identifikátor. Jazyk MiniC# obsahuje následující klíčová slova:

```
abstract  as        base      bool      break
byte      case      char      class     const
continue  decimal   default   do        double
else      enum      float     for       foreach
if        in        int       is        long
new       object    out       override  params
private   protected public    readonly  ref
return    short     static    string    switch
this      virtual   void      while     true
false     null
```

• Booleovské literály

Jazyk MiniC# obsahuje dva booleovské literály `true` (pravda) a `false` (nepravda), jež odpovídají klíčovým slovům. Typem (viz dále) literálu je typ `bool`.

• Celočíselné literály

Celá čísla lze zapsat ve zdrojovém programu pomocí dvou forem celočíselných literálů: v desítkové soustavě a v šestnáctkové soustavě. Celočíselné literály slouží pro zápis konstant celočíselných typů `byte`, `short`, `int` a `long`.

Desítkový celočíselný literál je složen alespoň z jednoho znaku `0` až `9`, po kterých může následovat přípona `l` nebo `L` pro určení rozsahu čísla. Šestnáctkový celočíselný literál je složen z předpony `0x` nebo `0X` následované alespoň jedním znakem šestnáctkových číslic, tedy `0` až

9, A až F nebo a až f. Na konci může být opět přípona l nebo L pro určení, že typ literálu je long.

Hodnota literálu je určena obvyklým způsobem. Typ celočíselného literálu může být typ int (32 bitů) nebo long (64 bitů). Pokud je literál uveden na místě, kde je vyžadován např. typ byte, kompilátor provede automaticky konverzi.

• Literály reálných čísel

Literály reálných čísel slouží pro zápis konstant typů float, double a decimal. Tyto literály se skládají z celé části (posloupnost znaků 0 až 9), desetinné tečky ., desetinné části (posloupnost znaků 0 až 9), exponentu (znak E nebo e následovaný hodnotou exponentu) a přípony f, F, d, D, m, M. Všechny části mohou být v určitých situacích vynechány.

Hodnota literálu je určena obvyklým způsobem. Typ tohoto literálu je určen následujícím způsobem: Jestliže přípona není uvedena, typem literálu je typ double. Je-li přípona f nebo F, typem literálu je typ float. Je-li přípona d nebo D, typem literálu je typ double. Je-li přípona m nebo M, typem literálu je typ decimal. Není-li možné hodnotu literálu reprezentovat určeným typem, nastane chyba při překladu.

Příklady literálů reálných čísel jsou např.

10.0 (double), 10f (float), 15.465m (decimal), 1E-5 (double), 1.45e-5m (decimal).

• Znakové literály

Znakové literály reprezentují jednotlivé Unicode znaky. Typem tohoto literálu je typ char.

Znakový literál se skládá z dvojice apostrofů, mezi nimiž je uzavřen jediný znak, nebo úniková sekvence. Znakem může být libovolný znak, kromě apostrofu a znaků konce řádku. Únikové sekvence jsou následující:

Úniková sekvence	Název znaku	Hexadecimálně
\’	apostrof	0x0027
\"	uvozovka	0x0022
\\	obrácené lomítko	0x005C
\0	prázdný znak	0x0000
\a	zvonek	0x0007
\b	klávesa zpět	0x0008
\f	nová stránka	0x000C
\n	nový řádek	0x000A
\r	návrat vozíku	0x000D
\t	horizontální tabulátor	0x0009
\v	vertikální tabulátor	0x000B
\x*HHHH*	znak s hexadecimálním kódem	0x*HHHH*
\u*HHHH*	znak s Unicode kódem	0x*HHHH*

Příklady literálů reálných čísel jsou např.

’a’, ’*’, ’\’’, ’\r’, ’\xF’, ’\uFC8A’.

• Řetězcové literály

MiniC# podporuje dvě formy řetězcových literálů: obvyklý řetězcový literál a doslovný řetězcový literál.

Obvyklý řetězcový literál je složen z dvojice znaků " (uvozovka), mezi nimiž je řetězec znaků nebo únikových sekvencí. Význam únikových sekvencí je stejný jako u znakových literálů.

Doslovný řetězcový literál začíná znakem `@`, následovaným uvozovkou. Následuje posloupnost libovolných Unicode znaků, včetně mezer a konců řádků, jež jsou interpretovány doslovně. Jedině dvojice znaků `""`, představuje únikovou sekvenci pro uvozovku. Na konci je znak uvozovka `"`. Únikové sekvence v doslovném literálu nejsou zpracovávány. Doslovný řetězcový literál může zabírat více řádků.

Příklady řetězcových literálů jsou např.

```
"Jazyk MiniC#",
"Jazyk \r\n MiniC#",
"C:\\dokumenty\\CSharp",
@"C:\dokumenty\CSharp",

@"První řádek
Druhý řádek

Čtvrtý řádek"
```

• Operátory a znaménka

Operátory jsou používány ve výrazech, které zahrnují více operandů. Znaménka slouží jako spojovače nebo oddělovače. Jazyk MiniC# obsahuje následující operátory a znaménka:

```
{    }    [    ]    (    )    .    ,    :    ;
+    -    *    /    %    &    |    ^    !    ~
=    <    >    ?    ++   --   &&   ||   <<   >>
==   !=   <=   >=   +=   -=   *=   /=   %=   &=
|=   ^=   <<=  >>=
```

2.4. Syntaktická struktura programu

Zdrojový text programu

MiniC# dovoluje uchovávat zdrojový text programu ve více souborech. Při překladu jsou všechny zdrojové soubory zpracovávány dohromady, zdrojové soubory mohou volně referencovat ostatní soubory. Toto chování je stejné, jako by se zpracovával jediný zdrojový soubor vzniklý zřetězením všech zdrojových souborů. Předsunuté deklarace nejsou v MiniC# potřeba, neboť pořadí deklarací je, až na několik výjimek, bezvýznamné.

Základními koncepty programu v jazyce MiniC# jsou typy a jejich členy. Typy se definují pomocí deklarací. Program může obsahovat deklarace pouze pro výčtové typy a třídy. To ovšem nic nemění na tom, že je možno používat předdefinované hodnotové a odkazové typy.

Deklarace tříd mohou obsahovat tzv. funkční členy (metody, vlastnosti, konstruktory), uvnitř jejich těl se vyskytují příkazy jazyka, jež vyjadřují akce prováděné funkčním členem. Jazyk MiniC# obsahuje poměrně velké množství příkazů.

Součástí příkazů jsou výrazy, jež představují prováděné výpočty. Výrazy se skládají z operátorů a operandů. Operandy mohou být jednoduché výrazy (jména, literály) nebo další složené výrazy.

Vstupní bod programu

Běh programu začíná aktivací určené metody, tzv. vstupního bodu programu. Vstupním bodem programu může být metoda z libovolné třídy s jednou z následujících hlaviček:

```
static void Main()
static void Main(string[] args)
static int Main()
static int Main(string[] args)
```

2.5. Třídy

Třídy jsou nejpodstatnější typy v jazyku MiniC#. Obsahují datové členy a funkční členy v jedné struktuře. Navíc umožňují používat v plné míře dědičnost a polymorfismus. Třídy poskytují definici pro dynamicky vytvářené instance – objekty.

Třídy vznikají pomocí deklarace třídy. Deklarace třídy se skládá z hlavičky a z těla třídy. Hlavička třídy obsahuje klíčové slovo `class`, před nímž může být modifikátor abstraktní třídy `abstract`. Dále obsahuje identifikátor určující jméno třídy a dále volitelně dvojtečku a jméno základní třídy. Tělo třídy se skládá z deklarací členů třídy (viz dále) uzavřených mezi složenými závorkami.

[modifikátor] `class` *jméno-třídy [*`:` *základní-třída]*
`{` *seznam-deklarací-členů* `}`

Jestliže základní třída není uvedena, základní třídou deklarované třídy se stává třída `object`. Základní třídou nemohou být předdefinované třídy `Array`, `Enum`, nebo `string`. Dědičná hierarchie tříd nesmí obsahovat definici kruhem. Tedy pokud třída `A` dědí z třídy `B`, třída `B` nemůže (ani nepřímo) dědit z třídy `A`.

Třída může obsahovat následující druhy členů:

- Konstanty – konstantní hodnoty asociované s třídou.
- Datové položky – hodnoty tvořící stav objektu, nebo hodnoty asociované s třídou.
- Metody – výpočty a akce, které mohou být prováděny objekty, nebo jsou asociované s třídou.
- Vlastnosti – akce umožňující přístup k charakteristikám (vlastnostem) objektů, nebo asociovaných s třídou.
- Konstruktory – akce potřebné pro inicializaci objektů nebo třídy.

Všechny druhy členů mají přiřazenu svoji přístupnost, která určuje oblasti programu, kde je člen přístupný k použití. Existují tři možné přístupnosti:

- `public` – přístup není nijak omezen.
- `protected` – přístup je povolen jen v rámci deklarace třídy a v třídách zděděných z této třídy.
- `private` – přístup je povolen jen v rámci deklarace třídy.

V deklaracích členů se mohou vyskytovat různé modifikátory. Vždy platí, že žádný modifikátor nesmí být uveden vícekrát. Modifikátory mohou obsahovat jen jeden z modifikátorů přístupu `public`, `protected`, `private`. Pokud žádný modifikátor přístupu není uveden, člen má přístupnost `private`.

Konstanty

Konstanty jsou členy, jež reprezentují konstantní hodnoty asociované s třídou, které lze spočítat v době překladu.

Jedna deklarace může definovat více konstant se stejným typem a modifikátorem a má následující tvar:

[modifikátor] `const` *typ jméno*$_1$`=`*výraz*$_1$ *[*`,...,` *jméno*$_n$`=`*výraz*$_n$*]*`;`

Modifikátorem může být jeden z modifikátorů přístupnosti. Ačkoliv je konstanta statický člen, není povolen modifikátor `static`.

Typ konstanty není nijak omezen. Avšak výraz určující hodnotu konstanty musí být konstantní výraz, tedy určení jeho hodnoty je možné již při překladu.

Datové položky

Datové položky jsou členy, jež reprezentují stav objektu, nebo (pokud jsou statické) reprezentují hodnoty asociované s třídou.

Jedna deklarace může deklarovat více datových položek se stejným typem a modifikátory a má následující tvar:

[modifikátory] typ jméno$_1$*[*`=`*inicializátor*$_1$*]* *[*`,...,` *jméno*$_n$*[*`=`*inicializátor*$_n$*]]*`;`

Seznam modifikátorů může obsahovat jeden z modifikátorů přístupnosti a modifikátory `static` a `readonly`.

Inicializátor je nepovinný, ale pokud je uveden specifikuje počáteční hodnotu datové položky. Inicializátorem může být buď libovolný výraz, nebo inicializátor pole. Typ inicializátoru musí být implicitně konvertovatelný na typ položky.

Statické a instanční datové položky

Jestliže deklarace datové položky obsahuje modifikátor `static`, jedná se o statickou datovou položku. Pokud deklarace tento modifikátor neobsahuje, jedná se instanční datovou položku. Statické položky nejsou součástí instancí třídy, reprezentují právě jedno místo v paměti běžícího programu. Naopak instanční položky jsou vždy spojeny s instancemi třídy. Každý objekt obsahuje svoji sadu instančních položek třídy.

Datové položky jen pro čtení

Pokud deklarace datové položky obsahuje modifikátor `readonly`, jedná se o tzv. položku jen pro čtení. Přiřazení hodnoty do položky jen pro čtení je povoleno pouze v deklaraci inicializátorem nebo v konstruktoru. Položky jen pro čtení je možno používat podobně jako konstanty. Hodnota se jim však přiřazuje až za běhu programu. Jejich hodnoty tedy mohou být i dynamicky alokované objekty. Navíc hodnotu je možné nastavit např. pomocí parametrů konstruktoru.

Metody

Metody implementují výpočty nebo akce, jež mohou být prováděny objekty, nebo (v případě statických metod) asociované s třídou.

Deklarace metody má následující tvar:

[modifikátory] návratový-typ jméno `(` *[seznam-formálních-parametrů]* `)`
tělo-metody

Seznam modifikátorů může obsahovat jeden z modifikátorů přístupnosti a modifikátory `static`, `virtual`, `override`, `abstract`. Povoleny jsou jen některé kombinace modifikátorů. Návratový typ metody určuje typ hodnoty počítané a navrácené metodou. Pokud metoda nevrací hodnotu, návratovým typem je klíčové slovo `void`. Seznam formálních parametrů může být prázdný.

Tělo metody je tvořeno u abstraktních metod pouze středníkem, u ostatních metod blokem příkazů (viz sekce 2.10.).

Jméno metody a seznam jejích formálních parametrů tvoří *signaturu* metody. Jméno metody musí být odlišné od jmen všech ostatních členů třídy, které nejsou metody. Signatura metody musí být odlišná od všech ostatních signatur metod se stejným jménem.

Formální parametry metody

Formální parametry se používají pro předání aktuálních hodnot argumentů při aktivaci metody. Existují čtyři druhy formálních parametrů – *hodnotové* parametry, *odkazové* parametry, *výstupní* parametry a *pole parametrů*. Formální parametry jsou v seznamu odděleny čárkami. Pole parametrů může být uvedeno jen na konci seznamu.

Každé vyvolání metody si vytváří svoji vlastní kopii parametrů.

- **Hodnotové parametry** odpovídají lokálním proměnným metody s počáteční hodnotu danou při aktivaci metody hodnotou příslušného argumentu. Deklarace v rámci seznamu formálních parametrů vypadá takto:

  typ jméno

- **Odkazové parametry** nevytvářejí novou proměnnou, ale reprezentují proměnnou předanou jako příslušný argument. Deklarace v rámci seznamu formálních parametrů vypadá takto:

  `ref` *typ jméno*

  Proměnná předávána jako odkazový parametr musí být v místě aktivace metody definitivně přiřazena (viz sekce 2.1.).

- **Výstupní parametry** také nevytvářejí novou proměnnou a opět reprezentují proměnnou předanou jako příslušný argument. Deklarace v rámci seznamu formálních parametrů vypadá takto:

  `out` *typ jméno*

  Na rozdíl od odkazového parametru, proměnná předávána jako výstupní parametr nemusí být v místě aktivace metody definitivně přiřazena. Výstupní parametr musí být definitivně přiřazen před ukončením metody.

- **Pole parametrů** umožňuje předat metodě libovolný počet parametrů. Deklarace je následující:

  `params` *typ-pole jméno*

  Typ tohoto druhu parametru musí být jednodimenzionální pole. Uvnitř metody se tento parametr chová jako běžný parametr s daným typem pole. Při aktivaci metody je ovšem možné místo jediného parametru uvést libovolný počet argumentů, jejichž typ je konvertovatelný na typ prvků pole.

Statické a instanční metody

Jestliže deklarace metody obsahuje modifikátor `static`, potom je metoda statická. V opačném případě je metoda instanční. Samotnou instanci, nad níž je instanční metoda prováděna je možno získat uvedením klíčového slova `this`.

Virtuální metody

Instanční metody mohou být virtuální. Metoda je virtuální, pokud její deklarace obsahuje modifikátor `virtual`. Při vyvolání virtuální metody rozhoduje skutečný (za běhu programu) typ instance, nad níž je volání prováděno, která implementace metody bude provedena. U nevirtuálních metod je metoda pro provedení jednoznačně určena již v době překladu na základě typu výrazu uvedeného v programu, který představuje instanci.

Přepsané (override) metody

Virtuální metoda může být ve zděděné třídě přepsána (potlačena, overridden). Jestliže deklarace metody obsahuje modifikátor `override`, pak tato metoda přepisuje (potlačuje) virtuální metodu se stejnou signaturou v některé ze základních tříd. Pokud tato tzv. základní metoda neexistuje, nebo její hlavička plně neodpovídá hlavičce přepsané metody, nastane chyba při překladu. Zatímco virtuální metoda vytváří novou metodu, přepsaná metoda specializuje virtuální metodu poskytnutím nové implementace.

Abstraktní metody

Instanční metoda, jejíž deklarace obsahuje modifikátor `abstract` je abstraktní metoda. Abstraktní metoda je virtuální metoda, která ovšem neposkytuje počáteční implementaci. Z tohoto důvodu tělo abstraktní metody tvoří středník. Abstraktní metody se mohou nacházet pouze v abstraktních třídách. Zděděná třída, která není abstraktní, musí přepisovat všechny zděděné abstraktní metody.

Přetěžování (overloading) metod

Přetěžování metod umožňuje v deklaraci třídy uvést několik metod se stejným jménem, pokud mají navzájem jiné signatury. Při překladu aktivace metody s daným jménem kompilátor vybere nejvhodnější metodu, nebo pokud nelze nejvhodnější metoda nalézt, nastane chyba při překladu. Nejvhodnější metoda je vybírána tak, aby si nejlépe odpovídaly formální parametry metody a aktuální parametry uvedené při aktivaci.

Vlastnosti

Vlastnosti jsou členy tříd, jež poskytují přístup k charakteristikám objektů nebo tříd. Vlastnosti jsou přirozeným rozšířením datových položek. Oboje jsou pojmenovanými členy a syntaxe pro přístup k nim je stejný. Nicméně vlastnosti neoznačují místa v paměti. Namísto toho vlastnosti obsahují *akcesory*, jež určují příkazy, které se mají provést při čtení nebo zapisování vlastnosti.

Deklarace vlastnosti je následující:

[modifikátory] typ jméno
`{`
  `get` *tělo-akcesoru*$_1$
  `set` *tělo-akcesoru*$_2$
`}`

Seznam modifikátorů může obsahovat jeden z modifikátorů přístupnosti a modifikátory `static`, `virtual`, `override`, `abstract`. Povoleny jsou jen některé kombinace modifikátorů, a to stejně jako u metod. Pořadí akcesorů `get` a `set` může být i opačné, deklarace taktéž nemusí obsahovat oba akcesory, vždy ale alespoň jeden. Na začátku akcesorů jsou vyžadovány identifikátory `get` nebo `set`, ačkoliv to nejsou klíčová slova. U abstraktních vlastností těla akcesorů tvoří pouze středník, v ostatních případech blok příkazů (viz sekce 2.10.).

Akcesor `get` (getter) odpovídá bezparametrické metodě, jejíž návratový typ je stejný jako typ vlastnosti. Getter je volán při získávání hodnoty vlastnosti.

Akcesor `set` (setter) odpovídá metodě s jedním hodnotovým parametrem se jménem `value`, jehož typ je stejný jako typ vlastnosti. Tento akcesor nemá návratovou hodnotu. Setter je volán s argumentem, jež představuje novou hodnotu vlastnosti.

Statické a instanční vlastnosti

Podobně jako metody a datové položky i vlastnosti mohou být statické a instanční. Vlastnost, jejíž modifikátory obsahují modifikátor `static` je statická, v opačném případě je instanční.

Virtuální, přepsané a abstraktní vlastnosti

Stejně jako metody mohou být i vlastnosti virtuální, přepsané a abstraktní, jestliže jejich deklarace obsahuje modifikátory `virtual`, `override`, nebo `abstract`. Mechanismus virtuálních metod se přenáší z vlastnosti na její akcesory se stejným chováním jako metody. Při přepisování vlastnosti ze základní třídy, která má oba akcesory je možné přepisovat pouze jeden z nich, nebo oba dva.

Instanční konstruktory

Instanční konstruktory jsou členy třídy, které implementují akce potřebné pro inicializaci instance třídy. Deklarace instančního konstruktoru je následující:

[modifikátor] jméno-třídy `(` *[seznam-formálních-parametrů]* `)` *[inicializátor] tělo-konstruktoru*

Modifikátorem může být jeden z modifikátorů přístupnosti. Jméno konstruktoru musí být stejné jako jméno obsahující třídy. Seznam formálních parametrů může být prázdný a má stejný tvar i význam jako u metod. Tělo konstruktoru je tvořeno blokem příkazů.

Inicializátor konstruktoru nemusí být vždy uveden, ale pokud uveden je, má jeden z následujících tvarů:

`: base (`*[seznam-argumentů]* `)`

nebo

`: this (` *[seznam-argumentů]*`)`

Inicializátor konstruktoru určuje jiný instanční konstruktor a jeho argumenty, který se má vyvolat před provedením těla konstruktoru. Daným konstruktorem může být konstruktor ze stejné třídy (se slovem `this`) nebo z přímé základní třídy (se slovem `base`). Pokud inicializátor není uveden, překladač automaticky přidává volání bezparametrického konstruktoru základní třídy. Překladač hledá nejvhodnější konstruktor podobně jako u přetížených metod pomocí uvedených argumentů. Pokud žádný odpovídající konstruktor není nalezen, nastává chyba při překladu.

Seznam argumentů může být prázdný. Pokud není prázdný, obsahuje čárkami oddělené argumenty. Jejich popis je uveden u výrazu volání metody (viz sekce 2.9.).

Deklarace třídy, podobně jako u metod, nemůže obsahovat dva instanční konstruktory se stejnou signaturou. Instanční konstruktory se nedědí ze základních tříd.

Implicitní konstruktor

Pokud třída nedeklaruje žádný instanční konstruktor, tzv. implicitní konstruktor je automaticky vygenerován. Implicitní konstruktor nemá parametry a pouze volá bezparametrický konstruktor přímé základní třídy. Jestliže třída je abstraktní, pak implicitní konstruktor obsahuje modifikátor `protected` v opačném případě modifikátor `public`.

Statický konstruktor

Statický konstruktor je člen třídy, který obsahuje akce potřebné pro inicializaci třídy. Třída může deklarovat nejvýše jeden statický konstruktor a jeho deklarace má vždy tvar:

`static` *jméno-třídy* `( )` *tělo-konstruktoru*

Statický konstruktor vždy obsahuje modifikátor `static` a nemůže obsahovat žádný modifikátor přístupnosti, neboť tento konstruktor nelze explicitně volat. Statický konstruktor je volán buď při vytváření prvního objektu třídy, nebo při prvním přístupu k libovolnému statickému členu třídy. Jméno statického konstruktoru musí být opět stejné jako jméno obsahující třídy. Tělo konstruktoru tvoří blok příkazů.

2.6. Pole

Pole jsou datové struktury obsahující daný počet proměnných, ke kterým se přistupuje pomocí indexů. Všechny proměnné mají stejný typ a nazývají se *elementy* pole. Typ elementů může být jakýkoliv typ, kromě polí. Pole jsou odkazové typy, tudíž jsou vždy alokovány na hromadě programu.

Každé pole má svoji dimenzi, která určuje počet indexů pro přístup k prvkům. Dimenze je součásti typu polí. Pole tedy mohou být jednodimenzionální a vícedimenzionální.

Každá dimenze pole má asociovanou délku, která se samozřejmě nezáporná. Délky dimenzí nejsou součásti typu pole, ale jednotlivých instancí a jsou určeny při vzniku pole za běhu

programu. Délky dimenzí jsou po celý život instance pole neměnné. Indexy v jednotlivých dimenzích jsou vždy v rozmezí 0 až délka dimenze-1.

Typy polí se zapisují následujícím způsobem:

typ-elementů `[ ,,, ]`

Počet čárek uvnitř hranatých závorek zvýšený o jedna určuje dimenzi pole. Např. jednodimenzionální pole s elementy typu `int` se zapíše `int[]`, třídimenzionální pole typu `string` se zapíše `string[,,]`.

Pole jsou vytvářeny pomocí výrazů s operátorem `new`, k prvkům pole se přistupuje pomocí výrazu přístupu k prvku pole. Oba výrazy viz sekce 2.9.

Pole nelze explicitně deklarovat jako třídy nebo výčty, neboť mají pevně definovanou strukturu. Pole deklaruje kompilátor automaticky při použití. Každý typ pole automaticky dědí z třídy `Array`, a tudíž přebírá její členy.

Inicializátor pole

V deklaraci datové položky, v deklaraci lokální proměnné (viz sekce Příkazy 2.10.) nebo ve výrazu vytvoření pole je možné uvést inicializátor pole, který specifikuje počáteční hodnoty všech elementů pole. Inicializátor má následující tvar.

`{` *seznam-inicializátorů-proměnných* `}`

Inicializátory proměnných jsou v seznamu odděleny čárkami. Každý inicializátor proměnné je buď libovolný výraz, nebo v případě vícedimenzionálních polí, vnořený inicializátor pole. Deklarace nebo výraz vytvoření pole, v němž se inicializátor nachází, určuje typ elementů a dimenzi pole pro inicializátor.

Pro jednodimenzionální pole musí inicializátor obsahovat seznam výrazů, které lze přiřadit do proměnné typu elementů pole. Počet výrazů v inicializátoru určuje délku dimenze pro instanci. Inicializátor pro pole typu `int[]` může vypadat takto:

```
{0, 2, 4, 6, 8}
```

Pro vícedimenzionální pole musí inicializátor obsahovat tolik úrovní vnoření inicializátorů, kolik je dimenzí pole. Délku každé dimenze určuje počet prvku v dané úrovni zanoření. Pro vnořené inicializátory na stejné úrovní vnoření musí platit, že mají stejný počet prvků. Následující inicializátor inicializuje pole s typem `int[,]` s délkami dimenzí 5 a 2.

```
{{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}}
```

2.7. Výčtové typy

Výčtové typy (výčty) jsou samostatné hodnotové typy, které deklarují množinu pojmenovaných (celočíselných) konstant, jejichž typ je typ deklarovaného výčtu. Hodnoty, jež může proměnná s výčtovým typem nabývat nejsou omezeny jen na deklarované hodnoty, ale mohou to být všechny hodnoty tzv. podložního (underlying) typu `int`.

Výčtové členy automaticky dědí z předdefinované třídy `Enum` a tudíž přebírají její členy.

Výčtové typy vznikají pomocí jejich deklarací. Deklarace výčtu se skládá z hlavičky a z těla výčtového typu. Hlavička obsahuje klíčové slovo `enum` následované identifikátorem, který určuje jméno třídy. Tělo výčtu se skládá z deklarací členů výčtu (viz dále) uzavřených mezi složenými závorkami.

`enum` *jméno-výčtu*
`{` *seznam-deklarací-členů* `}`

Seznam deklarací členů může být prázdný. Pokud není prázdný, obsahuje deklarace členů výčtu oddělené čárkami.

Členy výčtů

Členy výčtu představují pojmenované konstanty výčtového typu. Každý člen má asociovanou konstantní hodnotu. Typem této konstanty je podložní typ `int`. Deklarace členu typu je následující:

jméno [= konstantní-výraz]

Žádné dva členy nemohou mít stejné jméno. Více členů může sdílet stejnou konstantní hodnotu. Pokud v deklaraci není uveden výraz určující konstantní hodnotu, hodnotu přiřadí překladač automaticky následovně:

- Pokud je daný člen prvním deklarovaným členem ve výčtu, jeho konstantní hodnota bude 0.
- Pokud daný člen není první, jeho hodnota bude rovna hodnotě předchozího členu zvýšené o 1.

Následující kousek kódu ukazuje deklaraci výčtového typu:

```
enum Color
{
  Red,        // hodnota 0
  Green = 5,  // hodnota 5
  Blue,       // hodnota 6
  Max = Blue  // hodnota 6
}
```

2.8. Konverze

Výrazy v jazyce MiniC# mají svůj typ (z hlediska typového systém). Konverze umožňují, aby výraz jednoho typu mohl být použit tam, kde se očekává výraz jiného typu. Konverze se dělí na *implicitní* a *explicitní*. Je-li konverze implicitní, pak nemusí být v programu přímo (explicitně) uvedena a konverzi provede kompilátor. Explicitní konverze umožňuje v programu konvertovat i výrazy, jež není kompilátor schopen implicitně konvertovat.

Implicitní konverze

Následující typy konverzí jsou prováděny implicitně:

- **Identická konverze**

  Identická konverze konvertuje výraz jednoho typu do toho samého typu. Má-li výraz požadovaný typ, pak je do tohoto typu konvertovatelný.

- **Implicitní numerické konverze**

  Výrazy následujících numerických typů lze implicitně konvertovat do cílových typů:

  - Z typu `byte` do typů `short`, `int`, `long`, `float`, `double`, `decimal`.
  - Z typu `short` do typů `int`, `long`, `float`, `double`, `decimal`.
  - Z typu `int` do typů `long`, `float`, `double`, `decimal`.
  - Z typu `long` do typů `float`, `double`, `decimal`.
  - Z typu `char` do typů `int`, `long`, `float`, `double`, `decimal`.
  - Z typu `float` do typu `double`.

  Konverze z typu `int` a `long` do typu `float` a z typu `long` do typu `double` mohou ztratit přesnost. Implicitní numerické konverze ovšem nikdy neztrácejí informaci.

- **Implicitní výčtová konverze**

  Výraz, který odpovídá literálu `0`, lze konvertovat do libovolného výčtového typu.

- **Implicitní konverze odkazových typů**

  Výrazy následujících odkazových typů lze implicitně konvertovat do cílových typů.

  - Z každého odkazového typu do typu `object`.
  - Z každé třídy `S` do třídy `T`, pokud `S` dědí z `T`.
  - Z každého pole do typu `Array`.
  - Výraz `null` do jakéhokoliv odkazového typu.

  Tento typ konverze může změnit typ odkazu, ale nikdy nemění ani typ, ani hodnoty referencovaného objektu.

- **Boxování (boxing)**

  Boxování umožňuje výraz jakéhokoliv hodnotového typu konvertovat na typ `object`. Konverze probíhá tak, že je na haldě programu alokován nový objekt a do něj je zkopírována hodnota výrazu.

- **Implicitní konverze konstantních výrazů**

  Konstantní výraz typu `int` je možné konvertovat na typ `byte` nebo `short`.

Explicitní konverze

Explicitní konverze jsou povoleny ve výrazu přetypování. Explicitní konverze jsou takové konverze, které nemusejí vždy uspět, nebo hodnoty výrazů při nich mohou ztrácet informaci, nebo konvertují natolik rozdílné typy, že si zaslouží explicitní notaci.

Následující typy konverzí je možné provést explicitně:

- Všechny implicitní konverze.
- **Explicitní numerické konverze** Explicitní numerické konverze jsou konverze mezi numerickými typy, pro něž neexistuje implicitní konverze:

  - Z typu `byte` do typu `char`.
  - Z typu `short` do typů `byte` nebo `char`.

- Z typu `int` do typů `byte`, `short` nebo `char`.
- Z typu `long` do typů `byte`, `short`, `int` nebo `char`.
- Z typu `char` do typů `byte` nebo `short`.
- Z typu `float` do typů `byte`, `short`, `int`, `long`, `char` nebo `decimal`.
- Z typu `double` do typů `byte`, `short`, `int`, `long`, `char`, `float` nebo `decimal`.
- Z typu `decimal` do typů `byte`, `short`, `int`, `long`, `char`, `float` nebo `double`.

Explicitní numerické konverze mohou ztrácet informace čísel. Při konverzi z většího celočíselného typu vždy dochází k ořezání hodnoty na určený počet bitů. Při konverzi z typů `float` a `double` do celočíselných typů dochází k zaokrouhlení. Při konverzi z typu `double` do typu `float` dochází k zaokrouhlení na nejbližší hodnotu typu `float`. Při konverzi z typů `float` nebo `double` do typu `decimal`, je zdrojová hodnota převedena do reprezentace `decimal` a zaokrouhlena na nejbližší číslo s přesností 28 desetinných míst.

- **Explicitní výčtové konverze**

  Explicitní výčtové konverze jsou následující:

  - Z typů `byte`, `short`, `int`, `long`, `char`, `float`, `double`, `decimal` do jakéhokoliv výčtového typu.
  - Z jakéhokoliv výčtového typu do typů `byte`, `short`, `int`, `long`, `char`, `float`, `double` nebo `decimal`.
  - Z jakéhokoliv výčtového typu do jiného výčtového typu.

  Tyto konverze probíhají tak, že hodnota výčtu je nahrazena hodnotou typu `int` a následně provedena příslušná konverze mezi typem `int` a druhým typem.

- **Explicitní konverze odkazových typů**

  Explicitní konverze odkazových typů jsou následující:

  - Z typu `object` do jakéhokoliv odkazového typu.
  - Z třídy `S` do třídy `T`, pokud `S` je základní třída `T`.
  - Z třídy `Array` do libovolného pole.

  Při této konverzi je potřeba provádět kontrolu typu objektu za běhu programu. Explicitní odkazová konverze uspěje, pokud je výraz `null`, nebo skutečný typ objektu, jež představuje operand konverze, je konvertovatelný do cílového typu pomocí implicitní odkazové konverze. Při neúspěchu konverze je vykonávání programu ukončeno chybou.

- **Odboxování (unboxing)**

  Odboxování povoluje explicitní konverzi z typu `object` na libovolný hodnotový typ. Konverze obsahuje kontrolu, zda operand je boxovaná hodnota daného hodnotového typu, v případě úspěchu je hodnota zkopírována z instance. Při neúspěchu konverze je vykonávání programu ukončeno chybou.

2.9. Výrazy

Výrazy jsou konstruovány z operandů a operátorů. Operátory určují, jakou operaci provést s operandy. Operátory jsou například `+`,`-`, `*`, `/` a `new`. Operandy mohou být například literály, datové položky, lokální proměnné, nebo další výrazy.

Pořadí vyhodnocování operátorů ve výrazech je dáno precedencí a asociativitou operátorů. Pokud se operand vyskytne mezi dvěma operátory, je nejdříve provedena operace s vyšší precedencí. Pokud je precedence operátoru stejná, rozhoduje asociativita operátoru – zleva nebo zprava. Například výraz `x + y * z` je vyhodnocen jako `x + (y * z)` neboť operátor `*` má vyšší prioritu než `+`. Výraz `x + y - z` je vyhodnocen jako `(x + y) - z` neboť `+` a `-` mají stejnou precedenci, ale jsou levě asociativní. Naopak výraz `x = y = z` je vyhodnocen jako `x = (y = z)`, neboť operátor `=` je asociativní zprava. Všechny binární operátory kromě přiřazení jsou asociativní zleva. Unární operátory a podmíněný výraz jsou asociativní zprava.

Následující části popisují druhy výrazů jazyka MiniC# v pořadí od nejvyšší priority:

Primární výrazy

Literály

Všechny literály uvedené v lexikální struktuře (2.3.) jsou primární výrazy.

Jednoduché jméno (identifikátor)

Jednoduché jméno je složeno pouze z identifikátoru. Význam jména je určen postupným hledáním deklarací se stejným jménem v aktuálním bloku příkazů, v nadřazených blocích, deklaraci funkčního členu, vyhledáním členu v obsaženém typu, nebo jména deklarovaného typu.

Uzávorkovaný výraz

Uzávorkovaný výraz tvaru `(` *výraz* `)` se používá pro potlačení pravidel precedence a asociativity operátorů. Výraz uvnitř závorek nesmí představovat typ nebo metodu.

Přístup k členu

Přístup k členu může mít jeden z následujících tvarů:

primární-výraz `.` *identifikátor*
předdefinovaný-typ `.` *identifikátor*
`base` `.` *identifikátor*

Předdefinovaným typem je jeden z typů `bool`, `byte`, `char`, `decimal`, `double`, `float`, `int`, `long`, `object`, `short`, `string`.

Klíčové slovo `base` slouží k přístupu k překrytým nebo přepsaným členům základní třídy.

Člen, jež je představován výrazem je určen mechanismem výběru členu (viz níže) v rámci typu, který přestavuje část výrazu před tečkou nebo v rámci typu výrazu před tečkou. Pokud část výrazu před tečkou určuje typ a ne hodnotu nebo proměnnou, potom vybraný člen musí být statický, v opačném případě musí být instanční.

Pokud je při běhu programu primární výraz vyhodnocen na `null`, k členu nelze přistoupit a program je ukončen chybou.

Výběr členu je proces, kterým je určován význam jména v rámci některého typu. Výběr členu nastává jako část vyhodnocení výrazů přístup k členu a jednoduché jméno. Výběr členu se jménem `N` v typu `T` probíhá následovně:

1. Získá se množina přístupných členů se jménem `N` deklarovaných v `T` a jeho základních typech. Členy s modifikátorem `override` jsou vynechány. Pokud je množina prázdná, člen se jménem `N` neexistuje.
2. Členy, které jsou skryté se taktéž vynechají.
3. Jestliže množina obsahuje jeden člen, který není metoda, pak tento člen je výsledkem výběru. Jinak, jestliže množina obsahuje pouze metody, výsledkem je skupina těchto metod. Jiný případ nastat nemůže.

Volání metody

Výraz volání metody má jeden z následujících tvarů:

přístup-k-členu `(` *[seznam-argumentů]* `)`
jednoduché-jméno `(` *[seznam-argumentů]* `)`

První část výrazů musí určovat skupinu metod, jinak nastane chyba překladu.

Pokud seznam argumentů není prázdný, pak se skládá z jednoho nebo více argumentů oddělených čárkami. Každý argument má jednu z následujících forem:

- *výraz* – argument je předáván hodnotou.
- `ref` *výraz* – argument je předáván odkazem, výraz musí určovat proměnnou.
- `out` *výraz* – argument je předáván jako výstupní, výraz musí určovat proměnnou.

Seznam argumentů poskytuje výrazy nebo proměnné pro formální parametry metody. Konkrétní metoda, jež se vyvolá, je vybrána jako nejvhodnější přetížená metoda ze skupiny metod a to podle typů a druhů argumentů. Pokud nejvhodnější metoda nelze určit, nastane chyba při překladu. Pokud návratový typ vybrané metody je `void`, tento výraz není možné použít jako součást dalšího výrazu.

Přístup k prvku pole nebo řetězce

Přístup k prvku pole nebo řetězce má následující tvar:

primární-výraz `[` *seznam-výrazů* `]`

Typ primárního výrazu musí být buď pole nebo typ `string`. Seznam výrazů představuje indexy. Odděleny jsou čárkami. Každý z těchto výrazů musí být konvertovatelný na typ `int`. Počet indexů musí odpovídat dimenzi pole, nebo musí být 1 u typu `string`.

Pokud je primární výraz při běhu programu vyhodnocen na `null`, k prvku nelze přistoupit a program končí chybou. Jestliže některý z indexů je mimo rozsah dané dimenze, program končí chybou. Výsledkem operace je hodnota uložená na daném indexu pole nebo řetězce.

Přístup k aktuální instanci `this`

Přístup k aktuální instanci se skládá z klíčového slova `this`. Typem tohoto výrazu je třída, v níž se tento výraz nachází.

Postfixová inkrementace a dekrementace `++`, `--`

Postfixová inkrementace má tvar *výraz* `++` a postfixová dekrementace má tvar *výraz* `--`. Výraz musí určovat proměnnou nebo vlastnost. Tyto operátory jsou předdefinované pro operandy typů `byte`, `short`, `int`, `long`, `char`, `float`, `double`, `decimal` a výčtových typů.

Při inkrementaci je hodnota proměnné nebo vlastnosti zvýšena o 1, při dekrementaci snížena o 1. Výsledkem výrazu je ale původní hodnota proměnné nebo vlastnosti.

Vytvoření objektu

Vytvoření objektu má následující tvar:

`new` *typ* `(` *[seznam-argumentů]* `)`

Typem může být jen třída, tato třída nesmí být abstraktní. Seznam argumentů může být prázdný a má stejný význam jako u vyvolání metody. Konstruktor, který se následně po alokaci objektu zavolá je vybrán z přístupných konstruktorů typu vzhledem k seznamu argumentů. Vybrán je nejvhodnější konstruktor podobně jako u volání metody. Výsledkem výrazu je nově vytvořený objekt.

Vytvoření pole

Vytvoření pole má jeden z následujících tvarů:

`new` *typ* `[` *seznam-výrazů* `]` *[inicializátor-pole]*
`new` *typ-pole inicializátor-pole*

V prvním tvaru typ nemůže být pole. Typ vytvořeného pole je určen uvedeným typem (typ elementů) a dimenze je určena počtem výrazů v seznamu výrazů. Inicializátor pole je popsán v sekci 2.6. o polích, v prvním tvaru není povinný. Výrazy v seznamu výrazů nemusí být konstantní, ale musí být konvertovatelné na typ `int`. Určují délky jednotlivých dimenzí.

V druhém tvaru je inicializátor pole povinný, protože určuje délky dimenzí pole.

Pokud je inicializátor uveden, jednotlivým prvkům pole jsou přiřazeny hodnoty uvedené v inicializátoru.

Výsledkem výrazu je nově vytvořený instance pole.

Unární výrazy

Unární výrazy mají vždy prefixovou notaci, tedy mají tvar *unární-výraz op*.

Unární operátory `+`, `-`, `!`, `~`

Operátor unární plus `+` je předdefinován pro operand následujících typů: `int`, `long`, `float`, `double`, `decimal`. Výsledkem operace je jednoduše uvedený operand.

Operátor unární mínus `-` je předdefinován opět pro operand následujících typů: `int`, `long`, `float`, `double`, `decimal`. Výsledkem operace je opačná hodnota operandu.

Operátor logické negace `!` je předdefinován pouze pro operand typu `bool`. Pokud je operand `true`, výsledkem je `false` a obráceně.

Operátor bitový doplněk `~` je předdefinován pro operand typů `int`, `long` a výčtových typů. Výsledkem je bitový doplněk hodnoty operandu.

Prefixová inkrementace a dekrementace ++, --

Pro tyto operátory platí to samé jako pro postfixové operátory `++` a `--` s tím rozdílem, že výsledkem těchto výrazů je nová hodnota proměnné po inkrementaci nebo dekrementaci.

Výraz změny typu

Výraz změny typu se používá pro explicitní změnu typu. Jeho tvar je následující:

`(` *typ* `)`*unární-výraz*

Unární výraz musí být explicitně konvertovatelný na uvedený typ. Pokud se jedná o explicitní odkazovou konverzi, změna typu nemusí uspět a program je ukončen chybou.

Binární výrazy

Multiplikativní binární operátory *, /,%

Operátory násobení `*`, dělení `/` a zbytek po dělení `%` jsou předdefinovány pro dvojice operandů se stejnými následujícími typy: `int`, `long`, `float`, `double`, `decimal`. Operátory jsou prováděny běžným způsobem s povoleným přetečením hodnot.

Aditivní binární operátory +, -

Operátory sčítání `+` a odčítání `-` jsou předdefinovány pro dvojice operandů se stejnými následujícími typy: `int`, `long`, `float`, `double`, `decimal` a výčtové typy. Operátory jsou prováděny běžným způsobem s povoleným přetečením hodnot.

Dále je operátor `+` používán jako operátor konkatenace řetězců typu `string`. V tomto případě musí být alespoň jeden z operandů typu `string`. Druhým operandem může být libovolný objekt. Operátor při provádění získá jeho textovou reprezentaci pomocí metody `ToString()`, která je definovaná ve třídě `object` a tedy obsahují ji všechny objekty.

Operátory bitového posunu <<, >>

Operátory bitového posunu vlevo `<<` a bitového posunu vpravo `>>` jsou předefinovány pro dvojice operandů, kde levý operand je typu `int`, nebo `long` a druhý operand je typu `int`. Levý operand představuje hodnotu a pravý operand počet bitových pozic k posunutí.

Relační operátory ==, !=, <, >, <=, >=

Výsledkem relačních operátorů je logická hodnota `true` nebo `false`.

Relační operátory rovno `==`, nerovno `!=`, menší `<`, větší `>`, menší nebo rovno `<=` a větší nebo rovno `>=` jsou předdefinovány pro dvojice operandů stejných numerických typů. Těmito typy mohou být: `int`, `long`, `float`, `double`, `decimal` a všechny výčtové typy. Operátory jsou prováděny běžným porovnáváním numerických hodnot operandů.

Operátory `==`, `!=` jsou dále předdefinovány pro dvojice operandů stejných následujících typů: `bool`, `object` a `string`. Operátor s typy `object` porovnává reference objektů, operátor s typy `string` porovnává řetězce pomocí ordinálních hodnot znaků na jednotlivých pozicích.

Operátory testování typu `is`, `as`

Binární operátor `is` je používán pro dynamické testování typu výrazů, tedy při běhu programu. Tvar výrazů s tímto operátorem je

výraz `is` *typ*

Výsledkem operátoru `is` je logická hodnota `true` nebo `false`.

Binární operátor `as` je používán pro explicitní konverzi typu výrazů. Je povolen jen pro odkazové typy. Tvar výrazů s tímto operátorem je

výraz `as` *odkazový-typ*

Typ celého výrazu je uvedený typ. Pokud je skutečný typ operandu implicitně konvertovatelný na uvedený typ, pak operace uspěje a výsledkem je operand. V opačném případě je výsledkem hodnota `null` a nedochází tak k chybě za běhu programu.

Logické operátory `&`, `^`, `|`

Logické operátory konjunkce (AND) `&`, exkluzivní disjunkce (XOR) `^` a disjunkce (OR) `|` jsou předdefinovány pro operandy celočíselných typů `int`, `long` a pro výčtových typů. Tyto operátory provádějí bitové logické operace běžným způsobem.

Dále jsou tyto operátory předdefinovány pro logické hodnoty typu `bool` opět obvyklým způsobem.

Podmíněné logické operátory `&&`, `||`

Tyto operátory jsou definovány jen pro operandy typu `bool` a jsou obdobou operátorů `&` a `|`. Jejich vlastností ovšem je, že vyhodnocují druhý operand pouze v případě nutnosti pro určení výsledku.

Ternární podmínkový operátor `?:`

Podmínkový operátor má tvar *podmínkový-výraz* `?` $výraz_1$ `:` $výraz_2$. Typ podmínkového výrazu musí být `bool`. Pokud je podmínka pravdivá, výsledkem celého výrazu je první výraz, v opačném případě je výsledkem druhý výraz. Oba uvedené výrazy musí být stejného typu, nebo jeden musí být implicitně konvertovatelný na typ druhého.

Operátory přiřazení `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`

Operátory přiřazení nastavují novou hodnotu uvedenou jako pravý operand do proměnné nebo vlastnosti uvedené jako levý operand. Operátor `=` je jednoduchý operátor přiřazení, ostatní jsou složené operátory přiřazení.

Jednoduchý operátor přiřazení má tvar *unární-výraz* `=` *výraz* a jednoduše přiřadí hodnotu pravého operandu do proměnné nebo vlastnosti, kterou představuje levý operand. Výraz na pravé straně musí být implicitně konvertovatelný na typ operandu na levé straně. Výsledkem je hodnota přiřazená levému operandu, typ výsledku je stejný jako typ levého operandu. V případě, že levý operand je vlastnost, musí mít `set` akcesor.

Složená přiřazení mají tvar *unární-výraz op*`=` *výraz*. *op* představuje jeden z operátorů `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`. Složené přiřazení tvaru x *op*`=` y je vyhodnoceno stejně jako x `=` x *op* y s tím rozdílem, že x je vyhodnoceno pouze jednou.

2.10. Příkazy

Akce programu v jazyku MiniC# jsou vyjadřovány pomocí příkazů. MiniC# nabízí množství příkazů převážně známých z jazyků C a C++.

Následující části uvádějí všechny příkazy jazyka MiniC#:

Blok

Bloky dovolují uvést více příkazů tam, kde je požadován jediný příkaz. Bloky také tvoří těla funkčních členů. Blok je složen z několika příkazů uzavřených ve složených závorkách, nebo může být prázdný:

`{` *[seznam-příkazů]* `}`

Prázdný příkaz

Prázdný příkaz nedělá nic. Používá se tam, kde je příkaz vyžadován, ale žádná akce není potřeba provést. Prázdný příkaz se skládá pouze ze středníku:

`;`

Deklarace lokálních proměnných

V rámci jednoho příkazu deklarace lokální proměnné je možno deklarovat více proměnných:

typ jméno$_1$*[=inicializátor*$_1$*]* *[*`,`... `,` *jméno*$_n$*[=inicializátor*$_n$*]]*`;`

Uvedený typ určuje typ deklarovaných proměnných. Rozsah platnosti deklarovaných proměnných je blok příkazů, ve kterém jsou deklarovány. Inicializátor může být výraz nebo inicializátor pole konvertovatelný na typ proměnné.

Deklarace lokálních konstant

V rámci jednoho příkazu deklarace lokálních konstant je možno deklarovat více konstant:

`const` *typ jméno*$_1$`=`*konstantní-výraz*$_1$ *[*`,`... `,` *jméno*$_n$`=`*konstantní-výraz*$_n$*]*`;`

Uvedený typ určuje typ deklarovaných konstant. Rozsah platnosti deklarované konstanty je blok příkazů ve kterém je deklarována. Konstantní výraz musí být konvertovatelný na typ konstanty.

Příkaz – výraz

Tento typ příkazu vyhodnotí daný výraz. Příkaz – výraz se skládá z výrazu následovaného středníkem:

výraz `;`

Ne všechny výrazy jsou v tomto příkazu povoleny. Povoleny jsou jen výrazy volání metody, vytvořená objektu, přiřazení, inkrementace a dekrementace (viz sekce 2.9.).

Podmíněný příkaz – `if`

Podmíněný příkaz vybere příkaz k provedení na základě vyhodnocení logické podmínky. Podmíněný příkaz má tvar:
`if (` *podmínkový-výraz* `)` *příkaz*$_1$ *[*`else` *příkaz*$_2$*]*

Podmínkový výraz je takový výraz, jehož typ je typ `bool`. První příkaz je proveden, pokud je podmínka vyhodnocena na hodnotu `true`. Pokud je podmínka vyhodnocena `false` a je přítomna sekce `else`, je proveden druhý příkaz. Uvedenými příkazy nemohou být jen příkazy deklarace proměnných nebo konstant.

Příkaz přepínač – `switch`

Příkaz `switch` vybere k provedení sekci příkazů uvedených v jeho těle za jedním z návěští, jež je označeno výrazem, jehož hodnota je stejná jako hodnota řídícího výrazu:

`switch (` *řídící-výraz* `)`
`{`
  `case` *konstantní-výraz*$_{11}$ `:`
  ...
  `case` *konstantní-výraz*$_{1m_1}$ `:` *seznam-příkazů*$_1$
  ...
  `case` *konstantní-výraz*$_{n1}$ `:`
  ...
  `case` *konstantní-výraz*$_{nm_n}$ `:` *seznam-příkazů*$_n$
  [`default :` *seznam-příkazů*$_{n+1}$*]*
`}`

Blok příkazu `switch` obsahuje sekce, které obsahují jedno nebo více návěští následované příkazy. Typ řídícího výrazu může být jeden z následujících typů: `byte`, `short`, `int`, `long`, `char`, `string` a libovolný výčtový typ. Konstantní výrazy ve všech návěštích musí být implicitně konvertovatelné na typ řídícího výrazu. Žádné dvě návěští nemohou obsahovat výraz se stejnou hodnotu. Sekce `default` může být uvedena nejvýše jednou.

Hodnota řídícího výrazu je postupně srovnávána s konstantními výrazy v návěštích. Jestliže je hodnota shodná, vykonají se příkazy následující za návěštím. Pokud hodnota řídícího výrazu není shodná s žádnou hodnotou uvedenou v návěštích, provedou se příkazy v sekci `default`, pokud tato sekce existuje.

V jazyku MiniC# není možné, aby vykonávání příkazů „propadlo“ do jiné sekce. Sekci příkazů je potřeba ukončit příkazem skoku `break`, `return` nebo `continue`.

Příkaz cyklu s podmínkou na začátku – `while`

Příkaz cyklu s podmínkou na začátku podmíněně cyklicky provádí daný příkaz. Cyklus `while` má následující formát:

`while (` *podmínkový-výraz* `)` *příkaz*

Příkaz je cyklicky vykonáván dokud je hodnota podmínkového výrazu rovna hodnotě `true`.

Příkazem `continue` uvedeným v rámci složeného příkazu v příkazu `while` se výpočet bezpodmínečně přesouvá k opětovnému vyhodnocení podmínky. Příkazem `break` se bezpodmínečně ukončuje vykonávání celého cyklu.

Příkaz cyklu s podmínkou na konci – `do`

Příkaz cyklu s podmínkou na konci podmíněně provádí daný příkaz, nejméně však jednou. Cyklus `do` má následující formát:

`do` *příkaz* `while (` *podmínkový-výraz* `) ;`

Příkaz je cyklicky vykonáván dokud je hodnota podmínkového výrazu rovna hodnotě `true`.

Příkazem `continue`, uvedeným v rámci složeného příkazu v příkazu `do`, se výpočet bezpodmínečně přesouvá k vyhodnocení podmínky. Příkazem `break` se bezpodmínečně ukončuje vykonávání celého cyklu.

Příkaz cyklu s řídící proměnnou – `for`

Formát příkazu for je následující:

`for (` *inicializátor-cyklu* `;` *podmínkový-výraz* `;` *iterátor* `)` *příkaz*

Inicializátor cyklu, podmínka i iterátor jsou nepovinné. Inicializátorem cyklu může být buď příkaz deklarace lokálních proměnných nebo seznam výrazů, jež mohou být příkazy, oddělených čárkami. Platnost deklarovaných proměnných sahá přes celý příkaz `for`. Iterátor je tvořen seznamem výrazů, jež mohou být příkazy, oddělených čárkami.

Příkaz `for` nejdříve jedinkrát vyhodnotí inicializátor cyklu. Následně se vyhodnotí podmínka. Pokud je její hodnota `true`, provede se uvedený příkaz. Jestliže se dosáhne konce uvedeného příkazu, provedou se výrazy v iterátoru a následuje další iterace začínající vyhodnocením podmínky. Neplatí-li podmínka příkaz je ukončen.

Příkazem `continue` uvedeným v rámci složeného příkazu v příkazu for se výpočet bezpodmínečně přesouvá na konec složeného příkazu. Příkazem `break` se bezpodmínečně ukončuje vykonávání celého cyklu.

Příkaz procházení kolekcí – `foreach`

Příkaz `foreach` prochází kolekci a pro každý element kolekce provede uvedený příkaz. Příkaz `foreach` má následující tvar:

`foreach (` *typ jméno* `in` *výraz* `)` *příkaz*

Typ a jméno představují lokální proměnnou, která je jen pro čtení a představuje aktuální element kolekce dané uvedeným výrazem. Typ výrazu kolekce musí odpovídat níže uvedenému vzoru a typ prvků kolekce musí být explicitně konvertovatelný na typ iterační proměnné.

Libovolný výraz `C` splňuje vzor pro kolekci, jestliže platí:

- Typ výrazu `C` obsahuje veřejnou instanční metodu se signaturou `GetEnumerator()`, jejíž návratový typ je třída, dále označena jako `E`.
- `E` obsahuje veřejnou instanční metodu s hlavičkou `bool MoveNext()`.
- `E` obsahuje veřejnou vlastnost se jménem `Current`, její typ je typem elementu kolekce.

Příkazem `continue` uvedeným v rámci složeného příkazu v příkazu `foreach` se výpočet bezpodmínečně přesouvá na konec složeného příkazu. Příkazem `break` se bezpodmínečně ukončuje vykonávání celého cyklu.

Příkaz ukončení cyklu nebo přepínače – `break`

Příkaz `break` ukončuje nejbližší příkaz `switch`, `while`, `do`, `for`, nebo `foreach`. Všechny následující příkazy v cyklu jsou vynechány. Příkaz `break` má tvar

`break ;`

Příkaz ukončení iterace cyklu – `continue`

Příkaz `continue` nepodmíněně začíná novou iteraci nejbližšího příkazu `while`, `do`, `for`, nebo `foreach`. Všechny následující příkazy v těle cyklu pro danou iteraci jsou vynechány. Příkaz `continue` má tvar

`continue ;`

Příkaz návratu z funkce – `return`

Příkaz `return` ukončuje vykonávání aktuálního funkčního členu (metody, akcesoru vlastnosti nebo konstruktoru) a vrací řízení programu funkci, která aktuální funkci aktivovala. Příkaz `return` má následující tvar:

`return` *[výraz]* `;`

Forma bez výrazu je povolena jen ve funkčních členech, které nemají návratový typ. Forma s výrazem je povolena jen ve funkčních členech, které mají návratový typ. Pokud je výraz uveden, musí být implicitně konvertovatelný na návratový typ funkčního členu. Hodnota výrazu se stává návratovou hodnotou funkčního členu. Následně je řízení programu vráceno volajícímu funkčnímu členu.

3. Principy konstrukce překladačů a interpretů

3.1. Struktura překladačů a interpretů

Překladače bývají tradičně děleny na několik částí.

První, hrubé, dělení rozděluje překladač na dvě části. První část, tzv. *front-end* (přední část), je závislá na zdrojovém jazyce. Tato část převede program do jisté interní formy. Druhá část, tzv. *back-end* (zadní část) je závislá na cílovém jazyce nebo architektuře a zpracovává program uvedený v interní formě. Toto dělení je výhodné zejména proto, že umožňuje využít stejnou přední část pro překladače do různých cílových architektur, použitím různých zadních částí. Ale také naopak, použít stejnou zadní část pro překladače různých jazyků do stejné cílové architektury. V podstatě stačí naprogramovat n předních částí překladačů, po jedné pro každý z n jazyků a m zadních částí pro m cílových architektur nebo jazyků, abychom získali $n \cdot m$ překladačů.

Druhé, podrobnější, dělení člení překladače na samostatné části, které zajišťují jednotlivé fáze překladu (viz část 1.4.). Tradičně těmito částmi bývají:

- Lexikální analyzátor
- Syntaktický analyzátor
- Sémantický analyzátor
- Generátor kódu
- Optimalizátor

Generátor kódu se vykytuje jen překladačů, které vytvářejí cílový program jako celek. Optimalizátor není povinnou součástí překladačů a obvykle bývá obsažen v kompilátorech jazyků pro robustní komerční aplikace. Obvykle se skládá ze samostatných optimalizačních úloh prováděných v různých fázích překladu.

Principy konstrukce částí překladačů jsou popsány v následujících sekcích.

3.2. Lexikální analyzátor

Úkolem lexikálního analyzátoru je rozpoznat atomy jazyka a vstupní program jako posloupnost textových znaků tak převést do tzv. první interní formy – posloupnosti atomů (lexikálních symbolů). Přitom lexikální analyzátor vynechává všechny další lexikální elementy jazyka, jež přímo nesouvisí s programem, tedy komentáře, mezery, konce řádků atd.

Lexikální analýza bývá vydělena ze syntaktické analýzy, neboť na popis lexikálních elementů stačí regulární gramatika, kdežto na popis celého programovacího jazyka je většinou potřeba bezkontextová gramatika. Na rozpoznání elementů tedy stačí jednodušší konečný automat, kdežto k rozpoznání celého programu je zapotřebí zásobníkový automat.

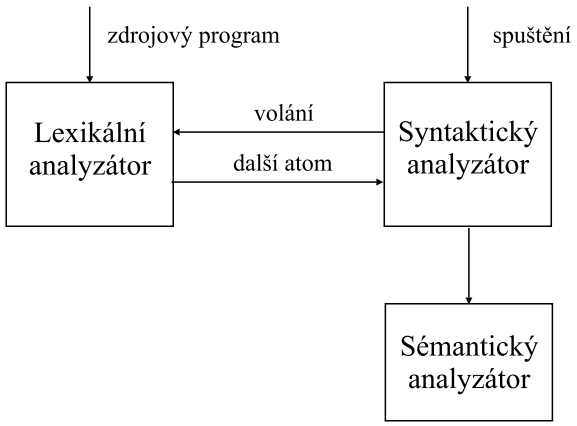
Symboly první interní formy jsou obvykle reprezentovány strukturovaným typem, jenž obsahuje typ atomu (obvykle celočíselná konstanta) a hodnotu atomu. Hodnotu nemusí mít všechny atomy, u nich zůstane hodnota nevyplněna.

Od lexikálního analyzátoru jsou požadovány následující vlastnosti:

- Jeho vstupem je zdrojový program jako posloupnost znaků.

- Analyzátor postupně čte znaky zdrojové posloupnosti.
- Jakmile další znak již nelze přidat tak, aby s předchozími znaky tvořil některý element jazyka, pak pokud element je atom (tj. není komentář, nebo mezera atd.), přidá tento nový atom do výstupní posloupnosti.
- Analyzátor nastaví svůj stav, tak aby mohl rozpoznat další element, čtení zdrojové posloupnosti tak pokračuje od posledního nezpracovaného znaku, nikoliv od začátku.
- Není-li možné z přečtených znaků sestavit žádný element, analyzátor může ukončit svoji činnost chybou, nebo chybu jen vyhlásit, znaky vynechat a pokračovat rozpoznáním dalšího symbolu.

Lexikální analyzátor bývá do překladače většinou zařazen jako podprogram (funkce). Syntaktický analyzátor tuto funkci volá vždy, když potřebuje další atom ze vstupu. To znamená, že po úspěšném rozpoznání analyzátor nový symbol nepřidává do výstupní posloupnosti, ale vrací jako výsledek volání funkce, činnost lexikálního analyzátoru je pozastavena a popsaným způsobem pokračuje až při dalším zavolání tohoto podprogramu.



Obrázek 2. Začlenění lexikálního analyzátoru do překladače

K popisu elementů jazyka se kromě regulárních gramatik používá také stručnější forma – regulární výrazy. Lexikální elementy obvykle bývají popsány regulárními gramatikami nebo výrazy každý zvlášť.

Lexikální analyzátor uvedených vlastností je možno sestavit např. následujícím postupem:

1. Z regulárních výrazů nebo gramatik jednotlivých lexikálních elementů jsou obvyklým postupem sestaveny konečné deterministické automaty.
2. Všechny konečné automaty jsou spojeny v jeden automat tak, že počáteční stavy všech automatů jsou sloučeny v jeden. Tím vznikne jeden nedeterministický konečný automat. Jeho koncové stavy odpovídají koncovým stavům původních automatů, tedy reprezentují konec rozpoznání elementů. Tyto koncové stavy se označí příslušnými lexikálními elementy.
3. Následně je z tohoto nedeterministické konečného automatu známým algoritmem sestaven deterministický automat. Jeho stavy tedy odpovídají množinám stavů původního

nedeterministického automatu. Koncový stav je takový, který obsahuje alespoň jeden koncový stav původního automatu, označení koncových stavů původních automatů se tak přenese na tento automat.

4. V kterémkoliv koncovém stavu by nyní automat mohl ukončit rozpoznání elementu jazyka, i když z tohoto stavu existují nějaké možné přechody. Pokud je ovšem požadováno, aby analyzátor vždy tvořil nejdelší možné elementy, pak je potřeba provést následující krok.

5. Je vytvořen nový stav F. Z každého koncového stavu je do nového stavu F veden přechod označený všemi znaky, pro než přechod z koncového stavu neexistuje. Tento přechod je navíc označen příslušným lexikálním elementem, čímž se indikuje rozpoznání tohoto elementu. Nakonec jsou všechny původní koncové stavy označeny jako nekoncové.

Analyzátor pracuje jako konečný deterministický automat. Pokud ovšem je prováděn přechod označený elementem jazyka, pak:

- Je-li rozpoznaný element atomem, je přidán na výstup a může dojít k rozpoznání dalšího atomu opětným zavoláním funkce lexikálního analyzátoru.
- Není-li rozpoznaný element atomem, automat ihned provede rozpoznání dalšího elementu.

Na začátku rozpoznání dalšího elementu se vždy musí automat posunout o jeden znak na vstupu zpět, neboť daný znak již byl přečten a rozhodl o tom, že předchozí znak ukončuje předchozí element.

Při lexikální analýze je u některých atomů (literály, identifikátory) potřeba určovat jejich hodnotu. Ve většině případů je zřejmé již při prvním znaku, který element se rozpoznává. V tomto případě je možné hodnotu atomu tvořit při každém přečtení dalšího znaku. V méně častých případech se musí hodnota určit až na konci rozpoznání ze všech znaků lexikálního symbolu.

V některých jazycích mají klíčová slova (samostatné atomy) stejný tvar např. jako identifikátory. V tomto případě, lexikální analyzátor je upraven tak, aby při rozpoznání identifikátoru zkontroloval, zda je klíčovým slovem. Místo identifikátoru je pak dáno na výstup dané klíčové slovo.

V průběhu lexikální analýzy může dojít k několika druhům chyb:

- Ve zdrojovém textu je znak, který se v jazyce vůbec nepoužívá.
- Některý atom je zapsán špatně a tudíž analyzátor nemůže dokončit rozpoznání, neboť v automatu analyzátoru není definován přechod pro následující znak.
- Při určování hodnoty atomu dojde například u čísel k překročení rozsahu čísla nebo ke ztrátě přesnosti.

Ve všech uvedených případech může analýza dále pokračovat, ale cílový program by neměl být vytvořen. V prvních dvou případech jsou nesprávné znaky prostě vynechány, ve třetím případě bude mít atom neplatnou hodnotu. V analýze je vhodné pokračovat z toho důvodu, že překladač může uživateli nahlásit více chyb při jednom překladu.

3.3. Syntaktický analyzátor

Syntaktický analyzátor bývá centrální částí překladače. Jeho úkolem je převést program z (lineární) první interní formy do druhé interní formy, která již bývá více strukturovaná. Nejčastěji se jedná o upravený derivační (syntaktický) strom, který reprezentuje strukturu programu.

V dnešní době je již konstrukce syntaktických analyzátorů známa pro širokou třídu jazyků. Konstrukce syntaktického analyzátoru je založena na konstrukci deterministického zásobníkového automatu pro analýzu shora dolů nebo zdola nahoru. Tyto typy analýzy jsou známy pro třídy bezkontextových jazyků s gramatikami $LL(1)$ a $LR(1)$.

Analýza zdola nahoru

Analyzátory pro analýzu zdola nahoru se nejčastěji sestavují postupem pro sestavení automatu pro gramatiku z třídy $LALR(1)$. Tabulka sestaveného automatu má stejnou velikost jako v případě $LR(0)$, nebo $SLR(1)$ automatu, ale třída rozpoznávaných jazyků je téměř tak široká jako $LR(1)$. Postup sestavení automatu pro gramatiku $LALR(1)$ je uveden v části 1.1.

Při konstrukci analyzátoru se mohou také vyskytnout konflikty, jestliže daná gramatika není $LALR(1)$ gramatikou. Opět existuje několik možných řešení:

- Nahradit konfliktní pravidla nekonfliktními se zachováním generovaného jazyka.
- Přejít k automatu pro gramatiku $LR(1)$, což obvykle stejně nepřinese úspěch.
- Odstranit konflikty ručně, tj. rozhodnout, která akce (přesun nebo redukce) se provede v dané konfliktní situaci.
- Upravit výchozí jazyk.

Konflikty přesun–redukce se často vyskytují v gramatikách popisujících výrazy. V některých gramatikách mohou např. pro výraz `i + i * i` existovat dva derivační stromy, tedy výraz může odpovídat buď výrazu `( i + i ) * i` nebo výrazu `i + ( i * i )`. Zde je možné využít požadované asociativity a precedence operátorů. V tomto příkladě bude v situaci po zpracování druhého `i` při `*` na vstupu rozhodnuto o přesunu a redukován na podvýraz bude až výraz `i * i`. Pokud by na vstupu byl operátor `+`, došlo by k redukci první části výrazu na podvýraz. Rozhodnutí je dáno tím, že operátory `+` a `*` mají stejnou asociativitu, ale `*` má vyšší precedenci než `+`.

Ošetření syntaktických chyb

Podobně jako v lexikální analýze je vhodné, aby při výskytu syntaktické chyby mohla analýza pokračovat dále. V tom případě je potřeba, aby překladač chybu vyhlásit, ale cílový program by neměl být vytvořen. Zotavení analýzy je možné několika způsoby:

- Chybná sekvence zásobníkových symbolů se ze zásobníku odstraní tak, aby analýza mohla pokračovat.
- Na zásobník se doplní další symboly tak, aby analýza mohla pokračovat.
- Chybná sekvence zásobníkových symbolů v zásobníku se nahradí správnou sekvencí.

Existuje několik metod pro automatické zotavení, například metoda významných symbolů. Většinou tyto metody nedávají uspokojivé výsledky, neboť automatické zotavení velmi pravděpodobně způsobí další syntaktickou chybu a chyby se začnou kumulovat.

Kvalitní překladače rozhodují o ošetření chyby na každém místě samostatně. Chyby odpovídají prázdným místům v tabulce automatu. Pro každou chybu je zanalyzováno, v jakých situacích nastává a je pro ni zvoleno konkrétní zotavení.

3.4. Sémantický analyzátor

Úkolem sémantického analyzátoru je určení významu a správnosti jednotlivých částí syntakticky správně zapsaného programu. V dnešní době neexistují automatické postupy konstrukce sémantických analyzátorů, neboť současné metody popisu sémantiky nejsou dostatečné nebo dostatečně formální pro automatické zpracování.

Sémantický analyzátor se obvykle skládá z podprogramů, které jsou volány v průběhu syntaktické analýzy nebo při následném průchodu derivačním stromem programu.

Atributové gramatiky

Při návrhu kompilátorů se používají tzv. *atributové gramatiky*. Tzv. *atributy* jsou sémantické informace vázané na uzly derivačního stromu, tedy na terminální a neterminální symboly gramatiky. Atributová gramatika přidává k běžné gramatice navíc množinu atributů a množinu sémantických pravidel.

Definice 3.1. Trojice $AG = (G, A, R)$ se nazývá *atributová gramatika*, jestliže platí:

- $G = (N, T, P, S)$ je gramatika,
- A je množina atributů
- R je množina sémantických pravidel.

Každému symbolu z $N \cup T$ atributová gramatika přiřazuje některé atributy z A. Atribut a symbolu $X \in N \cup T$ se značí $X.a$.

Každé sémantické pravidlu je navázáno na některé syntaktické pravidlo $X_0 \to X_1 \cdots X_k$ a má tvar

$$X_{i_0}.a_{j_0} = f(X_{i_1}.a_{j_1}, \ldots, X_{i_m}.a_{j_m}),\ i_p \in \{0, \ldots, k\},\ a_{j_p} \in A,\ p \in \{1, \ldots, m\}.$$

Tedy některému atributu některého symbolu v pravidle přiřazuje funkční hodnotu funkce f, jejíž argumenty jsou některé další atributy některých symbolů pravidla.

Atributové gramatiky umožňují přenášení hodnot atributů mezi uzly derivačního stromu po jeho hranách a to jak směrem od listů ke kořeni, tak i od kořene k listům.

Typickým použitím atributů je předávání informace o typu proměnné v rámci deklarace proměnných. Typ je přenášen od uvedení typu v deklaraci identifikátorem nebo klíčovým slovem k jednotlivým identifikátorům proměnných v deklaraci.

Ne vždy je možné všechny potřebné sémantické akce provést ihned při syntaktické analýze. Tato situace nastává např. tehdy, když v jazyce nezáleží na pořadí deklarace určitého prvku a jeho použití. Může nastat situace, že použití určitého prvku uvedením jeho identifikátoru

je syntakticky zpracováváno dříve než jeho deklarace a tedy význam tohoto identifikátoru nemůže být určen za běhu syntaktické analýzy.

Další sémantické akce pak probíhají při následném průchodu syntaktickým stromem programu.

Tabulky symbolů

V programech se obvykle vyskytují deklarace různých entit – tedy informace neproceduální povahy. Deklarace obsahují především jméno (symbol) deklarovaného prvku a další informace. V programu se k těmto prvkům přistupuje právě uvedením jejich jména. Proto jsou informace o deklaracích ukládány do kolekcí obecně nazývaných *tabulky symbolů*.

Nad tabulkami symbolů se nejčastěji provádějí dvě operace:

- Vložení nové deklarace.
- Vyhledání deklarace podle symbolu (jména).

Jelikož se s tabulkami symbolů pracuje poměrně intenzivně, je potřeba efektivně implementovat především operaci vyhledávání. Používají se hashovací tabulky.

Při překladu se většinou nepoužívá jediná tabulka symbolů, ale pro každý deklarační prostor je vytvořena nová tabulka, která je napojena na tabulku nadřazeného deklaračního prostoru. Při vyhledání symbolu je symbol nejdříve hledán v tabulce aktuálního deklaračního prostoru, pokud v ní není nalezen, pokračuje se v hledání v tabulce nadřazeného deklaračního prostoru atd.

Deklarační prostory jsou vnořovány, pokud jazyk podporuje např. vnořené deklarace funkcí nebo procedur, vnořené bloky příkazů apod.

Interní formy při a po sémantické analýze

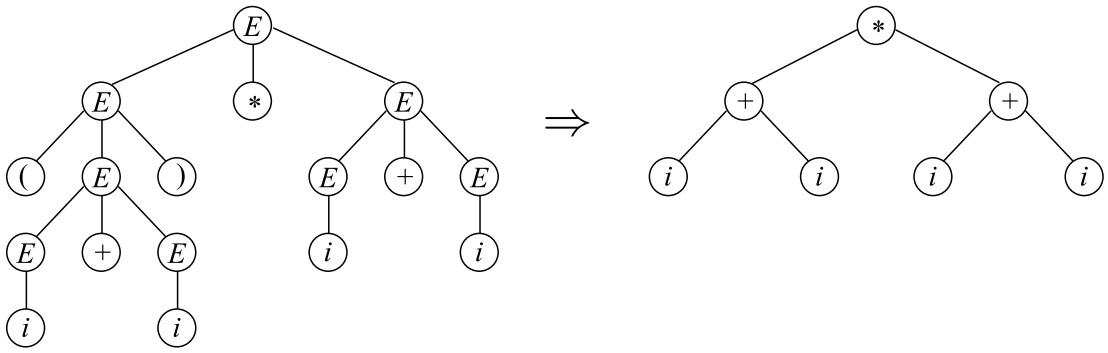
Abstraktní syntaktický strom

Syntaktickou analýzou je program převeden do druhé interní formy, kterou nejčastěji je upravený derivační strom. Z klasického derivačního stromu jsou sémantickými funkcemi za chodu syntaktické analýzy např. přemostěny uzly, které mají pouze jednoho potomka apod. tak, aby jednotlivé uzly odpovídaly více struktuře programu, místo toho, aby přesně zachycovaly odvození věty v gramatice, která může být kvůli jednoznačnosti značně složitější. Takto upravený derivační strom se nazývá *abstraktní syntaktický strom*.

V dalších fázích syntaktické analýzy může být abstraktní syntaktický strom, nebo jen jeho některé části, převedeny např. na jednu z následujících lineárních forem programu:

- čtveřice
- postfixová forma

Obě uvedené lineární formy jsou již poměrně blízké přímo strojovému kódu cílového počítače. Pokud ovšem není cílovým jazykem přímo strojový kód některého (třeba i abstraktního) počítače, ale některý vyšší jazyk, je vhodné cílový program generovat přímo z abstraktního syntaktického stromu.



Obrázek 3. Převod derivačního stromu programu na abstraktní syntaktický strom

Čtveřice

Části programu uvedené v této formě jsou tvořeny posloupností čtveřic, jejichž prvky jsou postupně: operace, první operand, druhý operand, výsledek. Každá čtveřice představuje provedení uvedené operace nad uvedenými operandy a uložení vypočítané hodnoty na místo výsledku. Různé operace mohou a nemusejí využívat všechny části čtveřice.

Postfixová forma

Postfixová forma je vhodná, pokud má být program interpretován, nebo spouštěn na některém tzv. zásobníkovém počítači.

Části programu uvedené v této formě jsou tvořeny posloupností různých instrukcí, které provádějí operace nad operandy, jež odebírají ze zásobníku a svůj výsledek vkládají opět na zásobník. Tato forma získala svůj název proto, že instrukce s operací je uvedena až za instrukcemi vložení svých operandů na zásobník.

3.5. Generátor kódu

Generování kódu je finální částí překladu, při níž se program převede z interní formy do cílového jazyka. Cílovým jazykem nejčastěji bývá přímo strojový kód některého počítače (procesoru). Generátor kódu není obsažen v překladačích, které cílový program jako celek nevytváří.

Jelikož spustitelné programy jsou velmi často závislé jednak na cílovém stroji ale také na daném operačním systému, mají poměrně složitý formát přizpůsobenou pro co nejrychlejší zpracování strojem. Navíc se požaduje, aby generovaný program byl co nejefektivnější a využíval co nejvíce možností daného stroje.

Z těchto důvodů bývají generátory kódu velmi složité a propracované části překladačů. Současná literatura z oblasti návrhu překladačů se nejvíce zabývá právě generováním kódu a optimalizacemi.

3.6. Zpracování chyb

Zpracování chyb při překladu programu je velmi důležité zejména proto, že při psaní programu je potřeba často ověřovat, zda nové části neobsahují chyby. Snad nikdy se nepodařilo napsat (dostatečně rozsáhlý) program, při jehož psaní programátor neudělal chybu. Překladač

by proto měl být programátorovy dobrou pomůckou při odhalování chyb v případě, že cílový program z důvodů chyb nemůže být vytvořen.

Kvalitní systém zpracování chyb překladače by měl mít tyto vlastnosti:

- Při jednom překladu by měl uživateli nahlásit co největší počet z vyskytnuvších se chyb.
- Každý výskyt chyby by měl mít uvedeno místo výskytu, včetně zdrojového souboru, řádku v něm, případně sloupce na tomto řádku.
- Každý výskyt chyby by měl nést informaci o druhu chyby. Pokud druh není zjevný, měl by být volen obecnější text.
- Překladač by měl uživateli nahlásit varování i o situacích, jež nejsou příčinou pro nevytvoření cílového programu, ale jsou programátorskými chybami (např. nedostupný kód, nepoužité proměnná apod.)

Chyby při překladu nastávají při lexikální, syntaktické a sémantické analýze. Jednotlivé druhy chyb jsou popsány v příslušných sekcích výše.

3.7. Překlad objektově orientovaných jazyků

Specifika překladu objektově orientovaných jazyků se týkají hlavně sémantické analýzy a reprezentace objektů v paměti počítače.

Při sémantické analýze se jedná především o vyhledání členů tříd s ohledem na dědičnost a přístupnost členů tříd nebo výběr přetížené metody.

Objekty je nutné reprezentovat v paměti tak, aby bylo možné co nejefektivněji implementovat mechanismus zasílání zpráv objektům pomocí instrukcí některého nižšího jazyka a aby byl co nejjednodušeji zajištěn princip zastupitelnosti předka a potomka.

Vyhledání členů tříd

Členy tříd většinou mívají definovanou tzv. přístupnost. Přístupnost omezuje místa v programu, odkud je možné k nim přistoupit. Těmito místy bývají nejčastěji celé deklarace tříd. Třídám je také dovoleno deklarovat vlastní člen nebo členy se stejným jménem jako člen deklarovaný v některé základní třídě a tedy význam stejného jména může být v různých třídách různý. Jméno může také reprezentovat skupinu přetížených metod a to deklarovaných v různých třídách.

Vyhledání členu podle jména je tedy vždy závislé na tom, v rámci které třídy je jméno uvedeno. Vyhledání členu se jménem `N` v třídě `T` pak obvykle probíhá takto:

- Nejdříve jsou vzaty všechny přístupné členy se jménem `N` v třídě `T`.
- Pokud je nalezen člen, který není metoda, pak je výsledkem hledání tento člen.
- Pokud je nalezena skupina přetížených metod, pak jsou z této skupiny odstraněny metody, jež poskytují novou implementaci pro polymorfní metody definované v základních třídách (přepsané metody). Dále jsou do této skupiny metod stejným způsobem přidány metody ze základních tříd.
- Pokud nebyl žádný člen nalezen, jsou stejným způsobem hledány členy v přímé základní třídě `T` atd.

Výběr přetížené metody

Třídám obvykle bývá dovoleno, aby obsahovaly tzv. přetížené metody, tedy metody se stejným jménem, ale s jiným počtem nebo typy formálních parametrů. Typy formálních parametrů jsou myšleny jejich datové typy (třídy), případně další modifikátory. Při volání metody s uvedeným jménem je nutné vybrat správnou metodu ze skupiny metod a to podle počtu a typů argumentů. Skupina přetížených metod je dodána vyhledáním členů tříd.

1. Ze skupiny metod jsou postupně odstraněny metody, jež nemají stejný počet formálních parametrů jako je počet uvedených argumentů.

2. Dále jsou porovnávány argumenty a formální parametry na stejných pozicích. Odstraněny jsou metody, u nichž nelze dosadit některý argument daného typu za formální parametr jiného typu. To, jestli je možné dosadit argument jiného typu než typ formálního parametru je dáno konverzemi jazyka. Vždy by mělo být ale možné dosadit argument typu, jehož základním typem je typ formálního parametru (princip zastupitelnosti předka a potomka).

3. Zbude-li více metod, pak je potřeba rozhodnout, která z daných metod je lépe aplikovatelná na dané argumenty. Vybrány budou metody u nichž je největší počet argumentů, které se nejlépe dosazují za dané parametry.

4. Pokud zbude více metod, kompilátor může vybrat některou metodu dalšími způsoby nebo nevybrat žádnou.

Překlad metod

V tělech instančních metod je kromě parametrů metody přístupná také aktuální instance objektu, nad níž je metoda volána. Z důvodů efektivity je vhodné přístup k aktuální instanci a formálním parametrům reprezentovat stejně.

Metody se tedy obvykle překládají jako funkce, jejichž první parametr je reference na aktuální objekt. Další parametry a návratový typ jsou stejné jako u metody.

Instanční metoda se signaturou *návratový-typ* `m(` *parametry* `)` definovaná ve třídě `C` je pak uvažována jako funkce *návratový-typ* `fm( ref C this`, *parametry* `)`.

Volání metody tvaru `o.m(` *argumenty*`)` je pak překládáno jako volání funkce `fm(ref o`, *argumenty*`)`.

Metody je takto možné překládat do nižších jazyků a interních forem obvyklým způsobem jako funkce, tedy s využitím aktivačního zásobníku pro uložení aktuálních parametrů a lokálních proměnných (viz sekce 1.3., část Funkce).

Reprezentace objektů v paměti

Podle principu zastupitelnosti předka potomkem je možné použít instanci třídy `B` všude tam, kde je možné použít instanci třídy `A` za předpokladu, že `B` přímo nebo nepřímo dědí z třídy `A`. Z tohoto důvodu je nutné, aby v přeloženém programu bylo možné stejným způsobem přistoupit k atributům (datovým položkám) jak objektu třídy `A`, tak třídy `B`. Dále se uvažuje pouze jednonásobná dědičnost.

Objekty je možné reprezentovat tak, že datové položky jsou vždy řazeny ve stejném pořadí a v objektu zděděné třídy `B` jsou nejdříve uloženy všechny položky z třídy `A` a až potom položky

třídy `B`. Pokud třída `A` dědí z dalších tříd, oblast třídy `A` v objektu třídy `B` podléhá stejným pravidlům.

Takto je možné zacházet s objekty zděděných tříd jako s objekty základní třídy, neboť datové položky přístupné na její úrovni se ve všech objektech třídy samé i zděděných tříd nacházejí na stejných pozicích od začátku objektu v paměti. Tímto jsou v objektech tříd tvořena jakási okénka objektů základních tříd.

Překlad polymorfismu – dynamické vazby

Polymorfismus je definován jako jev, při němž objekty různých tříd mohou reagovat na stejnou zprávu různým způsobem.

V objektově orientovaných jazycích je často způsob zasílání zpráv skryt za přímý přístup k datovým položkám objektu a volání metod objektu, přičemž polymorfismus je možný jen u volání metod.

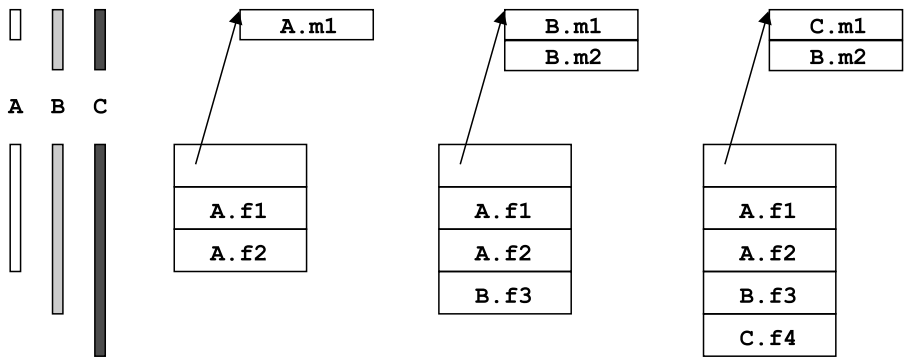
Reprezentaci objektů v paměti a překlad volání metod je potřeba provézt efektivním způsobem. Objekty proto obsahují na svém začátku ukazatel na svoji třídu, tedy skutečnou třídu objektu, ať je v programu objekt uvažován např. jako instance některé základní třídy. Ukazatel směřuje na záznam třídy ve statické oblasti programu, jehož hlavní součástí je tabulka polymorfních metod.

Tato tabulka je u každé třídy vyplněna již při překladu. Obsahuje ukazatele na všechny polymorfní metody třídy, tedy i zděděné. Tabulka polymorfních metod dané třídy přebírá záznamy z tabulky přímé základní třídy a přidává k nim další ukazatele na své nově deklarované polymorfní metody. Zároveň třída přepisuje ukazatele těch polymorfních metod, které přepisuje novou implementací.

Při volání je ukazatel na nepolymorfní metodu určen staticky již při překladu. Při volání polymorfní metody je však ukazatel získán z tabulky polymorfních metod až za běhu programu, tedy z tabulky skutečné instance objektu. Tím je zajištěno polymorfní chování. Volání metody pak probíhá běžným způsobem.

Reprezentace objektů zděděných tříd a implementace polymorfismu pomocí tabulek metod je ukázána na následujícím příkladu:

Nechť třída `A` definuje datové položky `f1` a `f2` a polymorfní metodu `m1`. Dále nechť třída `B` dědí z třídy `A` a definuje datovou položku `f3` a polymorfní metodu `m2` a dále redefinuje polymorfní metodu `m1`. A konečně nechť třída `C` dědí z třídy `B` a definuje datovou položku `f4` a redefinuje polymorfní metodu `m1`. Struktura objektů jednotlivých tříd za běhu programu v paměti je znázorněna na následujícím obrázku.



4. Návrh a architektura interpretu jazyka MiniC#

Hlavní součástí této diplomové práce je vypracování interpretačního kompilátoru jazyka – aplikace MiniC# Interpret. V této sekci je popsán softwarový návrh a architektura výpočetní části této aplikace. Z uživatelského hlediska je aplikace popsána v sekci 5.

Pro návrh byl zvolen objektově orientovaný přístup a pro implementaci jazyk C# ve verzi 2.0.

4.1. Základ architektury

Celý interpret je zvenčí komponenty reprezentován jedinou třídou `MiniCSharpCompiler`. Třída definuje především dvě veřejné metody, metoda `Compile` zajišťuje kompilaci předložených zdrojových souborů, metoda `Interpret` slouží pro spuštění interpretace přeloženého programu.

Metoda `Compile` postupně provádí jednotlivé části překladu programu.

1. Pro každý zdrojový soubor spustí syntaktickou analýzu, jež interně používá lexikální analyzátor. Výsledkem syntaktického analyzátoru je abstraktní syntaktický strom reprezentující program. Lexikální a syntaktická analýza jsou popsány v sekcích 4.2. a 4.3.

2. Pokud se v některém souboru vyskytly chyby, překlad končí neúspěchem. Zpracováním chyb se zabývá sekce 4.5.

3. Následně je nad programem reprezentovaným syntaktickým stromem provedena sémantická analýza.

   Jelikož jazyk MiniC# nevyžaduje nikdy tzv. předsunuté deklarace, většina sémantické analýzy probíhá až po syntaktické analýze a navíc v tomto pořadí:

   (a) Sémantická analýza hlaviček deklarací členů deklarovaných tříd a výčtů.

   (b) Pokud se vyskytnou chyby, překlad je ukončen.

   (c) Sémantická analýza deklarací tříd a výčtů jako celku.

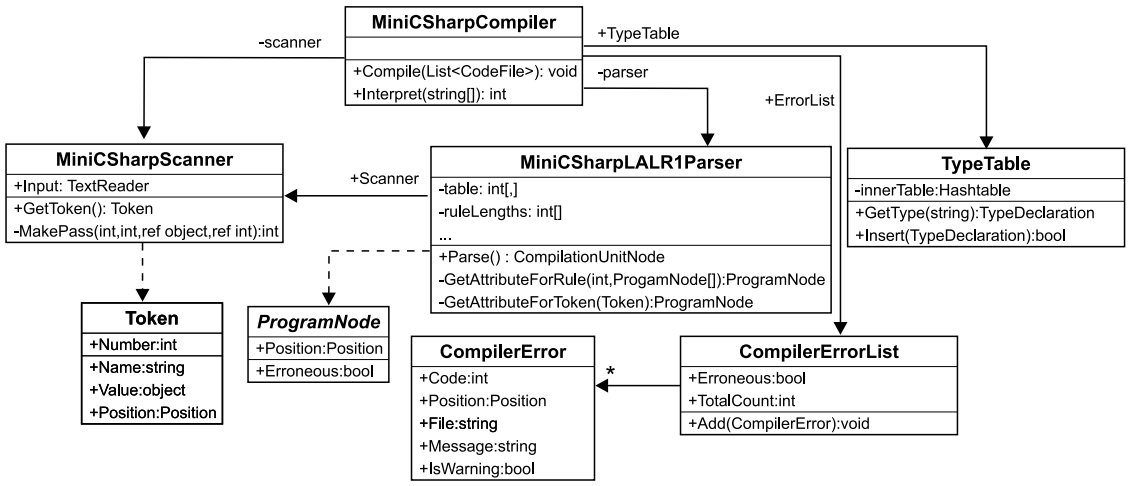
   (d) Pokud se vyskytnou chyby, překlad je ukončen.

   (e) Sémantická analýza proveditelných částí – inicializátorů členů a těla funkčních členů.

   (f) Pokud se vyskytnou chyby, překlad je ukončen.

   Tímto pořadím je vždy zaručeno, že všechny potřebné deklarace a další informace jsou známy a není tím nijak negativně ovlivněna výpočetní složitost překladu. Sémantická analýza je popsána v sekci 4.4.

4. Následně je proveden převod těl funkčních členů tříd, které obsahují spustitelný kód, tedy příkazy a výrazy, do jednodušeji interpretovatelné postfixové formy. Postfixová forma je tvořena posloupností instrukcí, jež jsou interpretovány s pomocí zásobníku. Tato část probíhá vždy bez chyb.



Obrázek 4. Základ architektury interpretu

4.2. Lexikální analýza

Lexikální analyzátor je tvořen jedinou třídou `MiniCSharpScanner`. Zdrojový kód lexikálního analyzátoru byl automaticky vygenerován programem ParserGenerator (viz [10]). Vstupem generátoru byla lexikální gramatiky jazyka MiniC# uvedená v příloze A.

Třída `MiniCSharpScanner` definuje vlastnost `Input`, která představuje vstupní řetězec analyzátoru. Typ vlastnosti `Input` je `TextReader`, což je standardní třída v .NET Framework. Vstupem může být velice jednoduše soubor, řetězec v paměti, nebo cokoliv jiného, stačí k němu vytvořit příslušný `TextReader`.

Algoritmus lexikálního analyzátoru je implementován v metodě `GetToken`. Při každém volání metoda navrací další lexikální symbol (atom) reprezentovaný objektem třídy `Token`.

Třída `Token` obsahuje pouze datové položky, sdružuje informace o lexikálním symbolu: číselný kód (`Number`), jméno (`Name`), hodnotu atomu (`Value`), pozici ve zdrojovém textu (`Position`).

Algoritmus lexikálního analyzátoru využívá spoustu dalších informací reprezentovaných soukromými členy třídy `MiniCSharpScanner`.

Přechodová funkce spolu s výpočty hodnot lexikálních symbolů je reprezentována soukromou metodou `MakePass`. Vstupními parametry jsou číslo aktuálního stavu, v němž se analyzátor nachází a znak na vstupu. Metoda je implementována jako velký přepínač podle čísla aktuálního stavu, uvnitř sekcí, které patří jednotlivým stavům, jsou další přepínače podle znaku na vstupu. Do této automaticky vygenerované metody bylo potřeba ručně připojit zpracování hodnot lexikálních symbolů. Návratovou hodnotou metody je vždy číslo nového stavu analyzátoru, pomocí výstupních parametrů se navrací číslo rozpoznaného atomu (nemusí být žádný) a „rozpočítaná“ hodnota atomu.

4.3. Syntaktická analýza

Syntaktický analyzátor je tvořen jedinou třídou `MiniCSharpLALR1Parser`. Zdrojový kód syntaktického analyzátoru byl opět vygenerován programem ParserGenerator (viz [10]). Vstupem pro generátor byla syntaktická gramatika jazyka MiniC# uvedená v příloze B.

Gramatika vychází z gramatiky jazyka C# (viz [6]). Vynechány byly symboly a pravidla, které jazyk MiniC# nepoužívá. Gramatika ovšem nebyla z třídy gramatik $LR(1)$ ani $LALR(1)$.

Některé části (určitá pravidla) jsou pro tvar $LALR(1)$ příliš specifické. Některé jednodušší případy bylo proto nutno nahradit ekvivalentními pravidly splňující podmínky. Složitější případy bylo nutno řešit pozměněním (rozšířením) jazyka ze syntaktického hlediska. Tyto případy jsou pak ošetřeny v sémantické analýze. Úpravy byly provedeny podle $LALR(1)$ gramatiky jazyka Java (viz [3]), jehož syntaxe také vychází z jazyka C.

Prvním takovým případem bylo to, že modifikátory členů tříd jsou příliš specifické – každý člen definoval jen svoje povolené modifikátory (viz část 2.5.). Gramatika byla upravena tak, že všechny členy mohou mít syntakticky všechny modifikátory, jejich správnost se kontroluje až při syntaktické analýze.

Druhým takovým problémem byl známý konflikt mezi syntaxí uzávorkovaného výrazu a výrazu změny typu (viz část 2.9.). Gramatika byla upravena tak, že v závorkách, v nichž je uveden typ ve výrazu přetypování je syntakticky povolen jakýkoliv výraz. Sémantiky je pak ošetřeno, že tímto výrazem musí být pouze identifikátor (který navíc určuje typ).

Syntaktická analýza se spouští metodou `Parse`. Tato metoda implementuje algoritmus syntaktického analyzátoru pro $LALR(1)$ gramatiky. Vstup analyzátoru je reprezentován objektem třídy `MiniCSharpScanner` (vlastnost `Scanner`). Pokaždé, když syntaktický analyzátor potřebuje číst další symbol ze vstupu, volá metodu `GetToken` tohoto objektu.

Metoda `Parse` také implementuje zotavení syntaktické analýzy po chybě pomocí metody významných symbolů.

Výstupem metody je abstraktní syntaktický strom předložené části programu. Jeho uzly jsou reprezentovány objekty tříd, jež jsou potomky třídy `ProgramNode`.

Algoritmus syntaktického analyzátoru používá pro svoji činnost spoustu dalších informací, jenž jsou reprezentovány soukromými členy třídy `MiniCSharpLALR1Parser`. Jedná se především o tabulku analyzátor s uvedenými přesuny a redukcemi. Dále to jsou metody pro vytvoření uzlů syntaktického stromu. Při získání nového atomu je volána metoda `GetAttributeForToken`. Při redukcích podle pravidel je volána metoda `GetAttributeForRule`. Parametrem druhé metody je pole atributů, jež odpovídají atributů již zanalyzovaných symbolů na pravé straně redukovaného pravidla. Obě metody jsou implementovány pomocí přepínače podle čísla atomu nebo pravidla. Akce prováděné při vytváření uzlů stromu byly do automaticky vygenerovaného zdrojového kódu analyzátoru zapsány ručně na označená místa.

4.4. Sémantická analýza

Sémantická analýza probíhá průchody abstraktním syntaktickým stromem programu vygenerovaným syntaktickým analyzátorem. Jednotlivé části sémantického analyzátoru jsou rozděleny do dedikovaných metod tříd, jež reprezentují typy uzlů syntaktického stromu.

Třídy, které reprezentují typy uzlů syntaktického stromu jsou potomky abstraktní třídy `ProgramNode`.

Tyto třídy lze hrubě rozdělit do několika kategorií, podle toho jakou část programu reprezentují na:

- Deklarace typů, tedy tříd a výčtových typů.

- Deklarace členů typů, tedy metod, vlastností, datových položek, členů výčtů atd.
- Výrazy.
- Příkazy.
- Třídy, jež reprezentují součásti deklarací, nebo příkazů nebo výrazů, např. parametry, argumenty, inicializátory apod.

Program v jazyku MiniC# se celý skládá z deklarace jednoho nebo více typů, jiné globální prvky neexistují. Sémantická analýza je vždy spuštěna vyvoláním určené metody pro uzly stromu, které reprezentují typy a v rámci těchto metod analýza postupuje dál do hloubky stromu voláním podobných metod potomků aktuálního uzlu.

Jak již bylo uvedeno výše sémantická analýza probíhá ve třech průchodech syntaktickým stromem.

- První průchod analyzuje hlavičky deklarací členů typů.
- Druhý průchod analyzuje deklarace typů jako celek.
- Třetí průchod analyzuje proveditelný kód členů typů. Jedná se jednak o inicializační výrazy např. členů výčtů nebo datových položek a jednak o těla tzv. funkčních členů tříd, tedy konstruktorů, metod a vlastností. V případě, že analýza spustitelného kódu v celé třídě proběhla úspěšně, jsou volány metody funkčních členů pro převod těl do postfixové formy vhodné pro interpretaci.

Reprezentace deklarací typů

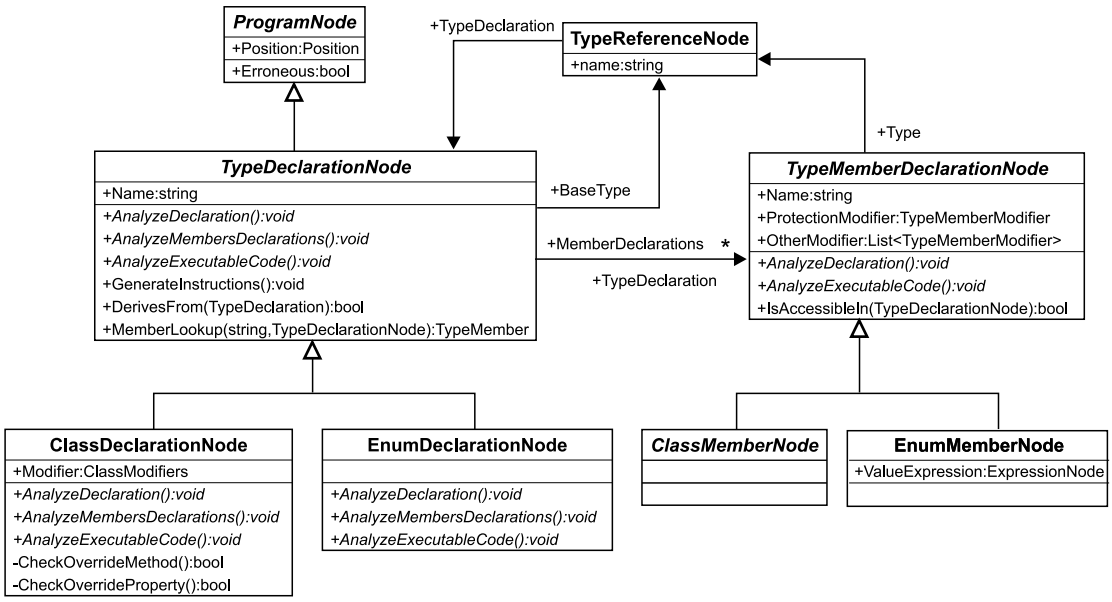
Základní třídou pro deklarace typů je třídy `TypeDeclarationNode`. Tato třída deklaruje virtuální metody pro jednotlivé části sémantické analýzy – `AnalyzeMembersDeclarations`, `AnalyzeDeclaration`, `AnalyzeExecutableCode` a dále referenci na základní typ `BaseType` a kolekci členů typu `MemberDeclarations`.

Deklarace výčtových typů jsou reprezentovány objekty třídy `EnumDeclarationNode`, deklarace tříd objekty třídy `ClassDeclarationNode`. Obě třídy dědí z třídy `TypeDeclarationNode`.

Pří sémantické analýze deklarací typů metodou `AnalyzeDeclaration` se u tříd kontroluje správnost uvedené základní třídy. Proto je nejdříve provedena analýza deklarace základní třídy. Jelikož jsou již zanalyzovány hlavičky deklarací členů aktuální třídy i základní třídy, dále se analyzuje správnost deklarací virtuálních, přepsaných a abstraktních metod a vlastností, v případě potřeby je generován implicitní konstruktor.

Reprezentace deklarací členů typů

Základní třídou pro deklarace členů typů je třída `TypeMemberDeclarationNode`. Tato třída deklaruje virtuální metody pro jednotlivé části sémantické analýzy členů – `AnalyzeDeclaration` a `AnalyzeExecutableCode`. Dále třída definuje odkaz `TypeDeclaration` na deklaraci typu, v němž je deklarace členu obsažena, modifikátor přístupnosti `ProtectionModifier`, kolekci ostatních modifikátorů `OtherModifiers`, referenci na datový typ členu `Type` a další členy.



Obrázek 5. Reprezentace deklarací typů a členů typů

Z třídy `TypeMemberDeclarationNode` dědí třída `EnumMemberNode` reprezentující deklaraci členu výčtového typu a třída `ClassMemberDeclarationNode` reprezentující členy tříd.

Z třídy `ClassMemberDeclarationNode` dále dědí třídy `ConstantDeclarationNode` pro deklarace konstant, `FieldDeclarationNode` pro deklarace datových položek, `PropertyDeclarationNode` pro deklarace vlastností a `FunctionMemberNode` pro deklarace funkčních členů.

Z třídy `FunctionMemberNode` ještě dále dědí třídy `ConstructorDeclarationNode` pro deklarace konstruktorů a `VirtualFunctionMemberNode` pro deklarace funkčních členů, jež mohou být virtuální.

Těmito členy jsou tedy metody reprezentované třídou `MethodDeclarationNode` a akcesory vlastností `PropertyAccessorDeclarationNode`.

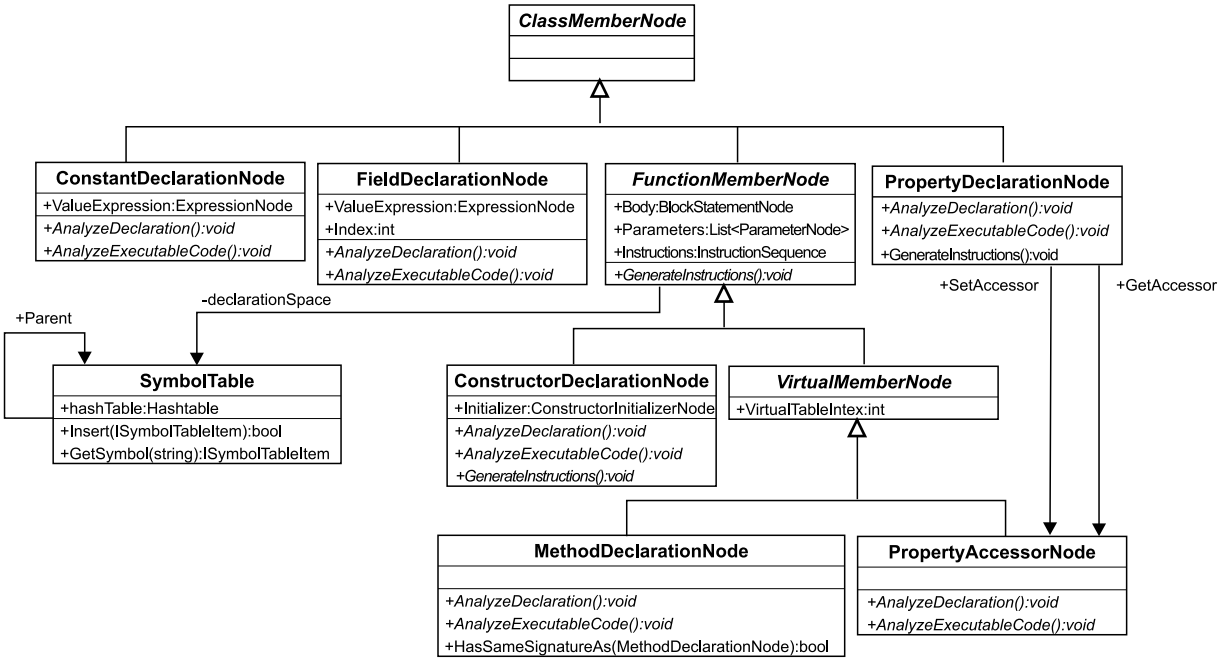
Všechny tyto uvedené třídy deklarují nové členy (vlastnosti, metody) potřebné pro analýzu a implementují metody sémantické analýzy.

Při kontrole deklarace členu metodou `AnalyzeDeclaration` se kontroluje, zda uvedené modifikátory jsou platné, zda existuje uvedený typ, zda se uvedené jméno vyskytuje u jediného členu v typu, u metod a konstruktorů navíc ještě typy a jména parametrů.

Reprezentace příkazů

Základní třídou pro reprezentaci příkazů je třída `StatementNode`. Třída deklaruje virtuální metody pro sémantickou analýzu `AnalyzeExecutableCode` a pro vytvoření postfixové formy programu `GenerateInstructions`. Dále třída deklaruje vlastnosti `Reachable` a `EndPointReachable`, které určují dostupnost začátku a konce příkazu.

Parametry metody `AnalyzeExecutableCode` jsou deklarační prostor, v němž se výraz vyskytuje, členy typu, v jehož rámci je člen uveden a dále nejbližší obsahující příkaz, jenž lze ukončit příkazem `break`, a nejbližší obsahující příkaz cyklu. Z těchto objektů je možné získat všechny potřebné údaje pro analýzu. Parametry se předávají dále při volání analýzy

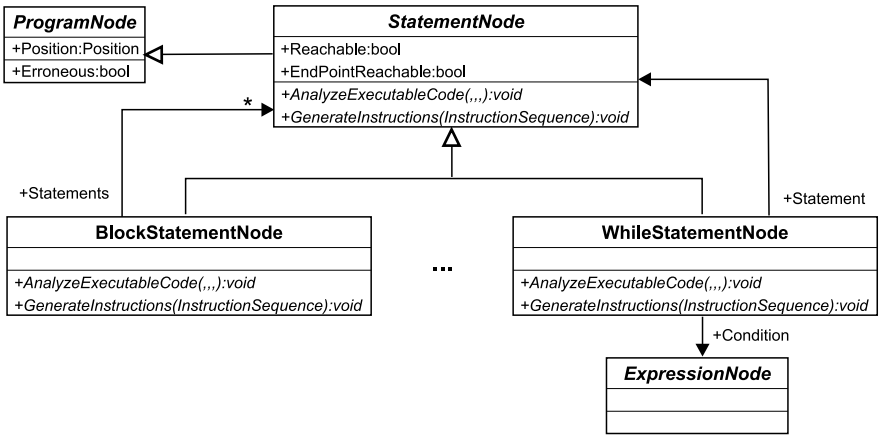


Obrázek 6. Reprezentace deklarací členů tříd

obsaženým příkazům a výrazům.

Jednotlivé druhy příkazů jsou reprezentovány samostatnými třídami, které dědí z třídy `StatementNode`. Deklarují svoje specifické vlastnosti, které reprezentují obsažené výrazy a příkazy a další specifické informace o příkazu. Samozřejmě přepisují virtuální metody třídy `StatementNode`, v nichž je implementována sémantická analýza příkazu.

Sémantická analýza je poměrně specifická pro každý druh příkazů. Ve všech příkazech se provádí určení dosažitelnosti konce příkazu a také se kontroluje definitivní přiřazenost lokálních proměnných v možných větvích vykonávání programu. U obsažených výrazů se kontrolují datové typy a klasifikace (viz níže).



Obrázek 7. Reprezentace příkazů

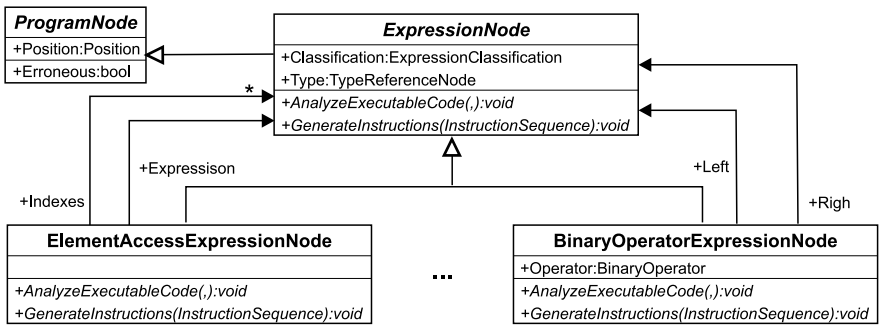
Reprezentace výrazů

Základní třídou pro reprezentaci výrazů je třída `ExpressionNode`. Třída deklaruje podobně jako u příkazů virtuální metody pro sémantickou analýzu `AnalyzeExecutableCode` a `GenerateInstructions`. Dále deklaruje vlastnost `Type` reprezentující datový typ výrazu a vlastnost `Classification` reprezentující tzv. klasifikaci výrazu, tzn. zda výraz určuje hodnotu nebo proměnnou, vlastnost, metodu nebo typ. Klasifikace je používána ve složených výrazech, některé části výrazů mohou mít jen určitou klasifikaci.

Parametry metody `AnalyzeExecutableCode` jsou deklarační prostor, v němž se výraz vyskytuje a členy typu, v jehož rámci je člen uveden. Z těchto objektů je možné získávat všechny potřebné údaje pro analýzu. Oba parametry se předávají při volání analýzy podvýrazů aktuálního výrazu.

Jednotlivé druhy výrazů jsou reprezentovány samostatnými třídami, které dědí z třídy `ExpressionNode`. Deklarují svoje specifické vlastnosti, které nejčastěji reprezentují podvýrazy a samozřejmě přepisují virtuální metody třídy `ExpressionNode`, v nichž je implementována sémantická analýza výrazu.

Sémantická analýza výrazů je pro každý druh poměrně specifická. Nejčastější prováděné akce jsou kontroly datového typu a klasifikace podvýrazů a následně určení datového typu a klasifikace celého výrazu. Většina výrazů představuje různé operace. V těchto případech je potřeba určit, zda operátor lze aplikovat na podvýrazy daných typů. K tomu slouží metoda rozhodující o přetíženém členu, která je použitelná jak pro metody a konstruktory, tak pro operátory. Tato metoda je implementován jako statická metoda `OverloadResolution` třídy `MiniCSharpSematicRules`. Algoritmy a metody pro zjišťování a generování konverzí se nachází jako statické metody v třídě `Conversions`. V případě výrazů jednoduché jméno a přístup ke členu se prohledává deklarační prostor, v němž se výraz vyskytuje, a hledá se, kterou deklarovanou entitu reprezentuje apod.



Obrázek 8. Reprezentace výrazů

Reprezentace deklaračních prostorů

Globální deklarační prostor obsahuje všechny deklarované typy. Tento deklarační prostor je reprezentován objektem třídy `TypeTable`. Objekt kompilátoru (třídy `MiniCSharpCompiler`) obsahuje jedinou instanci globálního deklaračního prostoru. Třída `TypeTable` nabízí operace vložení a vyhledání typu podle jména implementované v metodách `Insert` a `GetType`. Hodnotami v tabulce jsou přímo deklarované typy (objekty tříd

EnumDeclarationNode a ClassDeclarationNode). Tabulka je implementována pomocí hashovací tabulky.

Deklarační prostory vytvářené deklaracemi typů obsahují deklarace jejich členů, proto jsou reprezentovány kolekcí MemberDeclarations definované v třídě TypeDeclaration. Při vyhledávání členů tříd se berou v úvahu i členy ze základních tříd. Vyhledání členu typu, jehož jméno je uvedeno v rámci deklarace určitého typu je implementována v metodě MemberLookup ve třídě TypeDeclaration. Parametrem metody je hledané jméno a deklarace typu, v rámci jehož těla je jméno uvedeno. Algoritmus vyhledání bere v úvahu dědičné vztahy aktuálního typu a uvedeného typu a porovnává ji s přístupností členů.

Deklarační prostory, které vytvářejí deklarace funkčních členů a příkazy jsou reprezentovány objekty třídy SymbolTable. Tyto deklarační prostory mohou obsahovat deklarace, které překrývají jména obsažená v nadřazeném deklaračním prostoru.

Třída SymbolTable obsahuje hashovací tabulku s klíči, kterými jsou jména deklarovaných prvků. Hodnotami v tabulce jsou přímo deklarace s danými jmény. Dále třída SymbolTable obsahuje vlastnost Parent, která reprezentuje nadřazenou tabulku symbolů.

Třída SymbolTable deklaruje dvě metody Insert a GetSymbol, které představují operace vložení symbolu a vyhledání symbolu. Operace vyhledání je implementována tak, že jméno je vždy nejprve hledáno ve vlastní hashovací tabulce, pokud hledání neuspěje, provede se hledání stejným způsobem v nadřazené tabulce.

Reprezentace postfixové formy

Každý funkční člen třídy obsahuje posloupnost proveditelných instrukcí, která se vytváří po úspěšné sémantické analýze. Instrukce jsou při běhu programu prováděny s pomocí zásobníku. Třída FunctionMemberNode obsahuje vlastnost Instructions typu InstructionSequence.

Třída InstructionSequence deklaruje několik přetížených metod Add pro přidání instrukce. Prvním parametrem těchto metod je vždy kód instrukce, dalšími parametry mohou být různé hodnoty - indexy, konstanty, typy, funkční členy, návěští apod. Dále deklaruje metody CreateLabel a PlaceLabel pro vytvoření a následné vložení návěští na aktuální konec posloupnosti.

Spustitelný kód ve stromové formě je do těchto instrukcí převeden v rámci metod GenerateInstructions, které jsou definovány v třídách StatementNode a ExpressionNode. Při generování instrukcí je příkazům a výrazům předáván jako parametr těchto metod objekt představovaný vlastností Instructions aktuálního funkčního členu. V rámci těchto metod jsou přidávány příslušné instrukce a umísťovány návěští (cíle skoků) pomocí metod Add a PlaceLabel tohoto parametru.

Instrukce jsou reprezentovány objekty tříd, jež dědí z třídy Instruction. Třída Instruction deklaruje číselný kód instrukce a hlavně abstraktní metodu Interpret, kterou implementují jednotlivé třídy instrukcí. Třídy reprezentující instrukce se od sebe liší tím, jaké mají instrukce parametry.

Zde je výčet používaných instrukcí:

- Instrukce načítání dat do zásobníku – LoadConstant, LoadString, LoadNull, LoadLocal, LoadLocalAddress, LoadArgument, LoadArgumentAddress, LoadField, LoadFieldAddress, LoadStaticField, LoadStaticFieldAddress, LoadElement, LoadElementAddress, LoadFromAddress.

- Instrukce ukládání dat do paměti – `StoreLocal`, `StoreArgument`, `StoreField`, `StoreStaticField`, `StoreElement`, `StoreToAddress`.
- Instrukce manipulace se zásobníkem – `Pop`, `Duplicate`.
- Matematické, logické a relační instrukce – `Add`, `Subtract`, `Multiply`, `Devide`, `Remainder`, `ShiftLeft`, `ShiftRight`, `Not`, `And`, `Or`, `Xor`, `Negate`, `Equal`, `LessThan`, `GreaterThan`, `LessOrEqualThan`, `GreaterOrEqualThan`.
- Instrukce změny typu a konverzí – `IsInstance`, `CastClass`, `Box`, `Unbox`, `Convert`.
- Instrukce skoků – `Branch`, `BranchIfZeroOrNull`, `BranchIfNotZeroOrNull`, `Return`.
- Instrukce vytvoření objektu – `NewObject`, `NewArray`.
- Instrukce volání metod – `Call`, `CallVirtual`.

4.5. Zpracování chyb

V úvodu této sekce je popsán postup překladu programu. Jednotlivé části mohou generovat chyby. Překladač je navržen tak, aby při jednom překladu mohlo být uživateli předloženo co nejvíce vyskytnuvších se chyb. Pokud by ovšem chyby generovaly další chyby v následné fázi překladu, překlad je ukončen.

Chyby při překladu jsou reprezentovány objekty třídy `CompilerError`. Tato třída deklaruje následující vlastností: `Code` – číselný kód chyby, `Position` – řádek a sloupec, kde se chyba vyskytla, `File` – zdrojový soubor s chybou, `Message` – textový popis chyby, `IsWarning` – indikace, jestli se jedná o chybu nebo varování.

Objekt kompilátoru `MiniCSharpCompiler` obsahuje vlastnost `ErrorList`, který reprezentuje kolekci chyb nebo varování vzniklých během překladu. Tato kolekce poskytuje metodu `Add`, pomocí níž jednotlivé části překladače přidávají chyby.

Chyby při překladu vznikají při lexikální, syntaktické i sémantické analýze. Všechny části jsou implementovány tak, že po přidání chyby do seznamu je provedeno zotavené aktuální části analýzy.

Při lexikální analýze pozici chyby nastavuje přímo lexikální analyzátor, jedná se o pozici, na které začala aktuálního atomu.

Syntaktické chyby nastávají, pokud tabulka automatu neobsahuje žádnou akci pro danou nastalou situaci. V algoritmu analýzy v třídě `MiniCSharpLALR1Parser` je implementována metoda zotavení pomocí tzv. významných symbolů. Pozici chyby nastavuje syntaktický analyzátor, je to pozice, kterou si nese atom, který byl na vstupu při nastání chyby.

Pokud se při lexikální nebo syntaktické analýze vyskytnou chyby, nemá smysl, aby překlad pokračoval sémantickou analýzou, protože program v interní formě plně neodpovídá zdrojovému programu a zcela určitě by se kupily další chyby.

Při výskytu sémantické chyby v daném uzlu stromové reprezentace programu je uzel označen jako chybný nastavením příznaku `Erroneous` (deklarovaného v třídě `ProgramNode`) na hodnotu `true`. Pomocí tohoto příznaku se šíří informace o chybě jeho předchůdcům z nichž byla analýza volána. Tímto systémem je možné zamezit generování dalších a dalších chyb způsobených jedinou skutečnou chybou tak, že pokud je při analýze zjištěna chybnost některého uzlu jsou vynechány určité kontroly, které mohou generovat následné chyby.

Pozice chyby je určena podle pozice uzlu, který je chybný. Pozice uzlům syntaktického stromu nastavuje již syntaktický analyzátor, jedná se o vlastnost `Position` deklarovanou v třídě `ProgramNode`. Pro uzly, jež odpovídají atomům, je pozice převzata přímo atomu. Uzlům, jež vznikají při redukci pravidla, je pozice nastavena podle uzlu, který odpovídá prvnímu symbolu na pravé straně pravidla.

4.6. Interpretace

Provedením kompilace zůstává program uchován v syntaktickém stromu a v postfixové formě. Pokud je bez chyb, může být spuštěn – interpretován. Interpretace se spouští voláním metody `Interpret` třídy `MiniCSharpCompiler`.

Interpretace programu začíná voláním určeného funkčního členu – vstupního bodu programu. Nalezení vstupního bodu programu je součástí sémantické analýzy.

Interpretace postfixové formy probíhá v rámci interpretačního prostředí reprezentovaného objektem třídy `InterpretationEnvironment`. Hlavní součástí je tzv. vyhodnocovací zásobník a také zásobník pro aktivace funkčních členů. Dále prostředí obsahuje odkaz na aktuálně prováděný funkční člen a ukazatel aktuální instrukce.

Voláním vstupního bodu programu vznikne první aktivační záznam a následně interpret postupně provádí instrukce dané aktuálním funkčním členem a aktuální hodnotou ukazatele instrukcí. Instrukce se obecně chovají tak, že z vyhodnocovacího zásobníku odeberou několik objektů jako operandy a provedou nad nimi svoji operaci. Případnou výslednou hodnotu vloží zpět na zásobník. Přitom se posune ukazatel aktuální instrukce, nejčastěji na další instrukci. Skokové instrukce přenáší výpočet obecně na jinou instrukci než následující.

Při volání funkčního členu je na aktivační zásobník uložen aktuální stav prostředí, tedy aktuální funkční člen a ukazatel instrukce. Následně jsou na aktivační zásobník přesunuty argumenty volání funkčního členu a je vyhrazeno místo pro lokální proměnné.

Postfixová forma je vždy sestavena tak, aby provádění funkčního členu končilo provedením instrukce `Return`. Návratová hodnota se nachází na vrcholu vyhodnocovacího zásobníku. Instrukce `Return` odstraní argumenty a lokální proměnné z aktivačního zásobníku a obnoví aktuální funkční člen a ukazatel instrukce.

Program končí odstraněním nejhlubšího aktivačního záznamu z aktivačního zásobníku instrukcí `Return` na konci vstupního bodu programu.

V průběhu interpretace programu může dojít k chybám. Nejčastěji se jedné o klasické chyby – dělení nulou, přístup k instančnímu členu, pokud je aktuální instance `null`, překročení rozsahu pole, nesprávné přetypování atd. V těchto případech vykonávání programu bezpodmínečně končí chybou.

Reprezentace objektů v paměti

Objekty interpretovaného programu jsou v paměti uloženy jako instance potomků třídy `ObjectData` v interpretu.

Instance hodnotových typů programu jsou reprezentovány instancemi tříd `Int8ObjectData`, `Float64ObjectData` apod., které mají společnou základní třídu `ValueTypeObjectData`, která dědí z třídy `ObjectData`. Objekty těchto tříd obsahují vždy jen jednu datovou položku odpovídajícího (bitového) rozsahu.

Instance odkazových typů programu jsou reprezentovány instancemi tříd `ClassObjectData`, `ArrayObjectData` a `BoxedObjectData`, jejichž společnou základní

třídou je `ReferenceTypeObjectData`, která dědí z třídy `ObjectData`. Tyto objekty obsahují odkaz na deklarační záznam datového typu, jehož instanci reprezentují, z něhož se při volání virtuálních funkcí využívá tabulka virtuálních funkcí. Třída `ClassObjectData` dále obsahuje pole typu `ObjectData[]`, ve němž jsou ukládány jednotlivé datové položky. Třída `ArrayObjectData` obsahuje opět pole objektů typu `ObjectData`, v němž jsou uloženy elementy pole. Třída `BoxedObjectData` slouží pro reprezentaci boxovaných hodnot hodnotových typů, obsahuje datovou položku typu `ValueTypeObjectData`, v níž je uložena boxovaná hodnota.

5. Uživatelská a programátorská příručka

Součástí této diplomové práce je také přiložené CD, které obsahuje aplikaci MiniC# Interpret jednak ve spustitelném tvaru, tak také ve zdrojovém kódu. Spustitelné soubory aplikace se nacházejí v adresáři `bin`, který se nachází v hlavním adresáři CD.

5.1. Systémové prostředí

Aplikace MiniC# Interpret potřebuje ke svému běhu operační systém z rodiny Windows a nainstalovaný .NET Framework 2.0. Aplikace je složena ze 4 souborů (komponent):

- `MiniCSharpInterpreter.dll` – komponenta obsahující výpočetní část aplikace.
- `mcsbaselib.dll` – komponenta obsahující implementaci knihovny základnách tříd.
- `mcs.exe` – spustitelný soubor aplikace v konzolovém rozhraní.
- `MiniCSharpStudio.exe` – spustitelný soubor aplikace s grafickým uživatelským rozhraním.

5.2. Konzolové rozhraní

Konzolové rozhraní aplikace je velmi jednoduché. Veškerá komunikace s uživatelem probíhá přes standardní textový vstup a výstup programu. Aplikaci v konzolovém rozhraní je možno spustit z příkazového řádku systému Windows nebo odjinud následujícím způsobem:

`mcs.exe` <*parametry*> <*seznam zdrojových souborů*>

Následující výčet obsahuje možné parametry aplikace spolu s jejich významem:

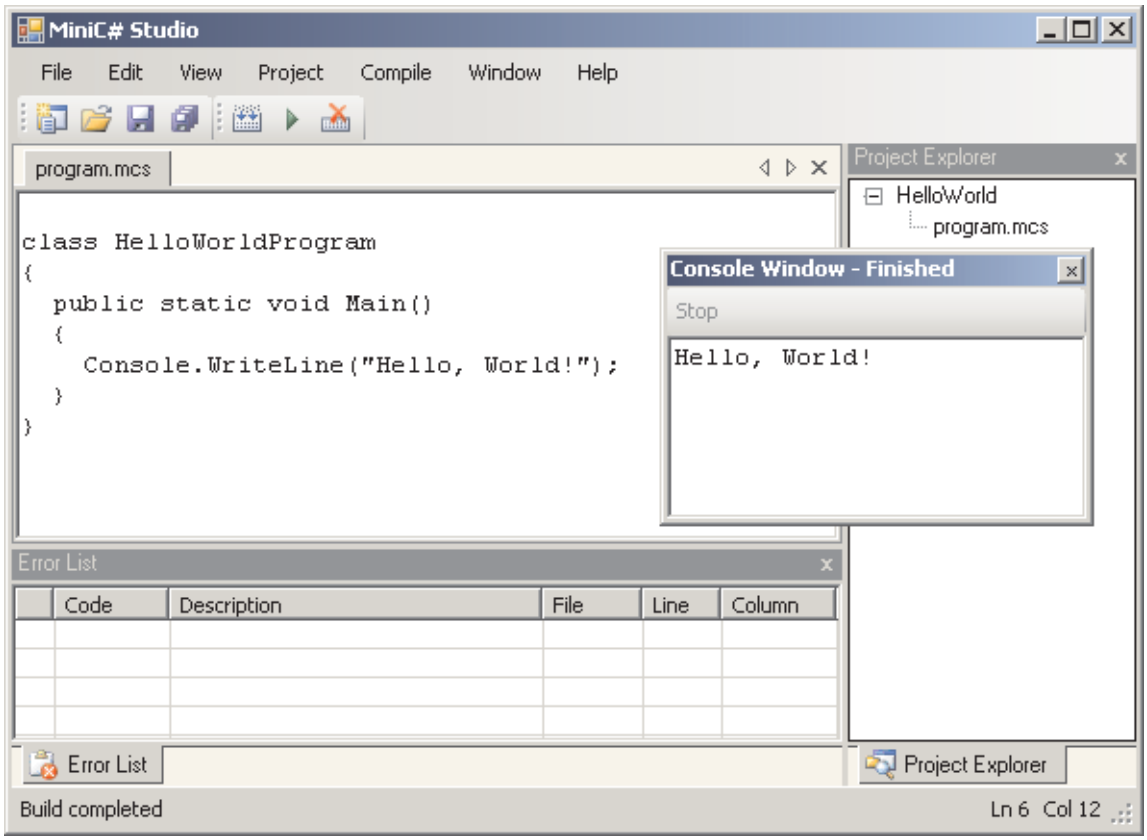
`/help` – zobrazí nápovědu k aplikaci `mcs.exe` s výpisem možných parametrů (je možná i krátká forma `/?`).

`/nologo` – potlačí informaci o aplikaci při spuštění.

`/compileonly` – po úspěšné kompilaci nebude provedena interpretace programu (krátká forma je `/c`).

`/args:`<*text*> – udává argumenty pro spouštěný programu, pokud je argumentů více, celý <*text*> musí být uzavřen v uvozovkách `""`.

Po spuštění aplikace `mcs.exe` je nejdříve provedena kompilace programu daného uvedenými zdrojovými soubory. Aplikace vypisuje chybová hlášení na standardní chybový výstup. Pokud kompilace skončila úspěšně a pokud není uveden parametr `/compileonly`, je provedena interpretace programu. Pokud jsou uvedeny argumenty programu parametrem `/args`, jsou předány spuštěnému programu. Po ukončení programu (případně následkem chyby za běhu programu) je ukončena i aplikace `mcs.exe`.



Obrázek 9. Integrované vývojové prostředí aplikace MiniC# Interpret

5.3. Integrované vývojové prostředí

Aplikace v grafickém uživatelském prostředí systému Windows se spouští programem `MiniCSharpStudio.exe`.

Hlavní okno aplikace se skládá z hlavního menu, panelu nástrojů, oblasti dokumentů, nástrojových oken a stavového řádku.

Základní entitou, s kterou aplikace pracuje, jsou projekty. Projekt sdružuje zdrojové soubory jednoho programu.

Hlavní menu obsahuje příkazy pro práci s aplikací. Menu je děleno do následujících podmenu:

- File – obsahuje příkazy pro vytváření, otevírání, uzavírání a ukládání projektů.
- Edit – obsahuje obvyklé příkazy pro editaci zdrojového textu programu.
- View – obsahuje příkazy pro změnu zobrazení oken aplikace.
- Project – obsahuje příkazy pro nastavení aktuálního projektu a pro přidávání, odebírání a přejmenovávání položek aktuálního projektu.
- Compile – obsahuje příkazy pro spuštění a zastavení kompilace a pro spuštění interpretace aktuálního projektu.

- Windows - obsahuje obvyklé příkazy pro správu oken aplikace.
- Help - obsahuje příkazy nápovědy aplikace.

Vybrané nejpoužívanější příkazy jsou také uvedeny na panelech nástrojů pod menu.
Zdrojové soubory programu se editují prostřednictvím editačních oken aplikace.
Pro pohodlnou práci s aplikací slouží okna nástrojů:

- Project Explorer – zobrazuje strukturu projektu a je možné zde přidávat, odebírat a přejmenovávat jednotlivé položky projektu.
- Error List – zobrazuje chyby, které se vyskytly při překladu projektu. Dvojklikem na řádek s chybou je možné ihned zobrazit místo s chybou ve zdrojovém souboru, který se otevře v oblasti dokumentů.
- Console Window – v tomto okně je možné komunikovat s interpretovaným program pomocí textového vstupu a výstupu.

Ve stavovém řádku na spodním okraji hlavního okna aplikace jsou zobrazovány informace o prováděných činnostech, aktuální pozice kurzoru v editačním okně apod.

5.4. Knihovna základních typů

Součástí aplikace je knihovna základních typů, jež se nachází v souboru `mcsbaselib.dll`. Knihovna obsahuje především definice typů, jež vyžaduje přímo programovací jazyk MiniC# a dále vybrané „systémové“ třídy např. pro práci se soubory, standardními vstupy a výstupy nebo např. se systémovým časem.

Následuje výčet většiny typů základní knihovny spolu s jejich popisem a popisem jejich členů. Některé členy uvedených tříd nejsou pro stručnost uvedeny, jedná se převážně o přepsané metody virtuálních metod třídy `object`. Kompletní popis knihovny je uveden na přiloženém CD a je také přístupný z menu Help aplikace MiniC# Studio.

Třída `object`

```
class object
```

Třída `object` představuje základní typ všech ostatních typů a tím tvoří kořen stromu hierarchie typů v programu jazyka MiniC#. Tato třída dále poskytuje základní metody společné pro všechny objekty.

Konstruktory:

```
public object()
```

Inicializuje novou instanci třídy `object`. Tento konstruktor je volán konstruktory zděděných tříd, ale může být použit i pro přímé vytvoření objektu.

Metody:

```
public virtual bool Equals(object obj)
```

Určuje, zda specifikovaný objekt `obj` je roven aktuální instanci. Tato metoda porovnává objekty pomocí shodnosti referencí. Zděděné třídy mohou tuto metodu přepsat tak, že objekty jsou porovnávány podle svých datových položek.

Návratová hodnota je rovna `true` jestliže `obj` je rovno aktuální instanci, jinak `false`.

```
public static bool Equals(object objA, object objB)
```

Určuje, zda specifikované objekty jsou si `objA` a `objB` rovny.

Návratová hodnota je `true`, jestliže `objA` určuje stejnou instanci jako `objB`, nebo jsou oba parametry `null`, nebo jestliže `objA.Equals(objB)` vrací `true`, jinak `false`.

```
public static bool ReferencesEquals(object objA, object objB)
```

Určuje, zda specifikované instance objektů `objA` a `objB` určují stejnou instanci. Návratová hodnota je `true`, jestliže `objA` určuje stejnou instanci jako `objB`, nebo jsou oba parametry `null`, jinak `false`.

```
public virtual string ToString()
```

Vrací objekt typu `string`, který reprezentuje aktuální objekt. Implementace metody ve třídě `object` vrací jméno typu instance. Zděděné typy obvykle přepisují tuto metodu a vracejí textovou reprezentující aktuální instance. Např. numerické typy vracejí textovou reprezentaci své číselné hodnoty.

Třída `string`

```
class string : object
```

Reprezentuje textové řetězce jako sekvenci znaků Unicode.

Konstruktory:

```
public string(char[] chars)
```

Inicializuje novou instanci třídy `string` na hodnotu, danou polem znaků.

```
public string(char[] chars, int startIndex, int length)
```

Inicializuje novou instanci třídy `string` na hodnotu danou polem znaků, pozicí počátečního znaku a délkou.

```
public string(char[] char, int count)
```

Inicializuje novou instanci třídy `string` na hodnotu danou opakováním znaku daným počtem.

Datové položky:

```
public static readonly string Empty
```

Reprezentuje prázdný řetězec `""`. Tato datová položka je pouze ke čtení.

Vlastnosti:

```
public int Length { get; }
```

Určuje počet znaku v aktuální instanci.

Metody:

```
public static int Compare(string strA, string strB)
```

Porovná dvě specifikované instance třídy `string`. Návratová hodnota je 32bitové číslo, které je záporné, pokud `strA` je menší než `strB`, je rovno nula, když `strA` je rovna `strB` nebo je větší než nula, pokud `strA` je větší než `strB`.

```
public static int Compare(string strA, string strB, bool ignoreCase)
```

Porovná dvě specifikované instance třídy `string`, parametr `ignoreCase` umožňuje určit zda při porovnání bude záležet na velikosti písmen. Návratová hodnota je 32bitové číslo, které je záporné, pokud `strA` je menší než `strB`, je rovno nula, když `strA` je rovna `strB` nebo je větší než nula, pokud `strA` je větší než `strB`.

```
public static string Concat(params object[] objects)
```

Vrací spojení řetězcových reprezentací prvků ve specifikovaném poli.

```
public static string Concat(params string[] strings)
```

Vrací spojení řetězců ve specifikovaném poli.

```
public static string Format(string format, params object[] objects)
```

Vytvoří kopii řetězce `format`, v níž nahradí formátovací položky v řetězci `format` textovými reprezentacemi odpovídajících objektů ve specifikovaném poli `objects`.

Řetězec `format` může obsahovat formátovací položky. Formátovací položky mají tvar `{` *index* [`,` *zarovnání*] `}`, která specifikuje povinný index a nepovinou délku a zarovnání formátovaného textu. *index* je celé nezáporné číslo, které určuje prvek pole `objects`, který má být formátován. *zarovnání* je celočíselná hodnota, která určuje minimální šířku oblasti, která bude obsahovat formátovanou hodnotu. Jestliže délka formátované hodnoty je menší než *zarovnání*, oblast je doplněna mezerami. Pokud je *zarovnání* záporné, text je zarovnán doleva, pokud je kladné, pak je text zarovnán doprava. Pokud *zarovnání* není uvedeno, délka oblasti bude stejná jako délka formátované hodnoty. Znaky `{` a `}` je možné ve formátovacím řetězci zapsat jako `{{` a `}}`.

```
public string Substring(int startIndex, int length)
```

Vrací nový řetězec, který odpovídá podřetězci aktuálního řetězce, který začíná znakem na indexu `startIndex` a má délku maximálně `length`.

```
public string ToUpper()
```

Vrací nový řetězec, který odpovídá aktuálnímu řetězci převedenému na velká písmena.

```
public string ToLower()
```

Vrací nový řetězec, který odpovídá aktuálnímu řetězci převedenému na malá písmena.

Typ `char`

```
class char : object
```

Reprezentuje 16bitový znak kódování Unicode.

Konstanty:

```
public const char MaxValue
```

Reprezentuje největší možnou hodnotu typu `char`. Hodnota této konstanty je hexadecimálně 0xFFFF.

`public const char MinValue`

Reprezentuje nejmenší možnou hodnotu typu `char`. Hodnota této konstanty je hexadecimálně 0x0000.

Metody:

`public static char ToUpper(char c)`

Vrací znak, jenž je ekvivalentem aktuální instance převedený na velké písmeno, nebo vrací `c`, pokud `c` je již velké písmeno, nebo znak `c` není písmeno.

`public static char ToLower(char c)`

Vrací znak, jenž je ekvivalentem aktuální instance převedený na malé písmeno, nebo vrací `c`, pokud `c` je již malé písmeno, nebo znak `c` není písmeno.

Typ `bool`

`class bool : object`

Reprezentuje booleovskou logickou hodnotu `true` nebo `false`.

Typ `byte`

`class byte : object`

Reprezentuje 8bitové celé číslo v rozsahu 0 až 255.

Konstanty:

`public const byte MaxValue`

Reprezentuje největší možnou hodnotu typu `byte`. Hodnota této konstanty je 255.

`public const byte MinValue`

Reprezentuje nejmenší možnou hodnotu typu `byte`. Hodnota této konstanty je 0.

Typ `short`

`class short : object`

Reprezentuje 16bitové celé číslo se znaménkem v rozsahu -32768 až 32767.

Konstanty:

`public const short MaxValue`

Reprezentuje největší možnou hodnotu typu `short`. Hodnota této konstanty je 32767.

`public const short MinValue`

Reprezentuje nejmenší možnou hodnotu typu `short`. Hodnota této konstanty je -32768.

Typ `int`

```
class int : object
```

Reprezentuje 32bitové celé číslo se znaménkem v rozsahu -2147483648 až 2147483647.

Konstanty:

```
public const int MaxValue
```

Reprezentuje největší možnou hodnotu typu `int`. Hodnota této konstanty je 2147483647.

```
public const int MinValue
```

Reprezentuje nejmenší možnou hodnotu typu `int`. Hodnota této konstanty je -2147483648.

Typ `long`

```
class long : object
```

Reprezentuje 64bitové celé číslo se znaménkem v rozsahu -9223372036854775808 až 9223372036854775807.

Konstanty:

```
public const long MaxValue
```

Reprezentuje největší možnou hodnotu typu `long`. Hodnota této konstanty je 9223372036854775807.

```
public const long MinValue
```

Reprezentuje nejmenší možnou hodnotu typu `long`. Hodnota této konstanty je -9223372036854775808.

Typ `float`

```
class float : object
```

Reprezentuje 32bitové číslo v plovoucí čárce dle standardu IEEE 754. Hodnoty reprezentované tímto typem jsou čísla z rozsahu -3.402823e38 až 3.402823e38 s přesností na 7 číslic, dále kladná i záporná nula, kladné a záporné nekonečno a hodnotu, která není číslo (NaN).

Konstanty:

```
public const float Epsilon
```

Reprezentuje nejmenší kladné číslo typu `float`. Hodnota této konstanty je 1.4e-45.

```
public const float MaxValue
```

Reprezentuje největší možné číslo typu `float`. Hodnota této konstanty je 3.402823e38.

```
public const float MinValue
```

Reprezentuje nejmenší možné číslo typu `float`. Hodnota této konstanty je -3.402823e38.

```
public const float NaN
```

Reprezentuje hodnotu typu `float`, která není číslo. Tato hodnota je např. výsledkem dělení nuly nulou. Pro testování hodnoty typu `float` na hodnotu `NaN` se používá metoda `IsNaN`.

```
public const float NegativeInfinity
```

Reprezentuje záporné nekonečno. Tato hodnota je např. výsledkem dělení záporného čísla nulou, nebo pokud výsledek některé operace je menší než `MinValue`. Pro testování hodnoty typu `float` na hodnotu `NegativeInfinity` se používá metoda `IsNegativeInfinity`.

```
public const float PositiveInfinity
```

Reprezentuje kladné nekonečno. Tato hodnota je např. výsledkem dělení kladného čísla nulou, nebo pokud výsledek některé operace je větší než `MaxValue`. Pro testování hodnoty typu `float` na hodnotu `PositiveInfinity` se používá metoda `IsPositiveInfinity`.

Metody:

```
public static bool IsNaN(float value)
```

Určuje, zda specifikovaná instance typu `float` je NaN.

```
public static bool IsInfinity(float value)
```

Určuje, zda specifikovaná instance typu `float` je kladné nebo záporné nekonečno.

```
public static bool IsNegativeInfinity(float value)
```

Určuje, zda specifikovaná instance typu `float` je záporné nekonečno.

```
public static bool IsPositiveInfinity(float value)
```

Určuje, zda specifikovaná instance typu `float` je kladné nekonečno.

Typ `double`

```
class double : object
```

Reprezentuje 64bitové číslo v plovoucí čárce dle standardu IEEE 754. Hodnoty reprezentované tímto typem jsou čísla z rozsahu -1.79769313486232e308 až 1.79769313486232e308 s přesností na 15 - 16 číslic, dále kladná i záporná nula, kladné a záporné nekonečno a hodnotu, která není číslo (NaN).

Konstanty:

```
public const double Epsilon
```

Reprezentuje nejmenší kladné číslo typu `double`. Hodnota této konstanty je 4.94065645841247e-324.

```
public const double MaxValue
```

Reprezentuje největší možné číslo typu `double`. Hodnota této konstanty je 1.79769313486232e308.

`public const double MinValue`

Reprezentuje nejmenší možné číslo typu `double`. Hodnota této konstanty je -1.79769313486232e308.

`public const double NaN`

Reprezentuje hodnotu typu `double`, která není číslo. Tato hodnota je např. výsledkem dělení nuly nulou. Pro testování hodnoty typu `double` na hodnotu `NaN` se používá metoda `IsNaN`.

`public const double NegativeInfinity`

Reprezentuje záporné nekonečno. Tato hodnota je např. výsledkem dělení záporného čísla nulou, nebo pokud výsledek některé operace je menší než `MinValue`. Pro testování hodnoty typu `double` na hodnotu `NegativeInfinity` se používá metoda `IsNegativeInfinity`.

`public const double PositiveInfinity`

Reprezentuje kladné nekonečno. Tato hodnota je např. výsledkem dělení kladného čísla nulou, nebo pokud výsledek některé operace je větší než `MaxValue`. Pro testování hodnoty typu `double` na hodnotu `PositiveInfinity` se používá metoda `IsPositiveInfinity`.

Metody:

`public static bool IsNaN(double value)`

Určuje, zda specifikovaná instance typu `double` je NaN.

`public static bool IsInfinity(double value)`

Určuje, zda specifikovaná instance typu `double` je kladné nebo záporné nekonečno.

`public static bool IsNegativeInfinity(double value)`

Určuje, zda specifikovaná instance typu `double` je záporné nekonečno.

`public static bool IsPositiveInfinity(double value)`

Určuje, zda specifikovaná instance typu `double` je kladné nekonečno.

Typ `decimal`

`class decimal : object`

Reprezentuje desetinná čísla v rozsahu -79228162514264337593543950335 až 79228162514264337593543950335 s přesností na 28 - 29 číslic. Instance typu `decimal` mají 128 bitů, 1 bit zabírá znaménko, 96 bitů celočíselná hodnota a zbytek zabírá dělící faktor 0 - 28, který představuje mocniny 10.

Konstanty:

`public const decimal MaxValue`

Reprezentuje největší možnou hodnotu typu `decimal`. Hodnota této konstanty je 79228162514264337593543950335.

`public const decimal MinValue`

Reprezentuje nejmenší možnou hodnotu typu `decimal`. Hodnota této konstanty je -79228162514264337593543950335.

Metody:

Třída `Array`

```
abstract class Array : object
```

Představuje základní typ všech polí, poskytuje společné metody a vlastnosti.

Vlastnosti:

```
public int Length { get; }
```

Vrací počet prvků ve všech dimenzích aktuální instance pole dohromady.

```
public int Rank { get; }
```

Vrací počet dimenzí pole.

Metody:

```
public int GetLength(int dim)
```

Vrací počet prvků aktuální instance ve specifikované dimenzi `dim`. Dimenze jsou číslovány od nuly.

```
public object GetValue(int[] indices)
```

Vrací hodnotu na specifikované pozici v aktuální instance typu `Array`. Pozice je dána polem indexů `indices`. Počet indexů musí být roven dimenzi aktuální instance. Tato metoda se používá v případě, kdy dimenze pole není známa při překladu. V opačném případě je vhodné použít syntaxi přístupu k prvku pole pomocí `[]`.

```
public void SetValue(object obj, int[] indices)
```

Nastavuje hodnotu na specifikované pozici v aktuální instanci typu `Array` na hodnotu `obj`. Pozice je dána polem indexů `indices`. Počet indexů musí roven dimenzi aktuální instance. Tato metoda se používá v případě, kdy dimenze pole není známa při překladu. V opačném případě je vhodné použít syntaxi přístupu k prvku pole pomocí `[]`.

Třída `Enum`

```
abstract class Enum : object
```

Představuje základní typ všech výčtových typů a poskytuje společné metody.

Metody:

```
public override bool Equals(object obj)
```

Určuje, zda aktuální instance a specifikovaný objekt `obj` mají stejnou hodnotu. Návratová hodnota je `true`, jestliže `obj` je stejného typu jako aktuální instance a číselná hodnota `obj` je stejná jako číselná hodnota aktuální instance, jinak `false`.

```
public override string ToString()
```

Vrací textovou reprezentaci hodnoty aktuální instance. Jestliže hodnota aktuální instance je rovna některé pojmenované konstantě výčtu, metoda vrací jméno této konstanty, jinak číselnou reprezentaci.

Třída `Console`

```
class Console : object
```

Reprezentuje standardní vstupní a výstupní textové streamy programu.

Metody:

```
public static void Write(object value)
```

Zapíše textovou reprezentaci objektu `value` do standardního výstupního streamu programu.

```
public static void Write(string format, params object[] objects)
```

Zapíše textovou reprezentaci objektů v poli `objects` do standardního výstupního streamu programu s použitím formátovacího řetězce `format`.

```
public static void WriteLine()
```

Zapíše na standardní výstupní stream konec řádku.

```
public static void WriteLine(object value)
```

Zapíše textovou reprezentaci objektu `value` následovanou koncem řádku do standardního výstupního streamu programu.

```
public static void Write(string format, params object[] objects)
```

Zapíše textovou reprezentaci objektů v poli `objects` následovanou koncem řádku do standardního výstupního streamu programu s použitím formátovacího řetězce `format`.

```
public static int Read()
```

Přečte a vrátí další znak ze standardního vstupního streamu programu. Znaky jsou reprezentovány jako čísla typu `int`. Jestliže na vstupu není žádný znak, vrací hodnotu -1.

```
public static string ReadLine()
```

Přečte další řádek ze standardního vstupního streamu programu a vrátí jej jako řetězec typu `string`.

Závěr

Cílem této diplomové práce bylo návrhnout a implementovat interpret objektově orientovaného programovacího jazyka. Zvolen byl interpret a nikoliv klasický kompilátor, neboť interpret neobsahuje generování spustitelných souborů programu pro určitou softwarovou a hardwarovou platformu, které je poměrně složité a přesahuje rámec rozsahu diplomové práce.

Nejdříve bylo potřeba navrhnout a specifikovat předmětný programovací jazyk. Jako vzor byl vybrán v dnešní době dobře známý jazyk C#. Opět z důvodu rozsahu práce nebyly do vytvářeného jazyka zahrnuty prvky, které nejsou důležité pro objektově orientovaný programovací jazyk. Nový jazyk byl nazván MiniC#.

Součástí specifikace bylo také vytvoření lexikální a syntaktické gramatiky jazyka. Při převodu syntaktické gramatiky na tvar $LALR(1)$ bylo potřeba řešit několik problémů způsobujících konflikty v konstrukci deterministického analyzátoru.

Pro implementaci byl zvolen programovací jazyk C# a vývojové prostředí MS Visual Studio 2005.

Zdrojové kódy lexikálního i syntaktického analyzátoru byly generovány automaticky programem ParserGenerator (viz [10]).

Nejrozsáhlejší částí byl návrh a implementace sémantického analyzátoru, neboť jazyk MiniC# obsahuje velké množství různorodých konstruktů – deklarací, příkazů, výrazů, a dále obsahuje poměrně přísná a propracovaná sémantická pravidla.

Poslední částí bylo vytvoření knihovny základních typů. Ta obsahuje jednak implementaci typů, které přímo vyžaduje kompilátor a jednak typy implementující některé systémové služby, aby bylo možno v tomto jazyce psát užitečné programy.

Přínosem této práce bylo prohloubení a osvojení poznatků z daných oblastí informatiky a dále vznik programovacího jazyka a interpretu, který je možno využít např. při výuce programování nebo konstrukce překladačů.

Výsledky uvedených prací jsou uvedeny v jednotlivých sekcích této diplomové práce.

Conclusions

The aim of this thesis was to design and implement an interpretative compiler of object-oriented programming language. The compiler was designed as interpretative because it does not involve generating of platform-dependent executable files, which is very complicated and is above the range of a thesis.

First, an objective programming language had to be designed and specified. C# language, which is currently well-known, was chosen as a model for this new language. Some C# features that are not important for object-oriented languages were not considered because it would go beyond the range of a thesis. The new language was named MiniC#.

Part of the language specification was to establish lexical and syntactic grammars. While converting syntactic grammar to $LALR(1)$ form there occurred some problems causing conflicts in construction of deterministic parser, and these had to be solved.

C# programming language and MS Visual Studio 2005 development environment were chosen for the implementation of compiler application.

Source codes of both lexical and syntactic analyzers were automatically generated using the ParserGenerator program (see [10]).

The most extensive part of this work ware design and implementation of a semantic analyzer because MiniC# language contains a large number of various constructs – declarations, statements, expressions, and also strict and precise semantic rules.

Last, the base types library had to be implemented. This library includes compiler-needed types and types that implement system services for writing useful programs.

The benefits of this thesis include the acquisition of knowledge from specific parts of computer science and also creation of a new programming language and compiler which can be used in education of programming or compiler design.

Results of the parts of work mentioned above are presented in the individual sections of this thesis.

Reference

[1] Češka, M. – Rábová, Z. *Gramatiky a jazyky.*
Nakladatelství VUT, Brno, 1992.

[2] Donnelly, C. – Stallman, R. *Bison, The YACC-compatible Parser Generator.*
Free Software Foundation, Boston, 1995.
http://www.gnu.org/software/bison/bison.html

[3] Gosling, J. – Joy, B. – Steele, G. – Bracha, G. *The Java Language Specification.*
Third Edition, Addison-Wesley, Boston, 2005.

[4] Kravál, I. *Základy objektově orientovaného programování.*
Computer Press, Praha, 1999.

[5] Melichar, B. a kol. *Konstrukce překladačů.*
Nakladatelství VUT, Praha, 2000.

[6] Microsoft Corporation. *C# Language Specification Version 1.2.*
Elektronická publikace, 2003.
http://msdn.microsoft.com

[7] Muchnick, S. S. *Compiler Design & Implementation.*
Morgan Kaufmann, San Francisco, 1997.

[8] Richter, J. *.NET Framework – programování aplikací.*
Grada Publishing, Praha, 2003.

[9] Skoupil, D. *Úvod do paradigmat programování.*
Technický report, KMI PřF UP, Olomouc, 1994

[10] Šimek, R. *Generátor syntaktických analyzátorů.*
Bakalářská práce, KMI PřF UP, Olomouc, 2004

[11] *Unicode Standard Specification.*
http://www.unicode.org

[12] Večerka, A. *Jazyk C++, Popis jazyka s příklady.*
Vydavatelství UP, Olomouc, 2000

[13] Vychodil, V. *Konstrukce zásobníkového automatu LALR(1).*
Elektronická publikace, 2001.
http://vychodil.ing.upol.cz

[14] Wilhelm, R. – Maurer, D. *Compiler Design.*
Addison–Wesley, Reading, MA, 2001.

A. Lexikální gramatika jazyka MiniC#

V této příloze je uvedena kompletní lexikální gramatika jazyka MiniC#.

Gramatika je uvedena výčtem regulárních výrazů lexikálních elementů jazyka. Syntaxe regulárních výrazů odpovídá syntaxi regulárních výrazů platformy .NET Framework.

whitespace:
```
([\u0009\u000B\u000C]|\p{Zs})+
```

lineterminator:
```
[\u000D\u000A\u2028\u2029]|\u000D\u000A
```

singlelinecomment:
```
//[^\u000D\u000A\u2028\u2029]*
```

multilinecomment:
```
/\*([^*]|(\u000A|\u000D\u000A)|\*+[^/])*\*+/
```

Lexikální symboly

identifier:
```
(\p{Lu}|\p{Ll}|\p{Lt}|\p{Lm}|\p{Lo}|\p{Nl}|_)
(\p{Lu}|\p{Ll}|\p{Lt}|\p{Lm}|\p{Lo}|\p{Nl}|\p{Mn}|
 \p{Mc}|\p{Nd}|\p{Pc}|\p{Cf})*
```

decimalintegerliteral:
```
[0-9]+(L|l)?
```

hexadecimalintegerliteral:
```
(0x|0X)([0-9A-Fa-f])+(L|l)?
```

realliteral:
```
([0-9]+)?\.[0-9]+((E|e)(\+|-)?[0-9]+)?(F|f|D|d|M|m)?|
[0-9]+(E|e)(\+|-)?[0-9]+(F|f|D|d|M|m)?|[0-9]+(F|f|D|d|M|m)
```

characterliteral:
```
'([^'\\\u000D\u000A\u2028\u2029]|\\'|\\"|\\\\|\\a|\\b|
  \\f|\\n|\\r|\\t|\\v|
  \\x[0-9a-fA-F]{1,4}|\\u[0-9a-fA-F]{4})'
```

regularstringliteral:
```
"([^"\\\u000D\u000A\u2028\u2029]|\\'|\\"|\\\\|\\a|\\b|
 \\f|\\n|\\r|\\t|\\v|
 \\x[0-9a-fA-F]{1,4}|\\u[0-9a-fA-F]{4})*"
```

verbatimstringliteral:
```
@"([^"]|"")*"
```

Klíčová slova:

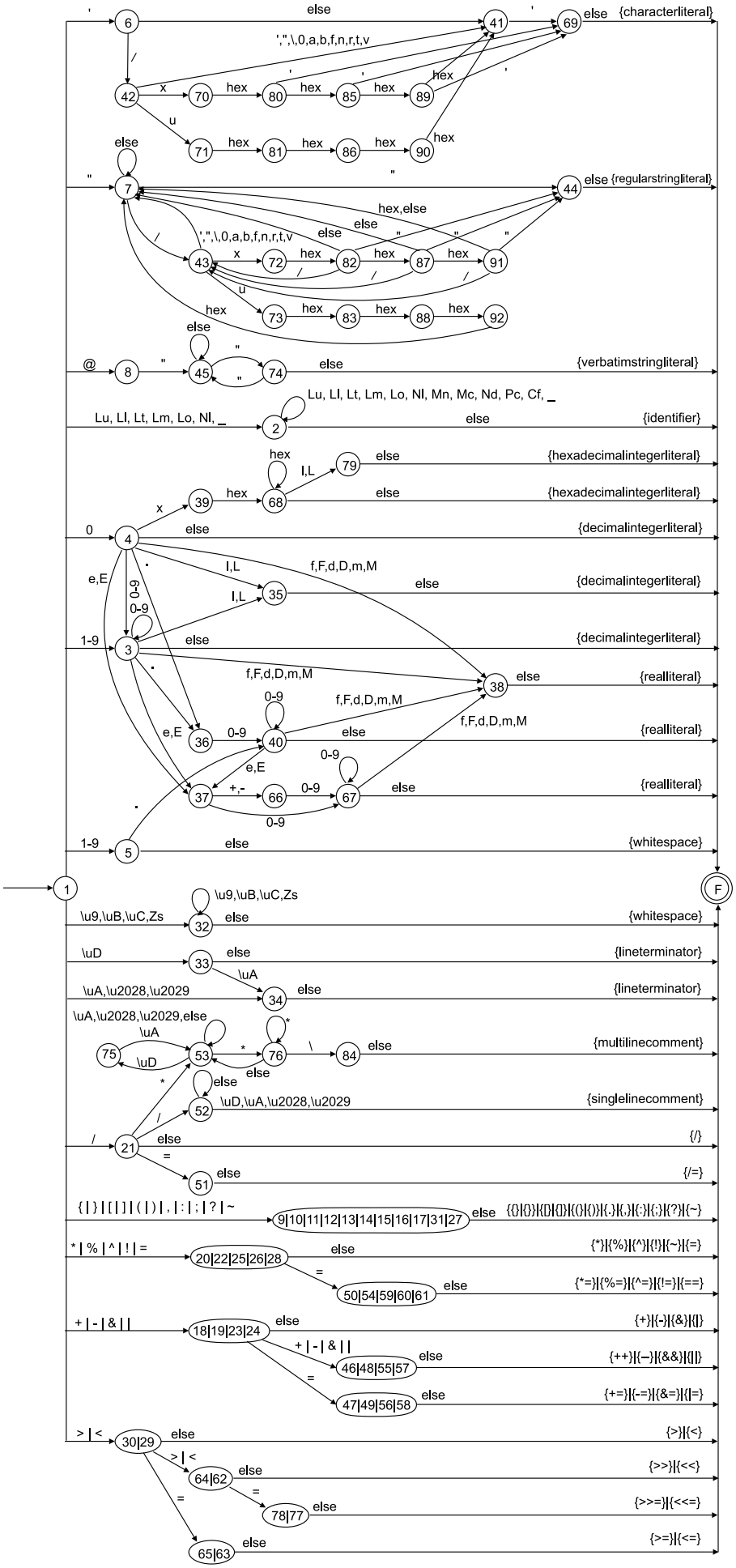
abstract	as	base	bool	break	byte	case	char
class	const	continue	decimal	default	do	double	else
else	enum	float	for	foreach	if	in	int
is	long	new	object	out	override	params	private
protected	public	readonly	ref	return	short	static	string
switch	this	virtual	void	while	true	false	null

Operátory, interpunkce:

{	}	[	]	(	)	.	,	:	;	+	-	*	/	%	&
|	^	!	~	=	<	>	?	++	--	&&	||	<<	>>	==	!=
<=	>=	+=	-=	*=	/=	%=	&=	|=	^=	<<=	>>=				

Schéma lexikálního analyzátoru jazyka MiniC# je na následujícím obrázku. Přechody označené else lze provézt při jakémkoliv jiném znaku na vstupu, než jaké jsou uvedeny na dalších přechodech vycházejících z téhož stavu a přechody označené hex značí všechny hexadecimální číslice.

Přechody a stavy označené více znaky a čísly oddělenými |, značí více přechodů a stavů se stejnou strukturou.


1
F
{characterliteral}
{regularstringliteral}
{verbatimstringliteral}
{identifier}
{hexadecimalintegerliteral}
{decimalintegerliteral}
{realliteral}
{whitespace}
{lineterminator}
{multilinecomment}
{singlelinecomment}
{/}
{/=}
{}|{}}|{[}|{]}|{(}|{)}|{.}|{,}|{:}|{;}|{?}|{~}
{*}|{%}|{^}|{!}|{~}|{=}
{*=}|{%=}|{^=}|{!=}|{==}
{+}|{-}|{&}|{|}
{++}|{--}|{&&}|{||}
{+=}|{-=}|{&=}|{|=}
{>}|{<}
{>>}|{<<}
{>>=}|{<<=}
{>=}|{<=}
else
hex
0-9
e,E
f,F,d,D,m,M
l,L
\u9,\uB,\uC,Zs
\uD
\uA
\uA,\u2028,\u2029
\uA,\u2028,\u2029,else
\uD,\uA,\u2028,\u2029
Lu, Ll, Lt, Lm, Lo, Nl, Mn, Mc, Nd, Pc, Cf, _
Lu, Ll, Lt, Lm, Lo, Nl, _
',",\,0,a,b,f,n,r,t,v
{ | } | [|] | (|) | , | : | ; | ? | ~
9|10|11|12|13|14|15|16|17|31|27
* | % | ^ | ! | =
20|22|25|26|28
50|54|59|60|61
+ | - | & | |
18|19|23|24
46|48|55|57
47|49|56|58
> | <
30|29
64|62
78|77
65|63

B. Syntaktická gramatika jazyka MiniC#

V této příloze je uvedena kompletní syntaktická gramatika jazyka MiniC#. Tato gramatika patří do třídy $LALR(1)$ gramatik.

Gramatika je uvedena výčtem pravidel. Terminálními symboly jsou všechny symboly uvedené `neproporcionálním` písmem a symboly *realliteral*, *characterliteral*, *decimalintegerliteral*, *hexadecimalintegerliteral*, *regularstringliteral* a *verbatimstringliteral*. Ostatní uvedené symboly jsou neterminální symboly. Startovním symbolem gramatiky je symbol *compilationunit*.

V uvedené gramatice jsou symboly *[ ]* označeny nepovinné části pravidel.

compilationunit : *typedeclarations*
typedeclarations : *[typedeclarations] typedeclaration*
typedeclaration : *classdeclaration*
| *enumdeclaration*
enumdeclaration : `enum` *identifier enumbody [;]*
enumbody : `{` *[enummemberdeclarations]* `}`
| `{` *enummemberdeclarations* `,` `}`
enummemberdeclarations : *[enummemberdeclarations ,] enummemberdeclaration*
enummemberdeclaration : *identifier [* `=` *constantexpression]*
classdeclaration : *[* `abstract` *]* `class` *identifier [* `:` *type] classbody [* `;` *]*
classbody : `{` *classmemberdeclarations* `}`
classmemberdeclarations : *[classmemberdeclarations] classmemberdeclaration*
classmemberdeclaration : *[classmembermodifiers] constantdeclaration*
| *[classmembermodifiers] fielddeclaration*
| *[classmembermodifiers] methoddeclaration*
| *[classmembermodifiers] propertydeclaration*
| *[classmembermodifiers] constructordeclaration*
classmembermodifiers : *[classmembermodifiers] classmembermodifier*
classmembermodifier : `public`
| `protected`
| `private`
| `static`
| `readonly`
| `virtual`
| `override`
| `abstract`
constantdeclaration : `const` *type constantdeclarators* `;`
constantdeclarators : *[constantdeclarators ,] constantdeclarator*
constantdeclarator : *identifier* `=` *constantexpression*
fielddeclaration : *type variabledeclarators* `;`
variabledeclarators : *[variabledeclarators ,] variabledeclarator*
variabledeclarator : *identifier [* `=` *variableinitializer]*
variableinitializer : *expression*
| *arrayinitializer*
methoddeclaration : *methodheader methodbody*
methodheader : *type identifier* `(` *[formalparameterlist]* `)`
| `void` *identifier* `(` *[formalparameterlist]* `)`
methodbody : *block*
| `;`
formalparameterlist : *fixedparameters [* `,` *parameterarray]*
| *parameterarray*

fixedparameters : *[fixedparameters* `,` *fixedparameter]*
fixedparameter : *[parametermodifier] type identifier*
parametermodifier : `ref`
| `out`
parameterarray : `params` *type identifier*
propertydeclaration : *type identifier* `{` *accessordeclarations* `}`
accessordeclarations : *[accessordeclaration] [accessordeclaration]*
accessordeclaration : *identifier accessorbody*
accessorbody : *block*
| `;`
constructordeclaration : *constructordeclarator constructorbody*
constructordeclarator : *identifier* `(` *[formalparameterlist]* `)` *[constructorinitializer]*
constructorinitializer : `:` `base` `(` *[argumentlist]* `)`
| `:` `this` `(` *[argumentlist]* `)`
constructorbody : *block*
| `;`
type : *nonarraytype*
| *arraytype*
nonarraytype : *predefinedtype*
| *identifier*
predefinedtype : `bool`
| `byte`
| `char`
| `decimal`
| `double`
| `float`
| `int`
| `long`
| `short`
| *predefinedreferencetype*
predefinedreferencetype : `object`
| `string`
arraytype : *predefinedtype rankspecifier*
| *identifier rankspecifier*
classtype : *identifier*
| *predefinedreferencetype*
rankspecifier : `[` *[dimseparators]* `]`
dimseparators : *[dimseparators]* `,`
arrayinitializer : `{` *[variableinitializerlist]* `}`
| `{` *variableinitializerlist* `,` `}`
variableinitializerlist : *[variableinitializerlist ,] variableinitializer*
expression : *assignment*
| *conditionalexpression*
assignment : *unaryexpression assignmentoperator expression*
assignmentoperator : `-=`
| `+=`
| `=`
| `*=`
| `/=`
| `%=`
| `&=`
| `|=`
| `^=`

| `<<=`
| `>>=`

conditionalexpression : *conditionalorexpression [* `?` *expression* `:` *expression]*
conditionalorexpression : *[conditionalorexpression* `||` *] conditionalandexpression*
conditionalandexpression : *[conditionalandexpression* `&&` *] inclusiveorexpression*
inclusiveorexpression : *[inclusiveorexpression* `|` *] exclusiveorexpression*
exclusiveorexpression : *[exclusiveorexpression* `^` *] andexpression*
andexpression : *[andexpression* `&` *] equalityexpression*
equalityexpression : *[equalityexpression equalityoperator] relationalexpression*
equalityoperator : `==`
| `!=`
relationalexpression : *[relationalexpression relationaloperator] shiftexpression*
| *relationalexpression* `is` *type*
| *relationalexpression* `as` *type*
relationaloperator : `>`
| `>`
| `<=`
| `>=`
shiftexpression : *[shiftexpression shiftoperator] additiveexpression*
shiftoperator : `<<`
| `>>`
additiveexpression : *[additiveexpression additiveoperator] multiplicativeexpression*
additiveoperator : `+`
| `-`
multiplicativeexpression : *[multiplicativeexpression multiplicatoveoperator] unaryexpression*
multiplicativeoperator : `*`
| `/`
| `%`
unaryexpression : `+` *unaryexpression*
| `-` *unaryexpression*
| `++` *unaryexpression*
| `--` *unaryexpression*
| *unaryexpressionnotpm*
unaryexpressionnotpm : `!` *unaryexpression*
| `~` *unaryexpression*
| *castexpression*
| *primaryexpression*
castexpression : `(` *predefinedtype [rankspecifier]* `)` *unaryexpression*
| `(` *expression* `)` *unaryexpressionnotpm*
| `(` *identifier rankspecifier* `)` *unaryexpression*
primaryexpression : *primarynonewarrayexpr*
| *arraycreationexpression*
| *identifier*
primarynonewarrayexpr : *literal*
| `this`
| `(` *expression* `)`
| *memberaccess*
| *objectcreationexpression*
| *elementaccess*
| *invocationexpression*
| *primaryexpression* `++`
| *primaryexpression* `--`
memberaccess : *primaryexpression* `.` *identifier*

| *predefinedtype* `.` *identifier*
| `base` `.` *identifier*

invocationexpression : *memberaccess* `(` *[argumentlist]* `)`
| *identifier* `(` *[argumentlist]* `)`

argumentlist : *[argumentlist* `,` *]* *argument*

argument : *expression*
| `ref` *expression*
| `out` *expression*

elementaccess : *primarynonewarrayexpr* `[` *expressionlist* `]`
| *identifier* `[` *expressionlist* `]`

expressionlist : *[expressionlist* `,` *]* *expression*

objectcreationexpression : `new` *classtype* `(` *[argumentlist]* `)`

arraycreationexpression : `new` *nonarraytype* `[` *expressionlist* `]` *[arrayinitializer]*
| `new` *nonarraytype rankspecifier arrayinitializer*

constantexpression : *expression*

booleanexpression : *expression*

statement : *declarationstatement*
| *embeddedstatement*

declarationstatement : *type variabledeclarators* `;`
| `const` *type constantdeclarators* `;`

embeddedstatement : *block*
| `;`
| *expression* `;`
| *ifstatement*
| *switchstatement*
| *whilestatement*
| *dostatement*
| *forstatement*
| *foreachstatement*
| `break` `;`
| `continue` `;`
| `return` `;`
| `return` *expression* `;`

block : `{` *[statementlist]* `}`

statementlist : *statement*
| *statementlist statement*

ifstatement : `if` `(` *booleanexpression* `)` *embeddedstatement [* `else` *embeddedstatement]*

switchstatement : `switch` `(` *expression* `)` *switchblock*

switchblock : `{` *switchsections* `}`

switchsections : *[switchsections]* *switchsection*

switchsection : *switchlabels statementlist*

switchlabels : *[switchlabels]* *switchlabel*

switchlabel : `case` *constantexpression* `:`
| `default` `:`

whilestatement : `while` `(` *booleanexpression* `)` *embeddedstatement*

dostatement : `do` *embeddedstatement* `while` `(` *booleanexpression* `)` `;`

forstatement : `for` `(` *[forinitializer]* `;` *[forcondition]* `;` *[foriterator]* `)` *embeddedstatement*

forinitializer : *localvariabledeclaration*
| *expressionlist*

forcondition : *booleanexpression*

foriterator : *expressionlist*

foreachstatement : `foreach` `(` *type identifier* `in` *expression* `)` *embeddedstatement*

literal | *realliteral*

| *characterliteral*
: *booleanliteral*
| *integerliteral*
| *stringliteral*
| *nullliteral*

booleanliteral : `true`
| `false`

integerliteral : *decimalintegerliteral*
| *hexadecimalintegerliteral*

stringliteral : *regularstringliteral*
| *verbatimstringliteral*

nullliteral : `null`