VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Implementace podpory Spark View Engine do Visual Studio 201x

Implementation of Support for Spark View Engine into Visual Studio 201x

2013 Bc. Karel Urbánek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Karel Urbánek**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Implementace podpory Spark View Engine do Visual Studio 201x
Implementation of Support for Spark View Engine into Visual Studio 201x

Zásady pro vypracování:

Cílem práce je naimplementovat podporu pro psaní šablon ve Spark View Engine (http://sparkviewengine.codeplex.com/) do IDE Microsoft Visual Studio (MS VS).

1. Prozkoumejte vlastnosti knihovny Spark View Engine a jejího použití v rámci prostředí ASP.NET.
2. Prozkoumejte možnosti implementace rozšíření pro Microsoft Visual Studio se zaměřením na editaci souborů zdrojového kódu.
3. Naimplementujte rozšíření Microsoft Visual Studio, které bude zvýrazňovat syntaxi v šablonách napsaných pro Spark View Engine.
4. Do rozšíření z bodu 3. doplňte IntelliSense pro jazyk C# v příslušných blocích kódu šablony.
5. Sestavte návod k využití knihovny Spark View Engine a naimplementovaného rozšíření.

Seznam doporučené odborné literatury:

[1] Spark Documentation <http://sparkviewengine.com/documentation>
[2] MSDN, Visual Studio - Extending the Editor <http://msdn.microsoft.com/library/dd885242%28VS.100%29.aspx>
[3] Dr. Ing. Miroslav Beneš. Překladače. Skriptum VŠB-TU Ostrava
[4] Aho, Alfred V., et al. Compilers: principles, techniques, and tools. Vol. 1009. Pearson/Addison Wesley, 2007.
[5] Carmo, J., Carlos Videira, and A. Silva. "Using Visual Studio Extensibility Mechanisms for Requirements Specification." 1st Conference on Innovative Views on .NET Technologies, Porto. 2005.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Jakub Macek**

Datum zadání: 16.11.2012

Datum odevzdání: 07.05.2013

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 2. května 2013

Karel Urbánek

.....

Rád bych na tomto místě poděkoval svému vedoucímu diplomové práce Ing. Jakubu Mackovi.

Abstrakt

Práce se zabývá popisem implementace podpory pro psaní šablon ve *Spark View Engine* ve vývojovém prostředí *Microsoft Visual Studio 2012*. V práci je popsána implementace zvýrazňování syntaxe v šablonách napsaných pro Spark View Engine, a dále pak implementace doplňování slov v šablonách napsaných pro Spark View Engine. Popisovaná funkcionalita je realizována formou *VSIX* doplňku pro Visual Studio 2012, za použití *Managed Exntensibility Frameworku* v prostředí *.NET Framework 4.5*. Popisované řešení je postaveno na principech lexikální, syntaktické a sémantické analýzy zdrojového kódu ve Spark View Engine šablonách. K těmto analýzám je využito knihovny *Irony .NET Language Implementation Kit* a také knihovny *Microsoft Roslyn.*

Klíčová slova: bezkontextové gramatiky, lexikální analýza, syntaktická analýza, sémantická analýza, LR syntaktické analyzátory, LALR syntaktické analyzátory, zvýrazňování syntaxe, doplňování slov, Spark View Engine, Irony .NET Language Implementation Kit, C#, Microsoft .NET Framework, Managed Extensibility Framework, Visual Studio 2012, Microsoft Roslyn

Abstract

The thesis contains a description of an implementation of support for writing templates using *Spark View Engine* in the IDE *Microsoft Visual Studio 2012*. The thesis describes an implementation of syntax highlighting in Spark View Engine templates as well as implementation of statement completion in Spark View Engine templates. The functionality described is realized as a VSIX extension to Visual Studio 2012 with the assistance of *Managed Extensibility Framework* using *.NET Framework 4.5.*. The said implementation is based on the principles of lexical, syntactic and semantic analysys of source code contained in Spark View Engine templates. These analyses are carried out by utilizing the libraries *Irony .NET Language Implementation Kit* and *Microsoft Roslyn.*

Keywords: context-free grammars, lexical analysis, syntactic analysis, semantic analysis, LR parsers, LALR parsers, syntax highlighting, statement completion, Spark View Engine, Irony .NET Language Implementation Kit, C#, Microsoft .NET Framework, Managed Extensibility Framework, Visual Studio 2012, Microsoft Roslyn

Seznam použitých zkratek a symbolů

API	–	Application Programing Interface
AST	–	Abstract Syntax Tree
BNF	–	Backus-Naur Form
EBNF	–	Extended Backus-Naur Form
EOF	–	End of File
EOS	–	End of Statement
LR	–	Left-to-right Rightmost
LALR	–	Look Ahead Left-to-right Rightmost
MEF	–	Managed Extensibility Framework
XML	–	eXtensible Markup Language
HTML	–	HyperText Markup Language
WPF	–	Windows Presentation Foundation

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Zvýrazňování syntaxe a doplňování slov jsou dnes standardní vlastnosti většiny editorů zdrojového kódu, jelikož se nemalou měrou podílí na zvýšení produktivity práce při psaní kódu.

Při zvýraznění syntaxe jsou jednotlivé symboly daného programovacího jazyka zobrazeny různou barvou, fontem nebo podbarvením, a to v závislosti na typu těchto symbolů. Zvýraznění syntaxe bývá také doplňováno o detekci syntaktických chyb, kdy chybové úseky kódu jsou nějakým způsobem odlišeny od úseků syntakticky bezchybných. Zvýrazněná syntaxe zlepšuje čitelnost kódu, a programátor se může v takto formátovaném kódu mnohem lépe a rychleji orientovat. Na první pohled je vidět, o jakou část kódu se jedná, a programátor může snadno přeskakovat ty části, které ho nezajímají, případně se zaměřit na ty části, které jsou pro něj v danou chvíli relevantní.

Doplňování slov (angl. *intellisense* nebo *statement completion*) pak představuje další důležitý aspekt zvyšující produktivitu práce v daném jazyku. Technologie doplňování slov je ve většině případů implementována jako malé okno, které automaticky vyskakuje v okně editoru v závislosti na pozici, na které se nachází kurzor. V okně intellisense jsou zobrazena slova, jež lze doplnit na aktuální pozici v editoru. Na pozadí této technologie probíhá syntaktická a sémantická analýza kódu, která bere v úvahu aktuální kontext a existující zdrojový kód. Výsledkem těchto analýz jsou textové řetězce, představující validní doplnění.

V práci se zabývám popisem implementace zvýrazňování syntaxe a doplňování slov pro šablony *Spark view engine* ve vývojovém prostředí Visual Studio 2012. Spark view engine, zkráceně *Spark*, je alternativním řešením k *ASP.NET* a *ASP.NET Razor view engine*. Tato řešení se používají pro implementaci webových stránek, za použití návrhového vzoru Model-View-Controller v prostředí Microsoft .NET frameworku. Jak Razor tak ASP.NET jsou technologie vyvíjené a podporované firmou Microsoft, a mají tedy plnou podporu v prostředí Microsoft Visual Studio, a to včetně zvýrazňování syntaxe a doplňování slov. Spark je však komunitní projekt, a jako takový dosud neměl podporu zvýrazňování syntaxe ani doplňování slov v prostředí Visual Studia 2012. Tato situace se však mění s publikací řešení, které je předmětem této práce.

Práci lze rozdělit do tří částí: první část—sekce 2 a 3—podává základní teoretický přehled o lexikální a syntaktické analýze, na nichž je postavené celé mnou implementované řešení zvýrazňování syntaxe a doplňování slov.

V druhé části se zabývám popisem knihoven a frameworků, které jsou použity ve výsledném řešení. Jedná se o sekci 4, v níž je nejprve vysvětlen princip činnosti *Irony*—generátoru lexikálních a syntaktických analyzátorů. Následně je na praktických příkladech demonstrováno použití tohoto frameworku. Dále v sekci 5 popisuji principy, na kterých je založena funkcionalita umožňující rozšiřovat prostředí Visual Studia o nové vlastnosti. V neposlední řadě se v kapitole 6 zabývám popisem knihovny Microsoft Roslyn, kterou jsem použil především pro syntaktickou a sémantickou analýzu úseků kódu Sparku, jenž přebírají syntaxi z jazyka C#.

Ve třetí, poslední, části následuje vlastní popis mnou implementovaného řešení. Nej-

prve jsou v sekci 7 popsány bezkontextové gramatiky použité pro generování lexikálních a syntaktických analyzátorů kódu Sparku. Poté následuje popis implementace zvýrazňování syntaxe v šablonách Spark View Engine—sekce 8—a nakonec popisuji způsob, jakým jsem naprogramoval funkcionalitu doplňování slov.

V přílohách je zahrnuto zjednodušené schéma, které znázorňuje propojení a komunikaci mezi jednotlivými komponentami, jež zajišťují funkcionalitu zvýrazňování syntaxe a doplňování slov. Druhá příloha pak obsahuje stručný návod, popisující jednotlivé kroky, které je nutné dodržet, aby bylo možno úspěšně nainstalovat a používat naimplementované řešení.

2 Lexikální analýza

2.1 Motivace

Jelikož se v práci zabývám implementací zvýrazňování syntaxe, bylo potřeba využít nějakého mechanizmu, který by mi umožnil roztřídit slova a znaky zapsané ve Spark šabloně do jednotlivých kategorií. Tyto kategorie jsou tvořeny *lexikálními jednotkami* (angl. *tokeny*). Druhy lexikálních jednotek mohou například být (příklady uvedené v závorkách jsou v jazyce C#):

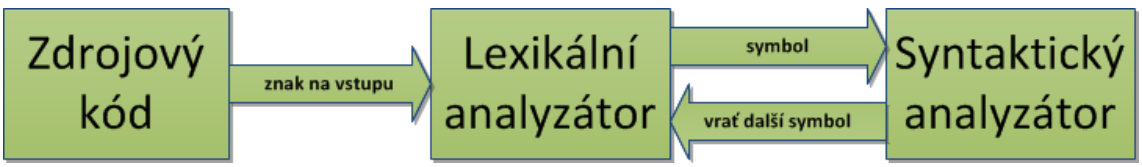
- klíčová slova (`int`, `true`, `if`),
- operátory (`>=`, `++`),
- textové řetězce a znaky (`"řetězec"`, `'a'`),
- číselné konstanty (`5`, `3.4f`, `0x4b`),
- komentáře (`//zakomentovaný kód`, `/*víceřádkový komentář*/`
- ...

Jakmile jsou znaky a slova zpracovány jako lexikální jednotky, zbývá už jen přiřadit *barvu* či barevný podklad (v závislosti na možnostech editoru zdrojového kódu) každému možnému typu lexikální jednotky. V editoru zdrojového kódu se pak daný znak nebo slovo zobrazí napsán přiřazenou barvou (popř. na přiřazeném barevném podkladu).

K vyřešení tohoto problému klasifikace znaků a symbolů se přímo nabízí mechanizmus *lexikálních analyzátorů* (angl. *lexer*, *scanner* nebo *lexical analyzer*), který využívá bezkontextových gramatik a konečných automatů.

2.2 Princip lexikální analýzy

Lexikální analýza (angl. *lexical analysis* nebo *scanning*) je proces, při kterém jsou postupně zleva doprava čteny znaky ze vstupního souboru (např. soubor obsahující definici třídy v objektově orientovaném jazyce) a z těchto znaků jsou sestavovány tzv. *lexémy* (angl. *lexeme*) nebo-li lexikální jednotky. Lexémy jsou poté dále zpracovávány ve fázi syntaktické analýzy, o které pojednává sekce 3. Spolupráci lexikálního analyzátoru se syntaktickým schématicky znázorňuje obrázek 1. Syntaktický analyzátor neustále posílá požadavky na



Obrázek 1: Schématické znázornění interakce lexikálního analyzátoru

další *symboly*, které lexikální analyzátor získává klasifikací lexémů identifikovaných ve

vstupním textu. Symbol (angl. *token*) je kategorie, do které spadá rozpoznaný lexém. Symbol je prakticky reprezentován jako číselná hodnota, případně jako hodnota výčtového typu. V syntaktické analýze (viz podsekce 3) je místo symbolu užíván výraz *terminál*.

2.2.1 Lexémy a symboly

Obecně lze říci, že každý lexém spadá do nějaké množiny lexémů, kterou nazýváme symbol. V závislosti na syntaxi jazyka může tato množina být pouze jednoprvková. Je třeba mít rovněž na paměti, že lexémy mohou být také bílé znaky (jako jsou tabulátory, konce řádků atp.) a tyto bílé znaky mohou mít speciální význam v daném programovacím jazyce. Bílé znaky však mohou být ignorovány, co se výstupu lexikální analýzy týče, a tím pádem nejsou přímo zahrnuty do lexikální analýzy. Na druhou stranu není výhodné bílé znaky zcela ignorovat, jelikož bez nich lze jen těžko určit přesnou pozici lexému (symbolu) v souboru se zdrojovým kódem. Pozice ve zdrojovém textu zaznamenané pro každý lexém lze poté využít například pro hlášení chyb při analýze—k popisu každé chyby lze připojit její pozici. Jiným případem ignorovaných znaků jsou také znaky uzavřené v symbolech komentáře zdrojového kódu.

Pokud bychom provedli lexikální analýzu kódu ve výpise 1, dostali bychom lexémy a symboly zaznamenané v tabulce 1.

```
1  string mujString = "string" + 5.toString();
```

Výpis 1: Ukázkový kód na vstupu lexikální analýzy

Lexém	Symbol česky	Symbol anglicky
`string`	klíčové slovo	keyword
`mujString`	identifikátor	identifier
`=`	operátor	operator
`"string"`	literál (řetězec)	literal (string)
`5`	číslo	number
`.`	operátor	operator
`toString`	identifikátor	identifier
`;`	konec příkazu	end of statement (EOF)

Tabulka 1: Výstup lexikální analýzy výpisu kódu 1

2.2.2 Vyhodnocení lexémů

Lexém je pouze shluk znaků jehož druh je znám. Aby bylo možno zkonstruovat symbol, je nutno postupně projít znaky daného lexému a vyprodukovat hodnotu atributu. Hodnota atributu společně s druhem lexému (číslo, textový řetězec apod.) pak určuje symbol, který může být předán syntaktickému analyzátoru. *"Symboly, které zahrnují celou třídu lexikálních jednotek (identifikátor, číslo, řetězec) jsou reprezentovány obvykle jako dvojice ⟨druh symbolu,*

hodnota⟩", přičemž druhá část dvojice může být pro některé symboly prázdná."[Beneš(1999), s. 7]. Kupříkladu symbol pro násobení "*"není třeba dále specifikovat hodnotu atributu, jelikož samotný fakt, že se jedná o symbol násobení poskytuje dostatek informací o tomto symbolu. Dalšími příklady takového symbolu mohou být závorky "(", ")", "[", "]", nebo operátory "+", "-", ">", "<", ">=", "<="apod.

Lexikální analyzátor obecně nijak nezpracovává kombinace symbolů a tento úkol je řešen až ve fázi syntaktické analýzy. Například symbol závorek "("a ")"je sice rozeznán lexikálním analyzátorem jako symbol, avšak již není dále zjišťováno, zda-li pro rozpoznanou závorku existuje její protějšek.

Budeme-li mít kód 2, po jeho vyhodnocení v rámci lexikální analýzy dostaneme dvojice ilustrované tabulkou 2 (hodnoty "l_par"a "r_par"představují levou respektive pravou kulatou závorku).

```
1  vysledek = (mnozstvi + zbytek) * 5,3;
```

Výpis 2: Ukázkový kód pro vyhodnocení lexémů

Symbol	
Druh lexému	Hodnota atributu
identifier	vysledek
equals	–
l_par	–
identifier	mnozstvi
plus	–
identifier	zbytek
r_par	–
times	–
number	5,3
semicolon	–

Tabulka 2: Vyhodnocení lexémů z kódu 2

2.2.3 Tabulka symbolů

Při lexikální a syntaktické analýze bývá často využito mechanizmu *tabulky symbolů*. Tabulka symbolů obsahuje identifikátory použité ve zdrojovém kódu a informace o těchto identifikátorech, čímž vzniká mechanizmus pro rychlý přístup k těmto identifikátorům. V této tabulce může také být uložena pozice lexému, typ proměnné, rozsah platnosti atp. Hodnota atributu bývá prakticky realizována jako ukazatel na příslušnou položku v *tabulce symbolů*. Ve fázi lexikální analýzy se rozpoznávají (některé) identifikátory a ty jsou následně zapsány do tabulky. Identifikátory které nelze rozpoznat během lexikální analýzy, jsou do tabulky zapsány v dalších fázích analýzy kódu (syntaktická a sémantická analýza) [Beneš(1999)].

2.2.4 Vzory symbolů

Symboly (tedy množiny lexémů) lze zadefinovat několika způsoby:

- regulární výrazy,
- slovní popis,
- regulární nebo lineární gramatika a
- graf přechodů konečného automatu.

Tuto definici nazýváme *vzor*. Pokud bychom užili regulárních výrazů pro definici vzoru a rozšířili tak tabulku 1, dostaneme tabulku 3 (vzory uvedené v této tabulce nejsou z důvodu úspory místa úplné).

Lexém	Symbol česky	Symbol anglicky	Vzor
`string`	klíčové slovo	keyword	(string|int|char|is|if|for|class|namespace)+
`mujString`	identifikátor	identifier	[a-zA-Z0-9]+
`=`	operátor	operator	(=|\< |\> |++)+
`"string"`	literál (řetězec)	literal (string)	".*"
`5`	číslo	number	[0-9]+
`.`	interpunkce	punctuation	[,|\.|,(,)]+
`toString`	identifikátor	identifier	[a-zA-Z0-9]+
`;`	konec příkazu	end of statement (EOF)	;

Tabulka 3: Výstup lexikální analýzy rozšířený o vzory symbolů

2.2.5 Konečné automaty

Jelikož princip konečného automatu je důležitý pro pochopení především lexikální analýzy, je třeba si nadefinovat a popsat mechanizmus konečných automatů.

Konečný automat lze chápat jako systém, který nabývá konečný počet stavů. Momentální stav automatu se může měnit v závislosti na okolních podnětech. V závislosti na tom, o jaký podmět se jedná, a na tom, v jakém stavu se automat právě nachází, se může aktuální stav automatu změnit. Automaty jsou nazývány konečnými, protože počet stavů kterých mohou tyto automaty nabývat je omezen—jedná se tedy o *konečnou* množinu stavů. Konečný automat lze zadat buď *grafem* automatu nebo *tabulkou* přechodů. V praktické implementaci konečných automatů je často využíváno právě tabulky přechodů konečného automatu. Mechanizmu konečného automatu je využíváno právě v lexikální analýze, jelikož symboly rozpoznávané lexikálním analyzátorem jsou prvky regulárního jazyka a konečné automaty nejsou nic jiného než právě rozpoznávače regulárních jazyků. Definice 2.1 formálně definuje konečný automat [Jančar(2007)].

Definice 2.1 *Konečný automat (zkr. KA) je dán uspořádanou pěticí* $A = (Q, \Sigma, \delta, q_0, F)$*, kde*

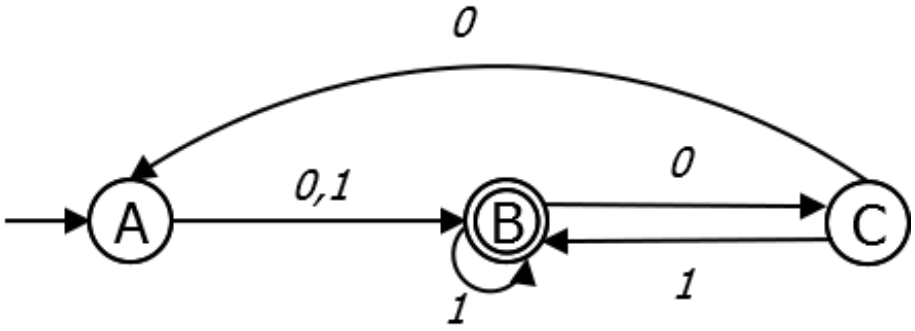
- Q *je konečná neprázdná množina* ***stavů****,*
- Σ *je konečná neprázdná množina zvaná vstupní* ***abeceda****,*
- $\delta : Q \times \Sigma \to Q$ *je* ***přechodová funkce****,*
- $q_0 \in Q$ *je* ***počáteční stav*** *a*
- $F \subseteq Q$ *je množina* ***přijímacích*** *(koncových)* ***stavů****.*

Jak již bylo řečeno, jedna z možností jak může automat být zadán je graf. Graf automatu je orientovaný ohodnocený graf, v němž:

- vrcholy jsou stavy automatu, tedy prvky z množiny Q v definici 2.1,
- počáteční stav q_0 je vyznačen šipkou směřující směrem do vrcholu grafu,
- koncové stavy pak mají uvnitř ještě jeden "vrchol"(kružnici),
- hrana jdoucí ze stavu q_n do stavu q_m nese všechny symboly abecedy, které převádějí automat ze stavu q_n do stavu q_m.

V případě, že automat setrvává daným symbolem v daném stavu, vedeme šipku (smyčku) z tohoto stavu zpět do stejného stavu.

Příklad konečného automatu je na obrázku 2.



Obrázek 2: Příklad konečného automatu

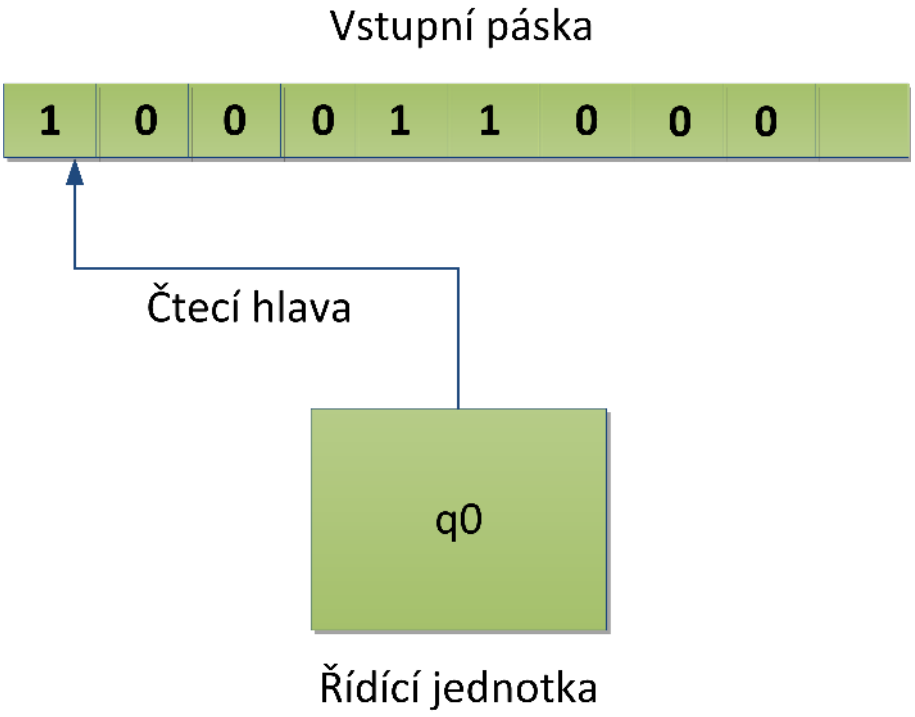
Formálně řečeno, graf na obrázku 2 představuje automat $A = (Q, \Sigma, \delta, q_0, F)$, kde $Q = \{A, B, C\}$, $\Sigma = \{0, 1\}$, $q_0 = \{A\}$, $F = \{B\}$ a přechodová funkce je definována jako $\delta(A, 0) = \delta(A, 1) \to 2$, $\delta(B, 0) \to C$, $\delta(B, 1) \to B$, $\delta(C, 0) \to A$, $\delta(C, 1) \to B$. Zápis $\delta(B, 0) \to C$ znamená, že nachází-li se automat ve stavu B, pak symbol 0 převádí automat do stavu C. Tabulka 4 je přechodovou tabulkou automatu na obrázku 2. Šipka jdoucí směrem do názvu stavu ($\to$) značí výchozí stav, naopak šipka jdoucí směrem ze stavu ($\leftarrow$) označuje stav přijímací.

	Vstup	
Akt. stav	0	1
→ A	B	B
← B	C	B
C	A	B

Tabulka 4: Přechodová tabulka pro konečný automat na obrázku 2

Princip činnosti konečného automatu Schématické znázornění konečného automatu představuje obrázek 3. Vidíme zde vstupní pásku, na níž je zapsáno vstupní slovo. Tato páska je pouze pro čtení. Čtecí hlava se pohybuje zleva doprava po vstupní pásce a postupně čte z pásky symboly na ní zapsané. Řídící jednotka pak představuje aktuální stav, ve kterém se automat nachází.

Na začátku se čtecí hlava nachází na prvním poli vstupní pásky a automat je ve výchozím stavu. Po každém posunu pásky a přečtení symbolu je automat v konfiguraci, která je dána aktuálním stavem řídící jednotky a dosud nepřečtenými symboly. Pokud automat přečte všechny vstupní symboly a nachází se v jednom z přijímacích stavů, slovo zapsané na pásce je *přijímáno* tímto automatem.



Obrázek 3: Schématické znázornění konečného automatu

Jazyk rozpoznávaný automatem

Definice 2.2 *Je-li dán konečný automat* $A = (Q, \Sigma, \delta, q_0, F)$*, pak zápis* $q_0 \xrightarrow{w} q_1$ *značí, že automat přejde ze stavu* q_0 *do stavu* q_1 *přečtením slova* w*.*

Nyní si můžeme nadefinovat, co je to *jazyk* přijímaný automatem.

Definice 2.3 *Je dán konečný automat* $A = (Q, \Sigma, \delta, q_0, F)$*. Slovo* $w \in \Sigma^*$ *je přijímáno automatem A, jestliže* $q_0 \xrightarrow{w} F$*.*

***Jazykem** přijímaným nebo-li rozpoznávaným automatem A je jazyk L(A) takový, že: L(A)=*$\{w \in \Sigma^* \mid q_0 \xrightarrow{w} F\}$.

Konečné automaty také existují v nedeterministické verzi, která ve své podstatě jen speciální verzí deterministického automatu a jsou tedy mezi sebou převoditelné. Další důležité operace z hlediska lexikální analýzy, které lze s automaty provádět jsou minimalizace a sjednocení. Veškeré tyto principy a algoritmy jsou popsané například v [Jančar(2007)] a zde se jejich popisem nebudu zabývat.

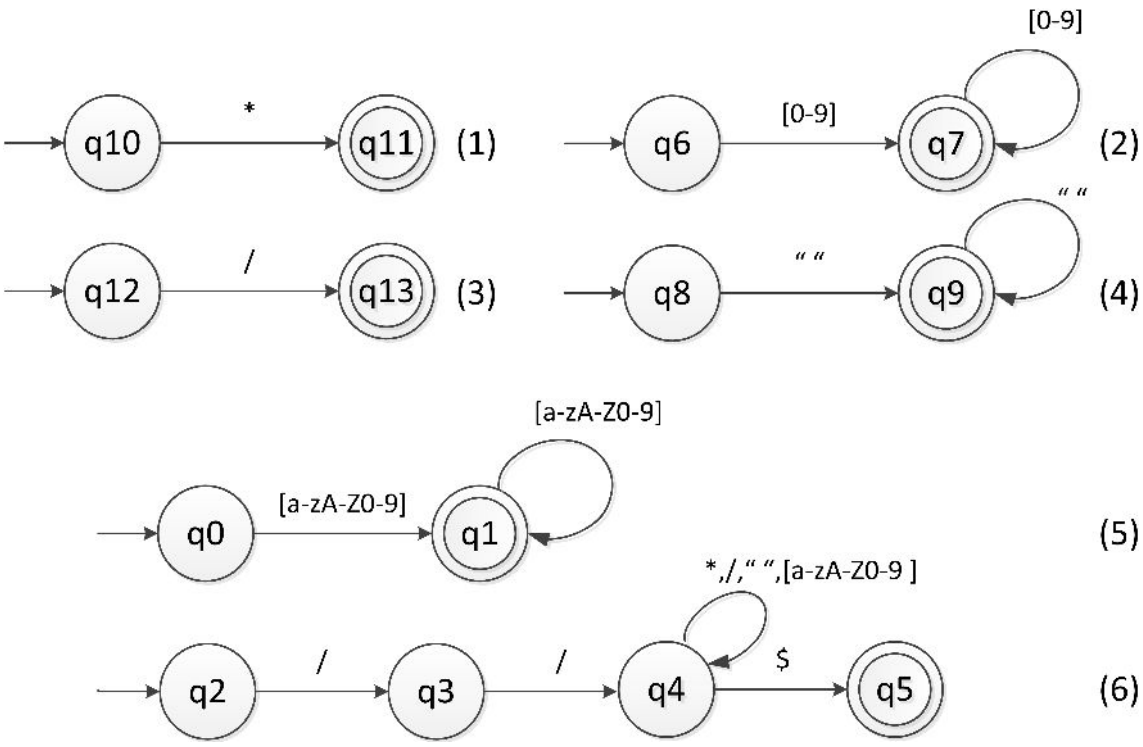
2.2.6 Lexikální analyzátor jako konečný automat

Lexikální analyzátor je ve své podstatě konečný deterministický automat, který vznikne sjednocením menších nedeterministických automatů rozpoznávajících symboly daného jazyka. Tímto spojením vznikne nedeterministický automat, který lze převést na automat deterministický užitím algoritmu převodu nedeterministického na deterministický automat [Beneš(1999)]. Výsledný automat pak stačí implementovat v jazyce, ve kterém je implementován lexikální analyzátor.

Mějme jazyk obsahující následující symboly:

1. operátor pro násobení "*",
2. čísla skládající se z posloupností číslic 0-9,
3. operátor pro dělení "/",
4. mezery " ",
5. identifikátory skládající se z kombinace celých čísel a písmen a
6. komentáře uvozené dvěma lomítky //.

Pro každý symbol vytvoříme automat, který jej přijímá. Grafy automatů přijímající výše definované symboly jsou znázorněny na obrázku 4 (čísla vedle grafů odpovídají číslům v seznamu symbolů výše). Znak "$"představuje konec řádku nebo souboru (angl. *EOF*). Pokud tyto jednotlivé automaty sjednotíme, výsledný automat převedeme na deterministický a ten následně minimalizujeme, dostáváme automat, jehož graf je znázorněn na obrázku 4. Tento deterministický automat se dostane do přijímacího stavu vždy, když ze vstupu bude čteno slovo, jenž je validním symbolem ve výše definovaném jazyce. Přímou implementací tohoto automatu vznikne lexikální analyzátor pro výše definovaný jazyk.



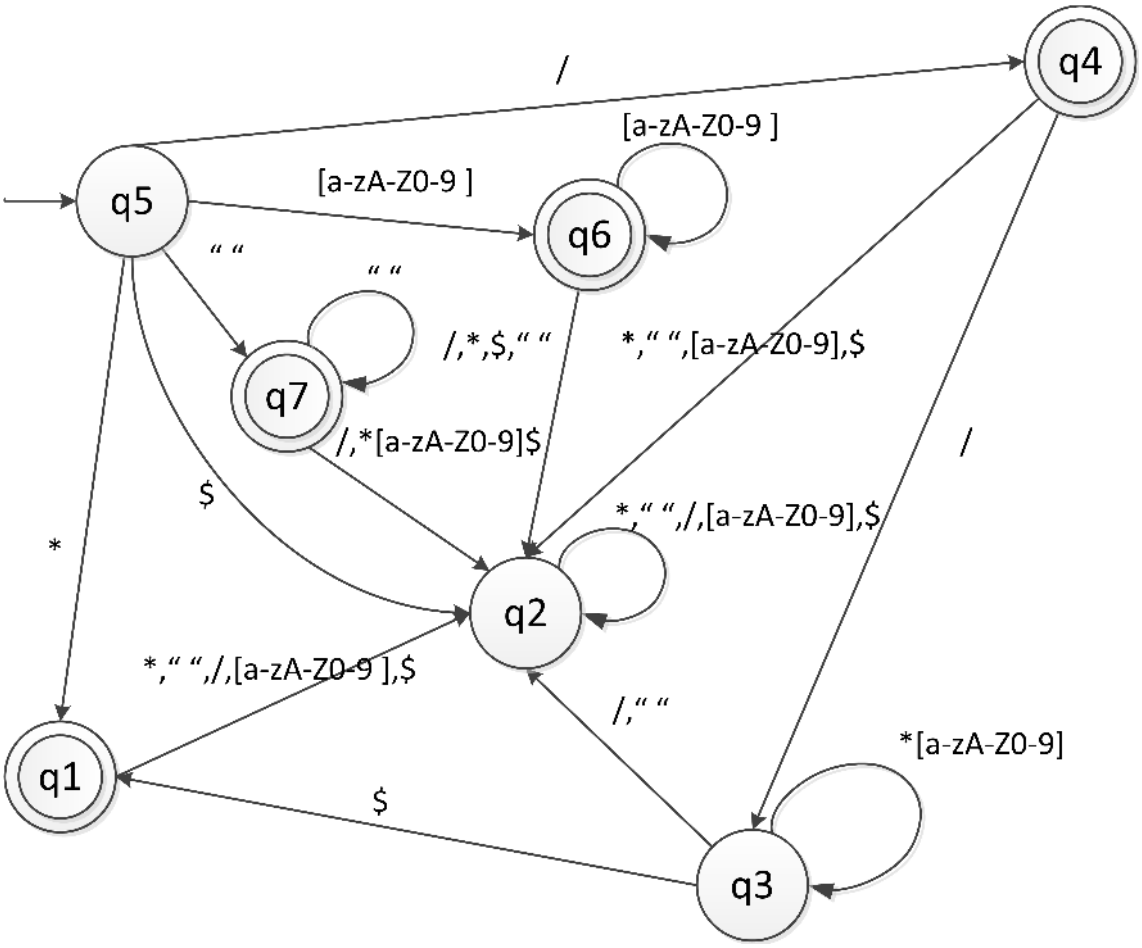
Obrázek 4: Automaty pro rozpoznání jednotlivých symbolů

2.2.7 Regulární výrazy

Regulární výrazy představují mechanizmus umožňující v dané sekvenci znaků vyhledat námi požadovanou sekvenci znaků, která je zadána vzorem popisující vyhledávanou množinu znaků. Tento vzor definující vyhledávaný řetězec znaků se nazývá *regulární výraz*. Pro pochopení regulárních výrazů je třeba si určit jejich syntaxi (způsob zápisu) a sémantiku. Následující definice vymezuje jakési teoretické regulární výrazy známé z oblasti teoretické informatiky, avšak jak je ilustrováno v sekci 3.1, tyto výrazy mají vztah k praktickým aplikacím.

Definice 2.4 *Regulárními výrazy nad abecedou Σ rozumíme množinu **RV**(Σ) slov v abecedě $\Sigma \cup \{\emptyset, \varepsilon, +, \cdot, ^*, (,)\}$, která splňuje tyto podmínky:*

- $\emptyset, \varepsilon, +, \cdot, ^*, (,) \notin \Sigma$,
- *symboly $\emptyset, \varepsilon$ a symbol **a** pro každé písmeno $a \in \Sigma$ jsou prvky **RV**(Σ).*
- *Jestliže $\alpha, \beta \in RV(\Sigma)$, pak také $(\alpha + \beta) \in RV(\Sigma)$, $(\alpha \cdot \beta) \in RV(\Sigma)$ a $(\alpha^*) \in RV(\Sigma)$.*
- *$RV(\Sigma)$ neobsahuje žádné další řetězce, tedy do $RV(\Sigma)$ patří právě ty řetězce (výrazy), které jsou konstruovány z $\emptyset, \varepsilon$ a písmen abecedy Σ výše uvedenými pravidly.*



Obrázek 5: Automaty simulující činnost lexikálního analyzátoru

Definice 2.5 *Regulární výraz α* ***reprezentuje jazyk****, který označujeme $[\alpha]$. Tento jazyk je dán následujícími pravidly:*

- $[\emptyset] = \emptyset$, $[\varepsilon] = \{\varepsilon\}$,$[\alpha] = \{\alpha\}$,
- $(\alpha + \beta) = [\alpha] \cup [\beta]$, $[(\alpha \cdot \beta)] = [\alpha] \cdot [\beta]$, $[(\alpha^*)] = [\alpha]^*$.

Bude-li dána abeceda $\Sigma = \{num, idf\}$, pak platným regulárním výrazem v této abecedě je např. výraz:$(((num \cdot idf) \cdot idf) + (idf + num))^*$.

Důležitým faktem je, že *ke každému regulárnímu výrazu α lze sestrojit konečný automat přijímající jazyk $[\alpha]$*. Důkaz tohoto tvrzení lze najít v [Jančar(2007)], kde lze rovněž nalézt více o regulárních výrazech.

3 Syntaktická analýza

3.1 Bezkontextové gramatiky

Jelikož mnou implementované řešení zvýrazňování syntaxe a doplňování slov stojí z velké části na principech syntaktické analýzy, bude v této sekci podán základní přehled o této problematice.

Bezkontextové gramatiky představují formální techniku, umožňující definici syntaxe daného programovacího jazyka nebo-li definici struktury tohoto jazyka. Pomocí gramatik můžeme výhodně definovat často používané techniky v cílovém programovacím jazyce. *"Jde o alternativu několika možností, opakování stejného jazykového výrazu a hlavně vnořování celých rozvětvených struktur mezi sebou"* [Habiballa(2005), str. 2]. Formální vyjádření bezkontextové gramatiky představuje definice 3.1 [Jančar(2007)].

Definice 3.1 *Bezkontextová gramatika je definována jako uspořádaná čtveřice* $G = (\Pi, \Sigma, S, P)$, *kde*

- Π *je konečná množina* ***symbolů***,
- Σ *je konečná množina* ***terminálních symbolů***,
- $S \in \Pi$ *je* ***počáteční neterminál*** *a*
- P *je konečná množina přepisovacích* ***pravidel*** *typu* $A \rightarrow \beta$, *kde*
  - A *je neterminál,* $A \in \pi$
  - β *je řetězec složený z terminálů a neterminálů, tedy* $\beta \in (\Pi \cup \Sigma)^*$.

Backusova-Naurova forma Abychom mohli s gramatikami pohodlně pracovat, je třeba si zavést nějaká pravidla, říkající jakým způsobem jsou definice gramatik zapsány a jakou sémantiku tento zápis má. V praxi se velmi často používá tzv. *Backusova-Naurova forma* (BNF) zápisu gramatik (popř. varianty této formy). Stejně jako u bezkontextových gramatik, i zde se pracuje s terminály, neterminály a přepisovacími pravidly.

Backusova-Naurova forma je definována následovně:

- neterminály jsou uzavřené v ostrých závorkách <, >,
- přepis neterminálu na terminály a neterminály (oddělení levé strany produkce od pravé) je označen symbolem ::=,
- pravé strany přepisovacích pravidel v rámci jedné levé strany jsou odděleny symbolem svislice |, který slouží pro oddělení možných výběrů pravých stran v rámci jedné levé strany,
- terminály se uzavírají do uvozovek a
- v rámci jedné pravé strany pravidla se terminály a neterminály oddělují mezerou.

Budeme-li mít například pravidlo gramatiky definované v BNF následovně:
<EXPR> ::= "num" | "num" "+" <EXPR>,
pak *<EXPR>* označuje neterminál a zároveň levou stranu pravidla, protože je nalevo od symbolu ::=. Pravidlo říká, že neterminál *EXPR* lze přepsat buď na terminál *num* nebo (symbol svislice |) na kombinaci terminálů a neterminálu *num + EXPR*.

Rozšířená Backusova-Naurova forma Při praktické implementaci syntaktických analyzátorů často opakovaně využíváme podobných konstrukcí a vzniká tak požadavek pohodlnějšího zápisu takovýchto konstrukcí. Z tohoto důvodu vznikla tzv. *Rozšířená Backusova-Naurova forma* (angl. *Extended Backus-Naur Form*) zkráceně *EBNF*. Jak již název napovídá, tato forma rozšiřuje BNF o několik dalších konstrukcí, které usnadňují zápis gramatik v generátorech syntaktických analyzátorů apod. Tyto konstrukce jsou syntakticky i sémanticky podobné konstrukcím užívaným v regulárních výrazech (viz podsekce 2.2.4). Tabulka 5 definuje symboly používané v EBNF a zároveň ji porovnává s BNF a regulárními výrazy [1]. Rozdíly mezi BNF a EBNF dále dokresluje tabulka 6. První řádek

Symbol	Význam	BNF	Regularní výraz
=	definice	::=	—
,	zřetězení	" "(mezera)	·
;	ukončení produkce	EOL (nový řádek)	—
|	volba mezi pravými stranami	|	+
[...]	volitelný výraz	—	+
{...}	opakování výrazu	—	*
"..."	terminál	"..."	—
'...'	terminál	'...'	—
(...)	seskupování	—	(...)

Tabulka 5: Symboly rozšířené BNF

tabulky 6 pouze ukazuje základní rozdíly v symbolice. Za povšimnutí stojí skutečnost, že zatímco v BNF je zřetězení dvou terminálů nebo neterminálů implicitně dáno mezerou mezi nimi, v EBNF je třeba explicitně zřetězit symbolem čárky. Druhý řádek tabulky dále ilustruje použití hranatých závorek pro volitelné výrazy. Třetí řádek ukazuje, jak lze použít symbolů složených závorek pro výrazy, které se mohou opakovat.

[1] Označení "výraz"v tabulce představuje jednotlivé pravé strany pravidel, tedy neterminály a zřetězení terminálů a neterminálů.

	BNF	EBNF
1	<BIT> ::= ″0″|″1″|″0″ ″1″	BIT = ″0″|″1″|″0″,″1″;
2	<BIT> ::= ″0″|″1″|″″	BIT = [″0″|″1″];
3	<BITS> ::= <BIT> | <BIT><BITS>	BIT = {″0″|″1″} | ″″;
	<BIT> ::= ″0″|″1″|″″	

Tabulka 6: Porovnání BNF a EBNF

Derivace v gramatice Mějme gramatiku jejíž pravidla jsou zadána v BNF následovně:

$$< ZEROOP > ::=< ZEROOP > \text{ ″+″ } < ZEROOP > \quad (1)$$
$$< ZEROOP > ::=< ZEROOP > \text{ ″−″ } < ZEROOP > \quad (2)$$
$$< ZEROOP > ::=< ZEROOP > \quad (3)$$
$$< ZEROOP > ::= \text{ ″0″} \quad (4)$$

Daná gramatika generuje například výraz $0 - 0 + 0$. Postup odvození tohoto výrazu je na obrázku 6. Postupnou aplikací pravidel 2, 4, 1, 4, 4 jsme dostali odvození nebo-li

<ZEROOP> $\overset{(2)}{\Longrightarrow}$ *<ZEROOP> ″-″<ZEROOP>* $\overset{(4)}{\Longrightarrow}$ *″0″ ″-″<ZEROOP>* $\overset{(1)}{\Longrightarrow}$
″0″ ″-″<ZEROOP> + <ZEROOP> $\overset{(4)}{\Longrightarrow}$ *″0″ ″-″ ″0″+ <ZEROOP>* $\overset{(4)}{\Longrightarrow}$
″0″ ″-″ ″0″+ ″0″

Obrázek 6: Levá derivace výrazu $0 - 0 + 0$

derivaci daného výrazu. V každém kroku jsme vždy nahrazovali nejlevější neterminál—takovéto odvození se nazývá *levá derivace*. Při bližším pohledu na pravidla gramatiky si lze všimnout, že stejný výraz $(0-0+0)$ je možné odvodit jinou sekvencí aplikace pravidel. Tato derivace je znázorněna na obrázku 7. Je vidět, že aplikací pravidel 1, 2, 4, 4, 4 jsme

<ZEROOP> $\overset{(1)}{\Longrightarrow}$ *<ZEROOP> ″+″<ZEROOP>* $\overset{(2)}{\Longrightarrow}$
<ZEROOP> ″-″<ZEROOP> ″+″<ZEROOP> $\overset{(4)}{\Longrightarrow}$
″0″ ″-″<ZEROOP> + <ZEROOP> $\overset{(4)}{\Longrightarrow}$ *″0″ ″-″ ″0″+ <ZEROOP>* $\overset{(4)}{\Longrightarrow}$
″0″ ″-″ ″0″+ ″0″

Obrázek 7: Jiná levá derivace výrazu $0 - 0 + 0$

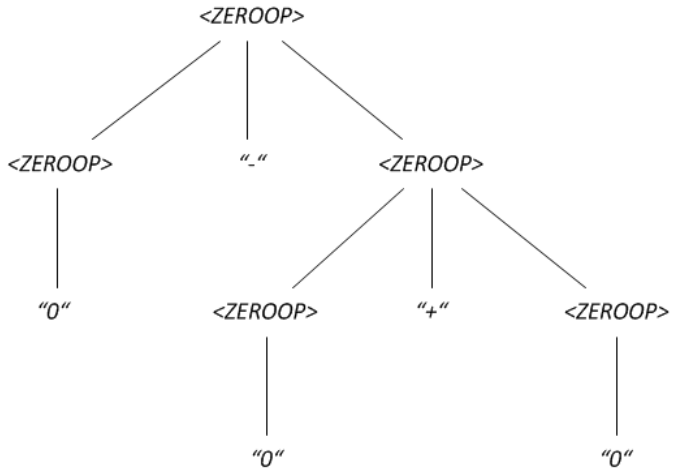
dostali stejný výraz, přičemž se opět jedná o levou derivaci. V gramatice tedy existují *dvě* levé derivace pro jeden a tentýž výraz.

V praxi se často užívá pravých derivací výrazů. Pravá derivace představuje takovou derivaci, kdy v každém jejím kroku je nahrazován *nejpravější* neterminál. Pro výraz $0-0+0$ by pravá derivace mohla být provedena jak ukazuje obrázek 8. Užitím jiné sekvence pravidel (2, 1, 4, 4, 4) jsme opět odvodili stejný výraz. Mělo by být zřejmé, že pro stejný výraz existuje nejméně jedna další pravá derivace různá od té na obrázku 8.

$<ZEROOP> \overset{(2)}{\Longrightarrow} <ZEROOP>\ ″\text{-}″ <ZEROOP> \overset{(1)}{\Longrightarrow}$
$<ZEROOP>\ ″\text{-}″ <ZEROOP>\ ″+″ <ZEROOP> \overset{(4)}{\Longrightarrow}$
$<ZEROOP>\ ″\text{-}″ <ZEROOP> + 0 \overset{(4)}{\Longrightarrow} <ZEROOP>\ ″\text{-}″\ ″0″ + ″0″ \overset{(4)}{\Longrightarrow} ″0″\ ″\text{-}″\ ″0″ + ″0″$

Obrázek 8: Pravá derivace výrazu $0 - 0 + 0$

Derivační stromy a jednoznačnost gramatik Derivační strom je ve své podstatě výsledek syntaktické analýzy zachycující strukturu analyzovaného vstupu, kde listy stromu odpovídají jednotlivým terminálům nebo-li symbolům analyzovaného vstupu. Kořen stromu pak představuje startovací neterminál a uzly z něj vycházející představují neterminály užité k danému odvození. Obrázky 9 a 10 představují derivační stromy pro odvození provedená v předchozím odstavci. Jak je patrné z obrázku 9, výsledná podoba



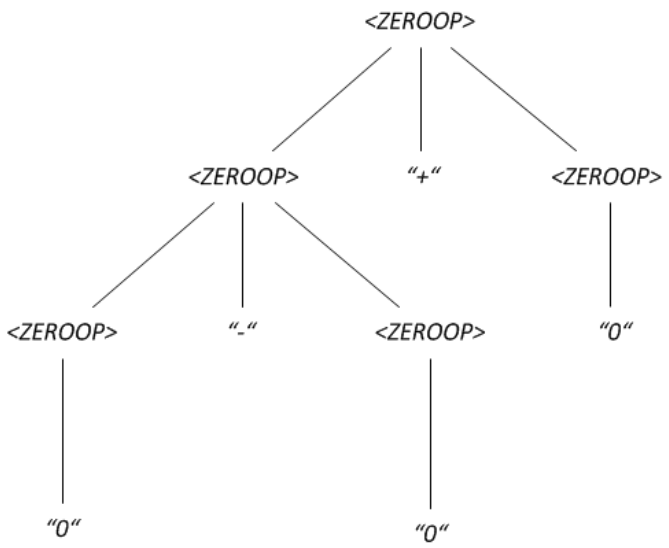
Obrázek 9: Derivační strom pro levou derivaci na obr. 6 a pravou derivaci na obr. 8

derivačního stromu nezávisí na pořadí přepisování neterminálů, jelikož pro dvě různá odvození (obr. 6 a 8) jsme dostali tentýž derivační strom. Druhým důležitým pozorováním je fakt, že pro levé odvození jednoho a téhož výrazu lze vytvořit dva různé derivační stromy.

Existují-li pro levé odvození stejného výrazu ve stejné gramatice dva různé derivační stromy, pak takovouto gramatiku označujeme jako *nejednoznačnou*.

Definice 3.2 *Bezkontextová gramatika G je jednoznačná právě tehdy, když každé slovo generované gramatikou L(G) má právě jeden derivační strom. V opačném případě je G nejednoznačná.*

(Ne)jednoznačnost gramatik je velmi důležitou vlastností z hlediska syntaktické analýzy neboť komplikuje, či dokonce znemožňuje automatickou tvorbu syntaktických analyzátorů. Tato problematika je dále rozebírána v sekcích 3.3 a 4.



Obrázek 10: Derivační strom pro levou derivaci na obr. 7

3.2 Syntaktická analýza zdola nahoru

Syntaktická analýza zdola nahoru identifikuje a začíná zpracováváním základních jednotek (terminálů) a z nich poté postupně odvozuje vyšší úrovně (terminály a neterminály) až se dostane na úroveň nejvyšší (neterminály). Název zdola nahoru vychází z konceptu derivačního stromu (viz podsekce 3.1), kdy analýza začíná v nejnižších úrovních stromu (listech—terminálech) a postupně se propracovává až ke kořenu stromu (startovacímu neterminálu). Toto zpracování začíná na nejlevějším listu derivačního stromu a pokračuje směrem nahoru a doprava. Jedná se o opačnou metodu k metodě syntaktické analýzy *shora nahoru*, o které lze více nalézt například v [Beneš(1999)].

Pokud má jazyk více pravidel, které umožňují začátek ze stejné nejlevější derivace, ale jejich výsledek se liší, potom gramatika takového jazyka může být analyzována syntaktickým analyzátorem shora nahoru. Takový jazyk již nelze analyzovat pomocí analýzy shora nahoru, aniž by bylo třeba hádat další postup a poté provádět zpětné vyhledávání metodou pokusů a oprav (angl. *backtracking*) [Aho et al.(2006)Aho, Sethi,, Ullman].

Algoritmus na obrázku 11 zjednodušeně ilustruje funkci analýzy zdola nahoru.

V praxi se však tohoto algoritmu využívající zpětného vyhledávání nepoužívá, protože většina konstrukcí daného programovacího jazyka umožňuje deterministickou analýzu bez návratů.

3.2.1 Posuvně-redukční zpracování

Posuvně-redukční zpracování (angl. *shift-reduce parsing*) je jedna z často využívaných metod syntaktické analýzy využívající pro svou činnost bezkontextových gramatik [Aho et al.(2006)Aho, Sethi,, Ullman]. Tato metoda je především používaná v generáto-

let I = vstupni retezec;
repeat
  vyber neprázdný podřetězec β z I kde $X \to \beta$ je produkcí;
  if β = **null then**
    vrat'se zpět;
  end
  jedno β nahraď X v řetězci I;
until $I = S$ *(startovací neterminál)* **or nejsou žádné další možnosti**;

Obrázek 11: Zjednodušený algoritmus syntaktické analýzy zdola nahoru

rech syntaktických analyzátorů. Jeden z důvodů popularity této metody je fakt, že velikost tříd rozpoznávaných touto metodou je větší než v případě jiných známých metod. Dalším důvodem, proč je tato metoda často aplikována v praxi je skutečnost, že její implementací lze dosáhnout dobrého výkonu—analýza vstupního řetězce je tedy poměrně rychlá [Sloane – Verity(2008)Sloane, Verity].

Při posuvně redukčním zpracování se postupně čte a zpracovává vstupní řetězec, aniž by docházelo k posunu zpět. Postupně tak dochází k výstavbě derivačního stromu. V každé fázi zpracování je nashromážděno několik podstromů vzniklých z již zpracované části vstupního textu. Při této činnosti je využíváno dvou typů akcí:

- **posun** (angl. *shift*) — při této akci dochází k posunu o jeden symbol vstupního řetězce; tím vznikne nový derivační podstrom obsahující jediný uzel—symbol, na který se právě posunulo,
- **redukce** (angl. *reduction*) — na právě zpracovaný derivační podstrom je aplikováno pravidlo gramatiky a vzniká tak nový derivační podstrom.

Kroky posunu a redukce jsou kombinovány a aplikovány na již zpracovaný řetězec a to tak dlouho, dokud nejsou všechny derivační podstromy zredukovány na jeden derivační strom, který reprezentuje derivaci vstupního textového řetězce [Aho et al.(2006)Aho, Sethi,, Ullman].

Princip posuvně-redukčního zpracování ilustruje následující příklad:

mějme bezkontextovou gramatiku $G = (\Pi, \Sigma, S, P)$, kde $\Pi = \{S, T\}, \Sigma = \{num, *, +, (,)\}, S = \{S\}$ a množina pravidel P je:

<S> ::= <T> "+"<S> | <T>

<T> ::= num "*"<T> | num | "("<S> ")".

Budeme hledat derivaci výrazu $num * num + num$, který tato gramatika generuje. Značka $\perp$ určuje aktuální pozici ve vstupním řetězci. Značení $\xrightarrow{\text{redukce: } T \to num * T}$ znamená,

že následující krok je redukce za použití pravidla gramatiky $T \to num * T$.

$$\perp num * num + num \xrightarrow{\text{posun}} num\perp * num + num \xrightarrow{\text{posun}} num * \perp num + num \xrightarrow{\text{posun}}$$
$$num * \underbrace{num}\perp + num \xrightarrow{\text{redukce: } T \to num} \underbrace{num * T}\perp + num \xrightarrow{\text{redukce: } T \to num * T}$$
$$T\perp + num \xrightarrow{\text{posun}} T + \perp num \xrightarrow{\text{posun}} T + \underbrace{num}\perp \xrightarrow{\text{redukce: } T \to num} T + \underbrace{T}\perp \xrightarrow{\text{redukce: } T \to S}$$
$$\underbrace{T + S}\perp \xrightarrow{\text{redukce: } S \to T + S} S$$

Postupnou aplikací redukcí a posunů jsme došli k počátečnímu startovacímu symbolu S. Z toho plyne, že výraz $num * num + num$ je platnou konstrukcí v zadané gramatice.

3.3 LR Parsery

Syntaktické analyzátory využívající posuvně-redukčního zpracování se nazývají *LR parsery*:

- L (angl. *left*) protože tyto parsery zpracovávají vstup zleva doprava,
- R (angl. *right*) protože při své činnosti konstruují nejpravější derivaci (v obráceném pořadí).

LR parsery jsou deterministické—ke své činnosti nepoužívají mechanizmu zpětného vyhledávaní či hádání dalšího kroku. Konstrukce LR syntaktických analyzátorů je často prováděna pomocí nástrojů pro generování parserů. Charakteristickým znakem LR parserů je způsob, jakým rozhodují o tom, kdy dojde k redukci, a které pravidlo bude při redukci vybráno.

Za názvem LR často následuje kvantifikátor (např. LR(1)), který říká kolik symbolů od aktuální pozice ve vstupním řetězci parser potřebuje znát, před tím, než se rozhodne o dalším kroku. Tato činnost musí být korigována s lexikálním analyzátorem (viz podsekce 2.2), který musí předem znát minimálně o jeden symbol více než parser [Aho et al.(2006)Aho, Sethi,, Ullman].

Jelikož redukce pracují s tou částí vstupního řetězce, která byla právě zpracována, lze si všimnout, že množina již zpracovaných symbolů je ve své podstatě zásobník. LR parsery proto ke své činnosti používají datovou strukturu zásobník a to následujícím způsobem:

- již zpracované části vstupního řetězce (angl. *left-hand side (LHS))* se ukládají na zásobník—zásobník tedy může obsahovat jak terminály tak neterminály,
- v případě akce **posunu** je symbol vstupního řetězce, přes který se posunuje, vložen na vrchol zásobníku (angl. *push*),
- pokud dojde k **redukci**, jeden nebo více symbolů představujících *pravou* stranu pravidla gramatiky (angl. *handle*) je vyjmut (angl. *pop*) ze zásobníku, a následně je levá strana (neterminál) tohoto pravidla vložena na vrchol zásobníku.

Poznámka 3.1 Handle je pravá strana nějakého pravidla gramatiky, která je na vrcholu zásobníku a pouze na vrcholu zásobníku. Handle musí být takové pravidlo gramatiky, jehož zpětnou aplikací (z pravé strany na levou) lze (aplikací přes další pravidla) dosáhnout startovacího neterminálu. Samotná přítomnost handlu na vrcholu zásobníku však automaticky neznamená, že bude provedena redukce [Sloane – Verity(2008)Sloane, Verity].

Tento postup se aplikuje dokud analyzátor nezahlásí úspěch—na zásobníku je pouze startovací neterminál a všechny symboly na vstupu byly zpracovány—či neúspěch. V případě úspěchu je vstup validním v dané gramatice, v případě neúspěchu je vstup nepřijat a je vygenerována syntaktická chyba.

3.3.1 Tabulky činností a kroků

Většina syntaktických analyzátorů užívá konceptu tabulky *činností* (angl. *action table*) a tabulky *kroků* (angl. *goto table*). Pro každou kombinaci vstupního symbolu a aktuálního stavu analyzátoru obsahuje tabulka akci, která bude provedena. Touto akci může být:

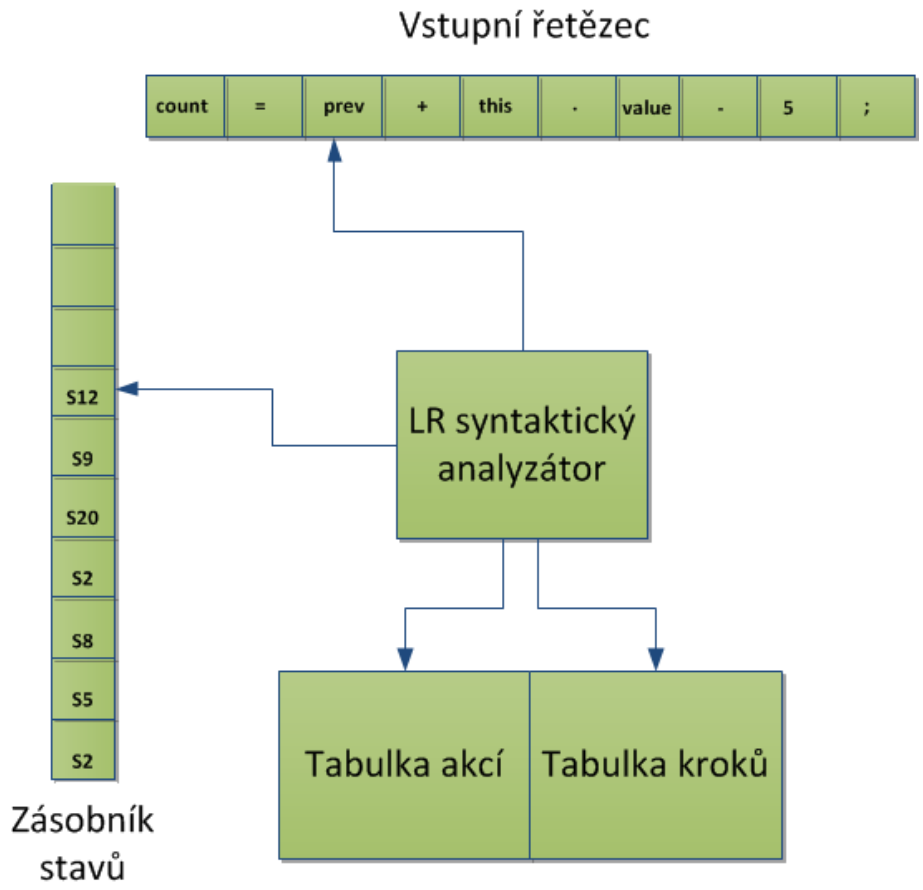
- **posun** do daného stavu,
- **redukce** dle daného pravidla,
- **přijmutí** vstupního řetězce.

Tabulka kroků pak říká, do jakého stavu se analyzátor přesune, nachází-li se v daném stavu (dán obsahem vrcholu zásobníku) v závislosti na tom, na který neterminál byla provedena ona redukce. Obrázek 12 pak představuje schématické znázornění LR parseru.

Činnost LR parseru, který používá tabulky činností a kroků ilustruje následující příklad. Mějme gramatiku $G = (\Pi, \Sigma, S, P)$, kde $\Pi = \{S, T\}, \Sigma = \{0, 1, *, +\}, S = \{S\}$ a množina pravidel P je:
<S>::= <S> ”*”<T> | <S> ”+”<T> | <T>
<T> ::= ”0”| ”1”.
Nejprve provedeme dekompozici pravidel, a každému pravidlu přiřadíme hodnoty a dostáváme obrázek 13.

Tabulka 7 představuje tabulku akcí (sloupce nadepsané ”Akce”) a přechodů (sloupce nadepsané ”Následující stav”). V tabulce akcí pak jsou zadefinovány akce pro každý možný vstupní symbol, přičemž symbol $ zde představuje konec vstupního řetězce. Jednotlivé hodnoty v tabulce akcí pak mají následující význam:

- **p*k*** – **posun** do stavu označeného číslem *k*,
- **r*k*** – **redukce** podle pravidla označeného číslem *k*,
- **akc** – konec analýzy; řetězec je přijat,
- **prázdné pole** – akce pro danou kombinaci není definována; analyzátor přechází do chybového stavu.



Obrázek 12: Schématické znázornění tabulkově řízeného LR parseru

$$S \to S * T \tag{5}$$
$$S \to S + T \tag{6}$$
$$S \to T \tag{7}$$
$$T \to 0 \tag{8}$$
$$T \to 1 \tag{9}$$

Obrázek 13: Dekomponovaná pravidla gramatiky G

Bude-li na vstupu řetězec "1+1", činnost analyzátoru nad tímto vstupem bude následující:

1. na vstupu je symbol "1", zbývající symboly představuje řetězec "+1"; zásobník obsahuje začáteční stav 0:
   (a) v tabulce *akcí* je nalezena akce pro stav 0 a terminál 1—akce $p2$.

Aktuální stav	**Akce**					**Následující stav**	
	*	+	*0*	*1*	*$*	*S*	*T*
0			p1	p2		3	4
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	p5	p6			**akc**		
4	r3	r3	r3	r3	r3		
5			p1	p2	r3		7
6			p1	p2	r3		8
7	r1	r1	r1	r1	r1		
8	p2	p2	p2	p2	p2		

Tabulka 7: Tabulka akcí a přechodů

(b) Stav 2 je uložen na vrchol zásobníku.

(c) Pozice ve vstupním řetězci se posune o jeden symbol doprava.

2. Na vrcholu zásobníku je hodnota 2—zásobník teď obsahuje dvě hodnoty $[0|2]$; vstupní řetězec je tedy "$+1$"a na vstupu je symbol "$+$".

3. V tabulce *akcí* je nalezena akce pro stav 2 (je na vrcholu zásobníku) a terminál "$+$"—akce $r5$.

4. provede se redukce podle pravidla 5:

   (a) Vrchol zásobníku 2 je vybrán, zásobník teď obsahuje pouze stav 0.

   (b) V tabulce *přechodů* (provádíme redukci) je nalezen následující stav pro neterminál T (levá strana pravidla 5) a stav 0 (na vrcholu zásobníku)—stav 4.

   (c) Stav 4 je vložen na vrchol zásobníku.

5. V tabulce *akcí* je nalezena akce pro symbol "+"a stav 4—$r3$.

6. ...

Obdobným způsobem bude analyzátor pokračovat v činnosti do té doby, než dojde k provedení akce **akc**—ze vstupu byl přečten symbol "$"a vstupní řetězec je akceptován. Celou činnost analyzátoru ilustruje tabulka 8. Pokud se podíváme do tabulky 8 na to, jaká pravidla byla použita při redukcích, zjistíme že se jedná o pravidla gramatiky $5, 3, 5, 2$. Během analýzy tedy byla vytvořena následující derivace(číslo v závorce indikuje použité

Aktuální stav	Řetězec na vstupu	Aplikovaní pravidla redukcí	Obsah zásobníku	Následující akce
0	1+1$		[0]	Posun do stavu 2
2	+1$		[0|2]	Redukce podle pravidla 5
4	+1$	5	[0|4]	Redukce podle pravidla 3
3	+1$	5,3	[0|3]	Posun do stavu 6
6	1$	5,3	[0|3|6]	Posun do stavu 2
2	$	5,3	[0|3|6|2]	Redukce podle pravidla 5
8	$	5,3,5	[0|3|6|8]	Redukce podle pravidla 2
3	$	5,3,5,2	[0|3]	Akceptuj

Tabulka 8: Tabulka jednotlivých kroků analyzátoru syntaxe

pravidlo):

$$1 + 1 \leftarrow T \Leftarrow \tag{5}$$
$$1 + T \leftarrow S \Leftarrow \tag{3}$$
$$1 + S \leftarrow T \Leftarrow \tag{5}$$
$$T + S \leftarrow S \tag{2}$$

Pokud odstraníme levé strany a obrátíme směr (jedná se o analýzu zdola nahoru) výše uvedené derivace, dostáváme:

$$S \Rightarrow S + T \Rightarrow S + 1 \Rightarrow T + 1 \Rightarrow 1 + 1,$$

což je pravá derivace věty "$1 + 1$"v gramatice G definované výše.

Protože algoritmus tvorby tabulky akcí a kroků, je poměrně komplexní záležitostí a z hlediska automatické tvorby syntaktických analyzátorů nepříliš důležitý, uvádím zde pouze odkaz na [Aho et al.(2006)Aho, Sethi,, Ullman], kde lze nalézt podrobné vysvětlení této problematiky.

3.3.2 Konflikty vnikající v LR parserech

V závislosti na definici gramatiky mohou při automatické konstrukci syntaktických analyzátorů vznikat konflikty několika typů. Konfliktem zde rozumíme situaci, kdy se analyzátor nachází ve stavu, ve kterém nelze deterministicky určit následující krok. Pokud si představíme konflikty v kontextu tabulky akcí, konflikt je zde indikován skutečností, že jedné kombinaci stavu a vstupního symbolu je přiřazeno více akcí—parser se tedy nemůže rozhodnout, kterou akci provést jako následující a vzniká konflikt.

Tato situace může vznikat v případě, kdy gramatika, ze které je generován analyzátor, je *nejednoznačná*, avšak konflikty mohou vznikat i u gramatik jednoznačných. V zásadě mohou vznikat dva typy konfliktů:

1. **konflikt posun-redukce** (angl. shift-reduce conflict)—vzniká v případě, kdy nějaké pravidlo gramatiky umožňuje provést redukci pro daný symbol a zároveň jiné pravidlo umožňuje provést posun pro stejný symbol. Konflikt posun-redukce často vzniká u rekurzivních pravidel v gramatice, kdy může nastat problém rozhodnout se kdy jedno pravidlo je ukončeno a kdy jiné právě začalo.

2. **konflikt redukce-redukce** (angl. *reduce-reduce conflict*)—vzniká v situaci, kdy v gramatika umožňuje více různým pravidlům redukci pro stejný symbol. V gramatice jsou tedy zadefinována 2 a více pravidel, která vedou na stejnou produkci. Tento typ konfliktu je méně častý a jeho přítomnost většinou znamená chybu v definici gramatiky—typicky nejednoznačnost.

Většinu těchto konfliktů lze vyřešit přepsáním pravidel gramatiky. Generátory parserů často mají přímo vestavěné mechanizmy, které umožňují v případě konfliktu nastavit preferovanou akci—posun nebo redukce. Pokud není nastavena žádná preference, generátor může zvolit akci sám.

Konflikt typu posun-redukce ilustruje gramatika na obrázku 14. V příkladové gra-

$G = (\Pi, \Sigma, S, P)$, kde $\Pi = \{DECL, PARAMS, PARAMSDECL, IDS\}, \Sigma = \{typ, id, (,), ;\}, S = \{DECL\}$ a množina pravidel P je definována:

$$DECL \rightarrow typ\ id(PARAMS) \mid typ\ id() \tag{1}$$
$$PARAMS \rightarrow PARAMSDECL \mid PARAMS; PARAMSDECL \tag{2}$$
$$PARAMSDECL \rightarrow typIDS \tag{3}$$
$$IDS \rightarrow id \mid IDS; id \tag{4}$$

Obrázek 14: Gramatika vedoucí na konflikt posun-redukce

matice na obrázku 14 vnikne konflikt v situaci, kdy na vstupu je symbol středníku ";". Tato gramatika není nejednoznačná—analyzátor pouze potřebuje vědět, jaké jsou další symboly (např. "; typ id"nebo "; id ;"nebo "; id"apod.). V tabulce akcí vzniknou dva záznamy pro stejný stav a symbol ";": jeden říkající analyzátoru že má provést posun do stavu 15 v případě, kdy je na vstupu ";", druhý říkající že pro stejný vstup má být provedena redukce podle pravidla 4. Odstraněním pravidel 3 a 4 a následným přepsáním pravidla 2 jak ukazuje obrázek 15 dojde k odstranění konfliktu. Známým příkladem

$$PARAMS \rightarrow typ\ id \mid PARAMS; typ\ id \mid PARAMS; id \tag{2}$$

Obrázek 15: Přepsání pravidla za účelem odstranění konfliktu posun-redukce

konfliktu posunu-redukce je konflikt "visícího else"(angl. *dangling else problem*), který je přítomen ve všech jazycích syntakticky vycházejících z jazyka ANSI C (například java, C#). Více o tomto konfliktu lze nalézt kupříkladu na [Chang(2009)].

Konflikt redukce-redukce ilustruje gramatika na obrázku 16. Tento konflikt je dán $G = (\Pi, \Sigma, S, P)$, kde $\Pi = \{SLOVA, SLOVO\}, \Sigma = \{slovo;\}, S = \{SLOVA\}$ a množina pravidel P:

$$SLOVA \rightarrow \epsilon \mid SLOVO \quad (1)$$
$$SLOVO \rightarrow \epsilon \mid slovo \quad (2)$$

Obrázek 16: Přepsání pravidla za účelem odstranění konfliktu posun-redukce

nejednoznačností gramatiky. Máme zde dvě pravidla, která umožňují generovat prázdné slovo ε—přímo přes pravidlo 1, nepřímo přes pravidlo 1 a neterminál $SLOVO$. Po vygenerování akční tabulky tedy vzniknou dvě akce redukce pro řetězec $ (konec vstupního řetězce), kdy jedna definuje redukci podle pravidla číslo 1, druhá podle pravidla číslo 2. Odstranění tohoto problému je poměrně přímočaré: stačí odebrat ε ve druhém pravidle a tím odstranit nejednoznačnost v gramatice. Poté již nebudou v gramatice žádné konflikty.

3.3.3 LALR parsery

LALR (Look-Ahead LR) parsery jsou často implementovány v nástrojích pro automatické generování syntaktických analyzátorů. Ve své podstatě jsou velmi podobné LR parserům. Mohou být rovněž implementovány stejným způsobem, za použití tabulek akcí a kroků. Rozdíl oproti LR parserům spočívá ve způsobu, kterým jsou generovány tabulky akcí a kroků. LR parsery vytvářejí akce pro všechny možné redukce z daného stavu a tedy vznikají poměrně rozsáhlé tabulky s mnoha stavy. LALR parsery však vytvářejí kombinace stavů, pokud daný symbol na vstupu při redukci nevytváří konflikt se záznamem v tabulce kroků. Tato skutečnost má za následek tvorbu mnohem menšího počtu stavů a potažmo menších tabulek za cenu menší rozlišovací schopnosti kombinací vstupních symbolů.

LALR parsery v porovnání s LR parsery tedy rozeznávají menší množinu jazyků a jejich použití pro některé jazyky jako například C++ může být poněkud komplikované a náročné, avšak proveditelné [Baxter(2010)]. Většina generátorů parserů využívá právě mechanizmu LALR parserů, jedná se například o genrátory *Yacc*, *GNU Bison* nebo *GOLD*.

4 Generátory lexikálních a syntaktických analyzátorů

Jelikož ruční implementace lexikálních, a především syntaktických, analyzátorů může být pracná a komplikovaná, lze si práci podstatně zjednodušit užitím generátoru. Použití generátoru má však i své nevýhody—programátor má jen omezenou kontrolu nad výslednou podobou parseru, a tím pádem je třeba být opatrnější při definici gramatik, aby nedocházelo ke konfliktům (viz podsekce 3.3.2). Další možnou nevýhodou je výkonnost vygenerovaného analyzátoru. I zde platí pravidlo, že generovaný kód má tendenci být výkonově slabší, než kód psaný ručně, který je šitý na míru vlastnostem analyzovaného jazyka. Tyto generátory typicky pracují s popisem gramatiky cílového jazyka v (E)BNF.

4.1 Irony .NET Language Implementation Kit

Pro generování syntaktického a lexikálního analyzátoru Sparku jsem použil generátor *Irony .NET Language Implementation Kit* (dále jen Irony), který si dává za cíl vybudovat kompletní sadu knihoven a nástrojů pro implementaci jazyků v prostředí Microsoft .NET [Ivantsov(2008)]. V současné době Irony obsahuje moduly pro výstavbu lexikálního a syntaktického analyzátoru.

Na rozdíl od většiny generátoru parserů, které používají meta jazyky pro definici vstupních gramatik, v Irony jsou gramatiky definovány přímo v jazyce C#. Tato definice probíhá v syntaxi podobné EBNF, která je uzpůsobena možnostem a schopnostem jazyka C#. Další charakteristickou vlastností Irony je skutečnost, že nedochází ke generování žádného kódu, protože Irony přímo využívá informací odvozených z definice gramatiky v C# k řízení procesu lexikální a syntaktické analýzy.

4.1.1 Zpracování vstupu

Při zpracování vstupního textu nejprve probíhá lexikální analýza. Výsledné symboly jsou pak předány syntaktickému analyzátoru, jehož výstupem je syntaktický nebo-li derivační strom. Toto je standardní procedura, kterou využívá většina dnešních parserů. V Irony je tato procedura rozšířena o jeden nebo více tzv. *filtr symbolů* (angl. *token filter*). Zpracování vstupu v Irony ilustruje obrázek 17. Jednotlivé moduly (sémantický analyzátor, filtr

Obrázek 17: Schématické znázornění zpracování vstupu v Irony

symbolů a parser) jsou v Irony propojeny pomocí C# *iterátorů* (co jsou iterátory v C# viz [Microsoft(2013a)]). Kód na obrázku 18 ilustruje, jak lze implementovat hlavní metodu filtru symbolů. Více o filtru symbolů pojednává podsekce 4.1.3. Použití iterátorů pro propojení modulů poskytuje více flexibility a volnosti při konstrukci jednotlivých modulů a zároveň zachovává možnost paralelního zpracování. Při zpracování dochází ke skokům

```
public override IEnumerable<Token> BeginFiltering(ParsingContext context,
                                                  IEnumerable<Token> tokens)
{
    foreach (Token token in tokens)
    {
        // Analyzuj příchozí symboly a prověd:
        // symbol můžeme buď předat dále, potlačit,
        // nebo přidat před nebo za něj další symboly.
        yield return token;
        // implicitně jsou tokeny předávány dále
    }
}
```

Obrázek 18: Prázdný filtr symbolů v Irony

mezi moduly, zatímco jednotlivé symboly jsou dále postupovány do dalších fází procesu [Ivantsov(2008)].

4.1.2 Lexikální analyzátor

Třída, která představuje lexikální analyzátor v implementaci Irony se jmenuje `Scanner`. Jedná se o generickou implementaci lexikálního analyzátoru pracujícího na principu popsaném v sekci 2. Třída `Scanner` je zamýšlena pro přímé použití—není třeba z ní dědit.

Uživatelsky definované chování pro daný jazyk je dáno množinou terminálů definovaných v gramatických pravidlech. Irony implicitně obsahuje sadu terminálů, které jsou potomky třídy `Terminal`. Všechny tyto třídy jsou ve své podstatě rozpoznávače symbolů. Mimo jiné zde existují třídy:

- `StringLiteral`—řetězce znaků,
- `NumberLiteral`—čísla,
- `IdentifierTerminal`—identifikátory a
- `CommentTerminal`—komentáře.

Tyto třídy pak mají vlastní podtřídy, které implementují daný terminál pro jazyky jako jsou C#, SQL, Python nebo Visual Basic. Kód na obrázku 19 ukazuje hlavní metodu třídy `Terminal`, kterou lze implementovat a definovat tak vlastní terminály. Objekt `ISourceStream` je rozhraní představující proud znaků na vstupu.

Algoritmus analýzy vstupu Analýza začíná na první pozici ve vstupním řetězci. Bílé znaky jsou přeskakovány až do doby, kdy analyzátor narazí na znak, který není bílý. Poté analyzátor postupně prochází všechny terminály definované ve vstupní gramatice a pro každý zavolá metodu `TryMatch` (viz obrázek 19). Metoda se poté pokouší sestavit terminál, představující validní symbol v dané gramatice čtením a analyzováním vstupu

```
public virtual Token TryMatch(ParsingContext context, ISourceStream source)
{
    // rozpoznej symbol ve ze vstupu (paramtr source)
    // pokud není žádný symbol rozpoznán, vrať "null"
    return null;
}
```

Obrázek 19: Hlavní metoda pro rozpoznávání symbolů

od aktuální pozice dále. V případě že dojde ke shodě na více validních symbolů, je vybrán ten, který obsahuje nejvíce znaků. Tento je pak předán dále ke zpracování a celý proces probíhá znova od koncové pozice právě identifikovaného symbolu až do nalezení konce vstupního řetězce. Analyzátor ve skutečnosti neprochází neustále všechny terminály. Pro rychlé vyhledání možných validních terminálů je využito konceptu hashovací tabulky, která je vytvořena při inicializaci z prefixů, které mohou být explicitně deklarovány pro každý terminál. Užití hashovací tabulky výrazně zrychluje proces lexikální analýzy.

Činnost lexikálního analyzátoru v Irony se v mnoha ohledech liší od jiných dostupných řešení:

1. lexikální analyzátor zcela ignoruje bílé znaky a to i v případě jazyků, ve kterých mají syntaktický význam. Nicméně informace o bílých znacích není úplně ignorována, protože každý symbol obsahuje svou pozici ve vstupním řetězci. Aplikací tohoto principu dochází ke zjednodušení lexikálního analyzátoru. Analýza bílých znaků pak probíhá v oddělené vrstvě filtru symbolů (viz výše).

2. Na rozdíl od jiných řešení, lexikální analyzátor v Irony nepotřebuje definovat stovky číselných konstant pro každý symbol. Symboly jsou přiřazovány prvkům gramatiky přímo ve fázi syntaktické analýzy na základě jejich obsahu. Symbol je tedy definován přímo svým obsahem.

3. Ve většině případů Irony nerozlišuje mezi klíčovými slovy a identifikátory. Všechny alfanumerické znaky jsou považovány za identifikátory a jejich odlišení od klíčových slov probíhá v parseru.

4.1.3 Filtr symbolů

Filtry symbolů jsou speciální procesory, který zachycují proud znaků ze vstupního textu (analyzovaného zdrojového kódu). Ačkoliv označení ”filtr”naznačuje, že znaky jsou z proudu pouze odebírány (filtrovány), není tomu tak a filtr může znaky jak odebírat, tak i přidávat či jinak upravovat. Důvodem pro zavedení tohoto mechanizmu je fakt, že mnoho úloh ve fázi lexikální a syntaktické analýzy je nejlépe zpracováno pomocí nějakého prostředníka, který je umístěn mezi lexikálním analyzátorem a parserem. Jedná se například o následující úlohy:

- zpracování podmíněných klauzulí pro kompilaci,

- kontrola párování různých typů závorek,
- zpracování zakomentovaných bloků kódu,
- zpracování speciálních komentářů dokumentace (javadoc, XML komentáře v C#),
- zpracování bílých znaků pro jazyky, ve kterých mají bílé znaky syntaktický význam (např. Python).

Vlastní implementace filtrů umožňují jejich znovupoužití v rámci zpracování vstupu, kdy každý filtr provádí svoji vlastní úlohu. Takto vniká lépe vrstvená architektura zlepšující znovupoužitelnost kódu. V Irony je filtr symbolů reprezentován třídou `TokenFilter`, která může být snadno poděděna za účelem implementace vlastního filtru. Implicitně jsou v Irony k dispozici dva filtry:

1. `BraceMatchFilter`—umožňuje párovat symboly ohraničující bloky kódu (typicky závorky) a
2. `CodeOutlineFilter`—pro zpracování odsazení kódu.

4.1.4 Syntaktický analyzátor

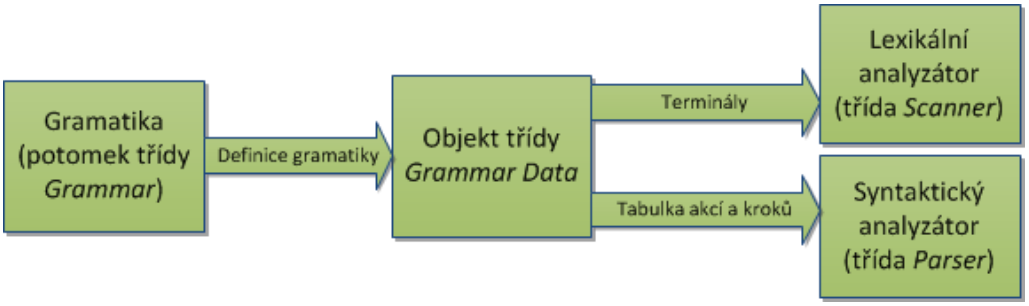
Poté co je vstupní řetězec zpracován lexikálním analyzátorem a případnými filtry, jsou výsledné symboly postoupeny syntaktickému analyzátoru. Syntaktický analyzátor v Irony je typu *LALR(1)* (viz sekce 3.3.3). Důležitými třídami pro fázi syntaktické analýzy jsou třídy `AstNode` a její potomek `Token`. Objekty třídy `AstNode` jsou vytvářeny v případě, kdy parser provádí redukci. Takovýto objekt pak odpovídá neterminálu, pro který byla tato redukce provedena. Na tento objekt (uzel) jsou pak napojeny jeho potomci. Listy (terminály/symboly) v této hierarchii pak představují objekty typu `Token`.

Aby bylo možno určit třídu, ze které bude daný uzel vytvořen, parser prohledává vlastnost (angl. *property*) `NodeType` objektu typu `NonTerminal` specifikovaného pro danou redukci. Při definování gramatiky je možno tuto vlastnost nastavit na patřičnou hodnotu pro každý neterminál. Pokud tato vlastnost natavena není, je použita implicitní hodnota `GenericNode`. Hodnotu `GenericNode` je vhodné použít jako implicitní, jelikož obsahuje vlastnost `ChildNodes`, která pak obsahuje seznam svých potomků.

4.1.5 Gramatiky v Irony

Lexikální a syntaktický analyzátor v Irony jsou představovány třídami, které jsou určeny pro přímé použití bez jakéhokoliv předefinování v rámci konkrétní implementace podpory pro cílový jazyk. Veškeré definice cílového jazyka jsou definovány v nějaké třídě, která dědí ze třídy `Grammar`. Filtry symbolů jsou rovněž definovány v potomcích třídy `Grammar`. Irony ovšem neužívá tyto potomky přímo, ale přes objekt(y) třídy `GrammarData`, které jsou konstruovány při inicializaci, a které reprezentují informace o gramatice cílového jazyka. Objekt třídy `GrammarData` je poté přímo využíván třídami `Scanner` a `Parser`. Lexikální analyzátor využívá terminály definované v objektu `GrammarData` a

syntaktický analyzátor využívá graf stavů (tabulka akcí a kroků, viz podsekce 3.3.1) definovaných v témže objektu třídy `GrammarData`. Zpracování vstupní gramatiky ilustruje obrázek 20.



Obrázek 20: Zpracování vstupní gramatiky

4.2 Irony v praxi

Pro ilustraci definice gramatik v Irony použiji jednoduchou gramatiku umožňující deklarovat celá čísla oddělená čárkou. Pravidla této gramatiky v EBNF jsou následující:

EXPRESSION = {INTEGERS};
INTEGERS = INTEGERDECLARATION | INTEGERDECLARATION , ",", INTEGERS;
INTEGERDECLARATION = "int", IDENTIFIER, {DIGIT};
ALPHABETICCHAR = "A"| "B"| "C"| "D"| "E"| "F"| "G"| "H"| "I"| "J"| "K"| "L"| "M"
| "N"| "O"| "P"| "Q"| "R"| "S"| "T"| "U"| "V"| "W"| "X"| "Y"| "Z";
DIGIT = "0"| "1"| "2"| "3"| "4"| "5"| "6"| "7"| "8"| "9";
IDENTIFIER = {ALPHABETTICCHAR | DIGIT};

Jak již bylo řečeno v podsekci 4.1.5, Irony používá pro definici vstupní gramatiky cílového jazyka potomky třídy `Grammar`. Potomku třídy je navíc možno přiřadit atribut `Language`, který slouží jako metadata pro popis definované gramatiky. Tato metadata je pak možno použít pro rozlišení mezi jednotlivými verzemi gramatik v rámci jedné implementace jazykové podpory. Vlastní definice gramatiky pak typicky následuje v konstruktoru. Tento první krok při definici gramatiky představuje kód na obrázku 21.

Jakmile máme definovánu třídu gramatiky, můžeme přistoupit k definici terminálů, která je na obrázku 22. Na řádku 12 je zadefinován terminál reprezentující celá čísla. První parametr konstruktoru představuje název terminálu, druhý pak typ čísla—v tomto případě se jedná o celá čísla. Řádek 13 pak definuje terminál identifikátoru (viz. obrázek 23). Řádky 14 a 15 deklarují terminály čárky ","a terminál "int"představující klíčové slovo uvozující deklaraci celého čísla.

Na obrázku 23 je kód, sloužící pro zadefinování terminálu identifikátoru, který se skládá z alfanumerických znaků. Dle specifikace Irony, je třeba podědit ze třídy `Terminal` (řádek 28), a dále pak v konstruktoru (řádek 30) zadefinovat jméno terminálu ("identifier"). Na řádku 32 je předefinována metoda `TryMatch` (vysvětlení v sekci 4.1.2). Samotné

```
[Language("Example Language", "v1.0", "An example language")]
public class ExampleLanguageGrammar : Grammar
{
    public ExampleLanguageGrammar()
    {
        // Definice terminálů, neterminálů a pravidel gramatiky
    }
}
```

Obrázek 21: Definice gramatiky v Irony

```
12  NumberLiteral integer = new NumberLiteral("integer", NumberOptions.IntOnly);
13  Terminal identifier = new IdentifierTerminal();
14  KeyTerm comma = new KeyTerm(",", "comma");
15  KeyTerm intKeyWord = new KeyTerm("int", "intKeyWord");
```

Obrázek 22: Definice terminálů v Irony

```
28      public class IdentifierTerminal : Terminal
29      {
30          public IdentifierTerminal() : base("identifier"){}
31
32          public override Token TryMatch(ParsingContext context, ISourceStream source)
33          {
34              var textValue = ReadBody(context, source);
35              if (textValue != null)
36                  return source.CreateToken(OutputTerminal, textValue);
37              return null;
38          }
39
40          public string ReadBody(ParsingContext context, ISourceStream source)
41          {
42              var sb = new StringBuilder();
43              string processed = string.Empty;
44              while (!source.EOF())
45              {
46                  sb.Append(source.PreviewChar);
47                  if (!char.IsLetterOrDigit(source.PreviewChar))
48                  {
49                      return processed;
50                  }
51                  source.PreviewPosition++;
52              }
53              return sb.ToString();
54          }
55      }
```

Obrázek 23: Definice identifikátoru v Irony

rozpoznávání identifikátoru pak probíhá v metodě `ReadBody`. Na řádku 44 je definován cyklus, který končí v případě, že byl nalezen konec vstupního řetězce. Dokud není nalezen konec vstupního řetězce, probíhá (řádky 44–52):

- postupná výstavba textového řetězce ze znaků na vstupu (řádek 46),
- testování, zda-li aktuální znak (`source.PreviewChar`) je alfanumerický (řádek 47)—pokud ne, je vrácena dosud zpracovaná sekvence znaků ze vstupu (řádek 49), jinak se pokračuje dále ve čtení vstupních znaků,
- posunutí aktuální pozice ve vstupním řetězce (řádek 51).

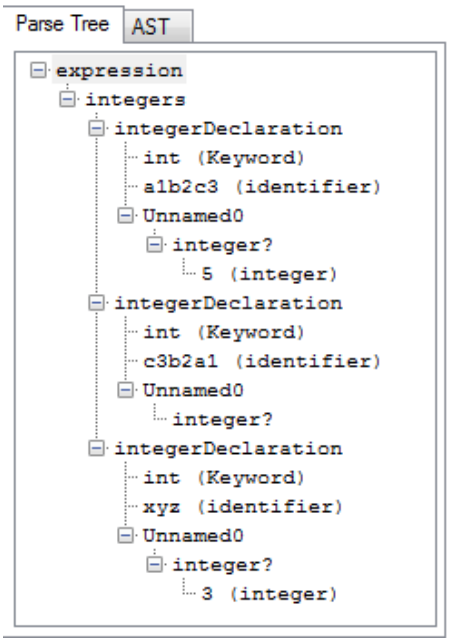
Jestliže jsou všechny znaky na vstupu alfanumerické, je vrácen celý vstupní řetězec (řádek 53) od pozice, na které začala analýza.

Obrázek 24 pak představuje poslední krok—definici neterminálů, pravidel gramatiky a označení klíčových slov. Na řádcích 17 a 18 jsou zadefinovány neterminály pro seznam deklarací celých čísel a deklaraci celého čísla. Řádek 19 pak představuje označení startovacího neterminálu. Na řádku 20 je zadefinováno 0–X opakování produkcí neterminálu `integerDeclaration` oddělených čárkou (terminál `comma`). Tyto produkce opakování jsou pak přiřazeny do neterminálu `integers`. Řádek 21 reprezentuje definici pravidla pro neterminál `integerDeclaration` a to přiřazením této definice do vlastnosti `Rule` zadefinované ve třídě `NonTerminal`. Toto pravidlo se skládá ze tří terminálů(`intKeyWord, identifier, integer`), které se zřetězují symbolem +. Volání metody `Q()` na terminálu `integer` říká, že tento terminál je nepovinný (odpovídá symbolům hranatých závorek v EBNF). Terminál `intKeyWord` je nakonec na řádku 22 označen jako klíčové slovo.

```
17          NonTerminal integers = new NonTerminal("integers");
18          NonTerminal integerDeclaration = new NonTerminal("integerDeclaration");
19          this.Root = integers;
20          integers.Rule = this.MakeStarRule(integers, comma, integerDeclaration);
21          integerDeclaration.Rule = intKeyWord + identifier + integer.Q();
22          this.MarkReservedWords(intKeyWord.Text);
```

Obrázek 24: Definice neterminálů, pravidel a klíčových slov v Irony

Pokud provedeme syntaktickou analýzu například výrazu $int\ a1b2c3\ 5, int\ c3b2a1, int\ xyz\ 3$ za pomoci nástroje *Irony Grammar Explorer*, dostaneme syntaktický strom na obrázku 25. Jednotlivé názvy uzlů odpovídají neterminálům, které byly použity pro vytvoření terminálů v jejich potomcích. Tyto názvy jsou definovány v konstruktoru třídy `NonTerminal`. Názvy symbolů každého terminálu jsou pak uvedeny v závorce za hodnotou terminálu. Veškeré tyto informace jsou programově přístupné jak bude ukázáno v části zabývající se implementací zvýrazňování syntaxe. Obrázek 26 pak ukazuje stavy a akce tabulky akcí a kroků.



Obrázek 25: Syntaktický strom v Irony Grammar Explorer

```
State S1
  Shift items:
    expression' -> expression ·EOF
  Transitions:

State S2
  Reduce items:
    expression -> integers ·
  Transitions:

State S3 (Inadequate)
  Shift items:
    integerDeclaration+ -> integerDeclaration+ ·comma integerDeclaration
  Reduce items:
    integers -> integerDeclaration+ · [EOF]
  Transitions: comma->S7

State S4
  Reduce items:
    integerDeclaration+ -> integerDeclaration ·
  Transitions:

State S5
  Shift items:
    integerDeclaration -> intKeyWord ·identifier Unnamed0
  Transitions: identifier->S8

State S6
  Reduce items:
    expression' -> expression EOF ·
  Transitions:

State S7
  Shift items:
    integerDeclaration+ -> integerDeclaration+ comma ·integerDeclaration
    integerDeclaration -> ·intKeyWord identifier Unnamed0
```

Obrázek 26: Stavy a akce parseru v Irony Grammar Explorer

5 Rozšiřování Visual Studia 2012

5.1 Managed Extensibility Framework

Managed Exntensibility framework (dále jen MEF) je knihovna, umožňující vytvářet malé, rychlé a snadno rozšiřitelné aplikace. Dává k dispozici mechanizmus, jež dovoluje používat rozšíření bez nutnosti provádět jakékoliv konfigurace. Zapouzdření kódu naprogramovaných rozšíření je snadné a zároveň nevznikají závislosti náchylné k porušení. Rozšíření v MEF je možné použít nejen v rámci jedné aplikace, ale také napříč vícero aplikacemi. Tato knihovna je součástí Microsoft .NET framework od verze 4.0.

5.2 Vlastnosti MEF

Při použití MEF není třeba explicitně registrovat dostupné komponenty, protože MEF poskytuje způsob, jak vyhledávat dostupné komponenty pomocí tzv. *kompozice*. Komponenta MEF deklarativně specifikuje jak své závislosti (tzv. *importy*), tak i své dostupné vlastnosti (tzv. *exporty*). Kompoziční mechanizmus MEF je zodpovědný za zahrnutí importů MEF komponenty při jejím vytvoření. Tento přístup řeší problémy při implementaci rozšíření jako jsou:

- nemožnost přidávat nové komponenty bez nutnosti modifikovat zdrojový kód (v případě přímého vložení kódu komponenty do aplikace),
- nutnost explicitně specifikovat dostupné rozšíření (v případě implementace komponenty přes rozhraní),
- nutnost vzájemné komunikace komponenty a prostředí, v němž běží přes pevně definované kanály (také v případě implementace komponenty přes rozhraní),
- problematická použitelnost komponenty v rámci více aplikací.

Protože MEF komponenty deklarativně specifikují své schopnosti, je snadné je nalézt za běhu aplikace. To znamená, že aplikace mohou využívat komponenty, a to aniž by bylo třeba napevno programovat reference nebo používat konfigurační soubory. Aplikace vyhledávají dostupné MEF komponenty za užití metadat. Není tedy nutno vytvářet instance komponent, nebo načítat assembly, nebo specifikovat, kdy a jak se mají rozšíření načítat. Každá komponenta specifikuje svůj export a může také specifikovat importy, což umožňuje rozšiřovat samotné komponenty [Microsoft(2013b)].

5.3 Komponenty, kompozice, export a import

Komponentní část (angl. *component part*) je třída nebo člen třídy, která může buď konzumovat jinou komponentu, nebo být konzumována jinou komponentou. *Kompoziční kontejner* (angl. *composition container*), jenž je poskytován hostující aplikací, je zodpovědný za udržování seznamu exportů a správu importů komponent. Typicky je tento kontejner vlastněn hostující aplikací a je reprezentován třídou *CompositionContainer*.

Jakákoliv funkcionalita může být exportována, pokud je implementována jako veřejná třída nebo veřejný člen třídy. Pro provedení exportu stačí přidat atribut `ExportAttribute` třídě nebo členu třídy, který má být exportován. Tento atribut specifikuje ujednání (angl. *conract*), dle kterého budou provedeny importy (jinou komponentní částí). Provedení exportu ukazuje výpis kódu 3. Atribut `Export` implicitně definuje kontrakt, který je představován typem exportované třídy (zde volání `typeof(ConentTypeDefinition)`). Není tedy třeba explicitně specifikovat exportovaný typ—kód na 4 je ekvivalentní výpisu 3.

```
1  [Export(typeof(ContentTypeDefinition))]
2  class TestContentTypeDefinition : ContentTypeDefinition { }
```

Výpis 3: Provedení exportu funkcionality

```
1  [Export]
2  class TestContentTypeDefinition : ContentTypeDefinition { }
```

Výpis 4: Provedení exportu funkcionality s implicitním typem

Pokud je požadavek načíst MEF export, je třeba znát importovaný kontrakt. Obvykle se jedná o typ tohoto exportu. Poté již stačí přidat atribut importu `Import` proměnné typu exportu. Kód 5 ilustruje provedení importu pro typ `IClassificationTypeRegistryService`.

```
1  [Import]
2  internal IClassificationTypeRegistryService  ClassificationRegistry ;
```

Výpis 5: Provedení importu typu IClassificationTypeRegistryService

5.4 VSIX balíčky

Jednou z možností, jak publikovat a rozmisťovat rozšíření naprogramovaná v MEF je *VSIX balíček* (angl. *VSIX Package*). Jedná se o komprimovaný soubor, který využívá standardu OPC (Open Packaging Conventions). Balíček obsahuje binární a podpůrné soubory společně s xml souborem definujícím druh VSIX balíčku, který je nutný pro správnou instalaci rozšíření přes tzv. manažera rozšíření (angl. *Exntension Manager*) Visual Studia. Rovněž je zde soubor manifestu (angl. *VSIX manifest file*), obsahující instalační informace o daném rozšíření (popř. rozšířeních). Ve své podstatě se jedná o XML soubor se strukturou definovanou výpisem 6.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <PackageManifest Version="2.0">
3    <Metadata>... </Metadata>
4     <InstallationTargets>... </InstallationTargets>
5     <Dependencies>... </Dependencies>
6     <Assets>... </Assets>
7  </PackageManifest>
```

Výpis 6: Struktura VSIX manifestu

Atribut *Version* elementu *PackageManifest* představuje verzi formátu manifestu—pro doplňky rozšiřující Visual Studio 2012 je hodnota verze rovna *2.0*. Element *Metadata* obsahuje data o samotném balíčku. Najdeme zde například jednoznačný identifikátor, jméno, popis, ikony atp. Element *InstallationTarges* pak popisuje, jak má být balíček instalován. Také je zde informace o verzi Visual Studia, pro kterou je daný balíček určen. V elementu *Dependencies* se definují závislosti daného balíčku (např. verze .NET Frameworku); tento element je nepovinný. Element *Assets* obsahuje seznam všech aktiv obsažených v balíčku. Tento element je povinný a nelze bez něj propagovat žádná rozšíření obsažená v balíčku. Manifest může také obsahovat libovolné další, uživatelsky definované elementy. Více o možnostech manifestu VSIX balíčků lze nalézt na [Microsoft(2013c)].

5.5 Rozšiřování Visual Studia 2012

Visual Studio umožňuje použití několika typů rozšíření v závislosti na velikosti a druhu cílového rozšíření. Jedním z nich jsou rozšíření založená na MEF, která umožňují vytvářet nová rozšíření užitím principů popsaných v sekci 5.1. Rozšíření se definují ve shodě s konvencemi MEF a jsou poté exportována jako MEF komponenty. Hostitelská aplikace pak spravuje tyto komponenty tím, že je vyhledává, registruje a aplikuje ve správném kontextu.

Aby bylo vůbec možno vytvářet rozšíření Visual Studio, je třeba nainstalovat vývojový kit Visual Studio SDK příslušné verze—v tomto případě tedy Visual Studio 2012 SDK—který obsahuje veškeré definice tříd, rozhraní, rozšíření a dalších souborů nutných k vytváření rozšíření Visual Studio.

5.6 Rozšiřování Editoru Visual Studia

Visual Studio umožňuje použít MEF rozšíření pro změnu či úpravu vzhledu a chování editoru zdrojových kódů. Uživatelské rozhraní editoru (a potažmo celého Visual Studia) je implementováno ve *Windows Presentation Foundation* (zkr. WPF), což umožňuje užití vlastností a prvků definovaných ve WPF. Mezi základní vlastnosti editoru, které lze rozšířit patří:

- typy obsahu (angl. *Content Types*),
- okraje editoru a posuvníky (angl. *margin* a *scrollbar*),
- značky (angl. *tag*,)
- textové efekty (angl. *adorments*) a
- doplňování slov (angl. *intellisense*).

5.6.1 Rozšiřování typů obsahu

Typy obsahu (angl. *content types*) představují definice typů textu, který je zpracováván editorem Visual Studia. Jedná se například o typy `"text"`, `"code"` nebo `"Csharp"`.

Nový typ obsahu je definován deklarací proměnné typu `ContentTypeDefinition`, které je předáno unikátní jméno. Pro registraci nového typu obsahu do editoru Visual Studia je třeba přidat proměnné představující nový typ následující atributy:

- `NameAttribute`—pro definici unikátního jména typu obsahu,
- `BaseDefinitionAttribute`—jméno typu obsahu, který nový typ obsahu rozšiřuje. Nový typ obsahu může rozšiřovat více již existujících typů.

Export nového typu obsahu rozšiřujícího existující typy `"code"` a `"projection"` znázorňuje výpis kódu 7.

```
1 [Export]
2 [Name("test")]
3 [BaseDefinition("code")]
4 [BaseDefinition("projection")]
5 internal static ContentTypeDefinition TestContentTypeDefinition;
```

Výpis 7: Export nového typu obsahu

Poté, co je zadefinován a exportován nový typ obsahu, je třeba tento typ svázat s koncovkou souborů, na které má být typ obsahu aplikován. Ve Visual Studiu jsou koncovky souborů registrovány deklarací proměnné typu `FileExtensionToContentTypeDefinition`, na kterou se aplikují atributy:

- `Export`,
- `FileExtensionAttribute`—jako parametr se předá jméno přípony,
- `ContentTypeAttribute`—typ obsahu, se kterým má být přípona svázána.

Svázání přípony ".txt"s typem obsahu "test"představuje kód ve výpisu 8

```
1 [Export]
2 [FileExtension(".txt")]
3 [ContentType("test")]
4 internal static FileExtensionToContentTypeDefinition TestFileExtensionDefinition;
```

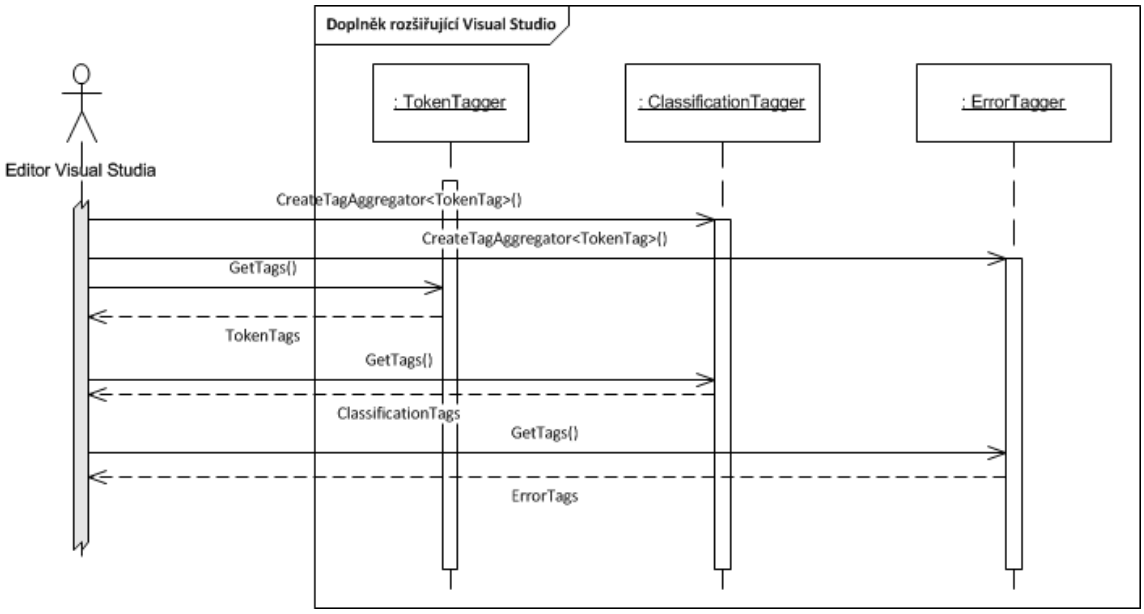
Výpis 8: Svázání typu obsahu s příponou souboru

5.6.2 Zvýrazňování syntaxe a podtrhávání chyb

Zvýrazňování syntaxe a podtrhávání chyb spadá do kategorie *značkování* (angl. *tagging*). V kontextu Visual Studia jsou *značky* (angl. *tagy*) představovány datovými atributy, které jsou asociovány s rozpětími (angl. *span*) textu. Chybové a klasifikační značky (angl. *error tags* a *classification tags*), které se mimo jiné požívají pro zvýrazňování syntaxe, jsou přímo podporovány editorem Visual Studia. Pro vytvoření služby zvýrazňování syntaxe je třeba vytvořit značkovací třídu (angl. *tagger*), která se bude starat o aplikaci klasifikačních značek na rozsahy textu a následně těmto klasifikacím přiřadí vizuální podobu (typicky

barvu). Pro označení chyb je nutno vytvořit značkovač, který chybám v textu přiřadí předdefinované chybové značky. Chybový text je většinou podtržen červenou čarou.

Pro zvýraznění syntaxe a podtržení chyb jsou tedy potřeba dva značkovače. MEF umožňuje zadefinovat společnou službu, která bude poskytovat značky pro symboly ve zdrojovém textu. Poté stačí tyto značky zpracovat klasifikačním a chybovým značkovačem. Tento (zjednodušený) postup je ilustrován sekvenčním diagramem na obrázku 27. Všechny značkovače implementují rozhraní `ITagger`, které definuje metodu



Obrázek 27: Proces značkování ve Visual Studio

`GetTags()`. Rovněž je potřeba objektu poskytovatele značek, který vzniká implementací rozhraní `ITaggerProvider`, jenž představuje MEF komponentní část hostovanou Visual Studiem. Vytvoření poskytovatele značek ilustruje výpis kódu 9

```
1  [Export(typeof(ITaggerProvider))]
2  [ContentType("test")]
3  [TagType(typeof(MyLanguageTokenTag))]
4  internal sealed class MyLanguageTokenTagProvider : ITaggerProvider
5  {
6      public ITagger<T> CreateTagger<T>(ITextBuffer buffer) where T : ITag
7      {
8          return new MyLanguageTokenTagger(buffer) as ITagger<T>;
9      }
10 }
```

Výpis 9: Vytvoření poskytovatele značek

Nejdůležitější metodou definovanou rozhraním `ITagger` je metoda `GetTags()`. Implementace této metody v konkrétním značkovači je pak zodpovědná za získání seznamu objektů definujících rozpětí značek. Rozpětí značek je představováno objekty tříd, které

implementují generické rozhraní `ITagSpan`. Toto rozhraní je následně typováno na třídu implementující rozhraní `ITag`, která představuje vlastní objekt značky (výpis kódu 11). Rozhraní `ITagger` je rovněž generického typu a je typováno na objekt značky. Tento postup je ilustrován výpisy kódu 10 a 11.

```
1 internal sealed class MyLanguageTokenTagger : ITagger<MyLanguageTokenTag>
2 {
3     public IEnumerable<ITagSpan<MyLanguageTokenTag>> GetTags(
          NormalizedSnapshotSpanCollection spans)
4     { }
5 }
```

Výpis 10: Implementace značkovače symbolů

```
1 public class MyLanguageTokenTag : ITag{}
```

Výpis 11: Definice objektu značky

Definice klasifikace pro zvýrazňování syntaxe Klasifikace značek je implementována podobně jako generický značkovač popsaný výše. Navíc je třeba zadefinovat poskytovatele klasifikace (angl. *classifier provider*), jehož cílem je identifikace značek a jejich typu s následným exportem této funkcionality do Visual Studia. Hlavním rozdílem oproti výše definovanému generickému značkovači je to, že metoda `GetTags()` vrací objekty implementující typu `ClassificationTag`. Klasifikační značkovač potřebuje pro svoji činnost naimportovat dvě služby:

- `IBufferTagAggregatorFactoryService`—umožňuje vytvořit objekt, který nese značky z generického značkovače definovaného ve výpisu kódu 10. Ty jsou pak čteny v klasifikátoru.
- `IClassificationTypeRegistryService`—slouží pro svázání klasifikačního typu (symbolu) s prezentačním formátem (barvou).

Kód 12 ukazuje, jak je tento import proveden.

```
1 [Import]
2 internal IBufferTagAggregatorFactoryService aggregatorFactory = null;
3
4 [Import]
5 internal IClassificationTypeRegistryService ClassificationTypeRegistry = null;
```

Výpis 12: Import služeb klasifikátoru

V praxi se pak pro každý klasifikační typ (symbol) provede export tohoto typu jak ilustruje kód 13. Klasifikačnímu typu přiřadíme jeho jméno pomocí parametru atributu `Name`. Toto jméno je pak použito pro specifikaci prezentace typu jak ilustruje kód na obrázku 28.

```
1 [Export(typeof(ClassificationTypeDefinition))]
2 [Name("keyword")]
3 internal static ClassificationTypeDefinition myLanguageKeyWord = null;
```

Výpis 13: Export klasifikačního typu

```
9     [Export(typeof(EditorFormatDefinition))]
10    [ClassificationType(ClassificationTypeNames = "keyword")]
11    [Name("keyword")]
12    [UserVisible(true)]
13    [Order(Before = Priority.Default)]
14    internal sealed class MyLnaguageKeyWord : ClassificationFormatDefinition
15    {
16        public MyLnaguageKeyWord()
17        {
18            this.DisplayName = "MyLanguage Keyword";
19            this.ForegroundColor = Colors.Blue;
20        }
21    }
```

Obrázek 28: Export klasifikačního formátu

Kód na obrázku 28 nejprve definuje vlastní export (řádek 9). Poté je na řádku 10 klasifikační formát svázán s klasifikačním typem. Následně je na řádku 11 přiřazeno klasifikačnímu formátu jméno, které je pak vyhledáno přes objekt služby `IClassificationTypeRegistryService`. Řádek 12 nastavuje viditelnost klasifikačního formátu pro uživatele. Pokud je nastaveno na `true`, je možno měnit definici formátu z rozhraní Visual Studia. Atribut `Order` na řádku 13 slouží pro definici priority klasifikačního formátu. Pokud bude pro daný formát nalezeno více validních formátů, hodnota tohoto atributu určí, který z formátů bude aplikován. V konstruktoru na řádku 16 je pak nastaveno jméno formátu (bude viditelné v menu Visual Studia) a specifikace formátu (řádek 19), která říká, že text bude zobrazen modře.

Implementace podtrhávání chyb Implementace podtrhávání chyb je pak velmi podobná implementaci klasifikátoru. Je třeba zadefinovat poskytovatele značek implementujícího rozhraní `ITaggerProvider`, avšak v tomto případě je nutno zadefinovat export typu `ErrorTag`. Druhý rozdíl spočívá v tom, že zatímco metoda klasifikátoru `GetTags()` vrací seznam objektů `ITagSpan<ClassificationTag>`, metoda značkovače chyb vrací seznam objektů `ITagSpan<ClassificationTag>`. Tyto objekty pak představují rozpětí textu, která mají být označena jako chybová (červené podtrhnutí). Zbytek implementace se nijak neliší od implementace třídy klasifikátoru.

5.6.3 Doplňování slov (Intellisense)

Funkcionalita doplňování slov je implementována pomocí několika provázaných komponent. Nejdůležitější z nich jsou:

- **Intellisemse Source**—zdrojový objekt, který poskytuje obsah jako je například seznam validních slov pro doplnění.
- **Intellisense Controller**—správce životního cyklu sezení (angl. *session*) doplňování slov.

- **Intellisense Session**—objekt sezení reprezentující aktivní doplňování slov.
- **Intellisense Presenter**—představuje prezentační vrstvu obsahu doplňování slov v editoru.
- **Intellisense Broker**—importován z MEF. Řídí interakci mezi ostatními komponentami intellisense.

Proces doplňování slov funguje tak, že rozšíření, které implementuje funkcionalitu doplňování slov, poskytne Visual Studiu objekt správce sezení, který je svázán s oknem editoru. Objekt správce sezení pak naslouchá událostem, které jsou vyvolány v okně editoru a v případě, že je to žádoucí, vytvoří sezení doplňování slov. Objekt sezení pak získá z objektu zdroje validní slova pro doplnění a ty jsou pak zobrazeny.

Pro implementaci funkcionality doplňování slov je nutno vytvořit objekt sezení `CompletionSession`, který bude vznikat jako odezva na úhozy na klávesnici a jiné typy událostí. Tvorbu objektu sezení bude mít na starost třída implementující rozhraní `IOleCommandTarget`. Tuto třídu pojmenujeme `CommandFilter`—filtr příkazů. Pokud `CommandFilter` zachytí příkaz, který spouští sezení doplňování slov, vytvoří nové sezení za pomocí objektu zprostředkovatele. Objekt zprostředkovatele implementuje rozhraní `ICompletionBroker` a je importován z Visual Studia prostřednictvím MEF poté, co je vytváře objekt `CommandFilter`u. Nejdůležitější metodou `CommandFilter`u je metoda `Exec()`, která inicializuje sezení v závislosti na předaných parametrech. Tato metoda mimo jiné definuje parametry:

- `ref Guid pguidCmdGroup`—identifikátor skupiny příkazů, do které spadá příkaz zachycený filtrem příkazů.
- `uint nCmdID`—identifikátor právě zachyceného příkazu.
- `IntPtr pvaIn`—číselná reprezentace znaku právě napsaného v okně editoru (pro příkazy nevkládající žádné znaky je tato hodnota rovna nule).

Jakmile máme vyřešeno vytváření sezení, potřebujeme mechanizmus, jenž poskytne samotná slova pro doplnění, která se zobrazí uživateli. Tuto funkcionalitu obstarává objekt implementující rozhraní `ICompletionSource`, nazývaný zdroj doplnění (angl. *completion source*). Objekt zdroje doplnění je dostupný přes poskytovatele zdroje doplnění (angl. *completion source provider*), který implementuje rozhraní `ICompletionSourceProvider`. Navíc je třeba tento objekt exportovat přes MEF a přiřadit mu požadovaný typ obsahu. Export poskytovatele doplnění ilustruje výpis kódu 14.

```
1  [Export(typeof(ICompletionSourceProvider))]
2  [ContentType("test")]
3  [Name("myLanguageCompletion")]
4  class OokCompletionSourceProvider : ICompletionSourceProvider
5  {
6      public ICompletionSource TryCreateCompletionSource(ITextBuffer textBuffer)
7      {
8          return new MyLanguageCompletionSource(textBuffer);
```

```
 9       }
10   }
```

Výpis 14: Export poskytovatele doplnění

Posledním krokem při implementaci doplňování slov je definice zdroje. V kontextu výpis kódu 14 by se jednalo o třídu `MyLanguageCompletionSource`. Rozhraní `ICompletionSource` implementované touto třídou definuje metodu `AugmentCompletionSession`, které je předán objekt sezení, seznam slov pro doplnění (`IList<CompletionSet>`) a rozpětí textu, na které jsou slova z tohoto seznamu aplikovatelné. V této metodě pak lze získat informace o akci, která si vyžádala vyvolání doplnění a na jejich základě přidat požadovaná slova do seznamu.

6 Microsoft Roslyn

Microsoft Roslyn je nové řešení poskytující API pro lexikální, syntaktickou a sémantickou analýzu kódu pro jazyky C# a Visual Basic. Toto API se skládá ze tří vrstev:

1. API kompilátoru—poskytuje objektovou reprezentaci informací dostupných v každé fázi kompilace, tedy jak syntaktické tak sémantické informace.
2. API služeb—poskytuje přístup k pracovním prostorům (angl. *Workspace*). Umožňuje pracovat s řešeními (angl. *Solution*) a projekty ve Visual Studio bez nutnosti se starat o závislosti nebo provádět jakékoliv konfigurace.
3. API Editoru—poskytují snadný přístup ke službám editoru Visual Studia.

6.1 Práce se syntaxí

Nejzákladnější strukturou při práci se syntaxí je *syntaktický strom* (angl. *syntax tree*), který nese jak syntaktickou, tak sémantickou informaci o analyzovaném kódu. Syntaktické stromy v Roslynu obsahují veškeré informace, obsažené v analyzovaném kódu a to včetně bílých znaků, komentářů a direktiv preprocesoru. Rovněž jsou zde informace o chybách v analyzovaném kódu ve formátu přeskočených nebo chybějících symbolů. Syntaktické stromy rovněž umožňují konstrukci a editaci zdrojového textu úpravou již existujícího stromu. Důležitou vlastností syntaktických stromů je skutečnost, že jsou neměnné—jakmile je syntaktický strom vytvořen, nelze objekt tohoto stromu měnit. To však není v rozporu s výše zmíněnou možností úprav stromu, jelikož ty probíhají na *hluboké kopii* (angl. *deep copy*) již existujícího stromu.

Syntaktický strom je tvořen několika druhy objektů, jedná se například o:

- **syntaktický uzel**—jeden z nejhlavnějších elementů stromu, který v podstatě reprezentuje neterminály použité pro konstrukci stromu. Je reprezentován třídou `SyntaxNode`. Vždy obsahuje potomky a to buď opět typu `SyntaxNode` (pokud je potomek neterminál) nebo `ChildNode` (pokud je potomek symbol). Kořen stromu pak nemá žádného předchůdce.
- **syntaktický symbol**—terminály (symboly), reprezentující nejmenší části kódu, které jsou reprezentovány třídou `SyntaxToken`. Syntaktické symboly obsahují jak text ze zdrojového kódu, tak skutečnou reprezentaci své hodnoty typovanou jako `object`.
- **druh**—každý uzel a symbol má vlastnost `Kind`, která přesně specifikuje o jaký prvek syntaxe se jedná (např. `AddExpression`, `OperatorToken`, `Statement`).

6.2 Práce se sémantikou

Syntaktické stromy reprezentují lexikální a syntaktickou strukturu analyzovaného kódu, avšak to samo o sobě nestačí pro identifikaci toho, na co se jednotlivé příkazy a deklarace odkazují. Budeme-li mít například deklaraci obsahující proměnnou typu `var`, není možné

nijak určit o jaký konkrétní typ se bude jednat pouze z informací syntaktického stromu. Pomocí sémantické analýzy je možno získat skutečný typ takovéto proměnné.

Sémantická analýza v Roslynu obsahuje následující klíčové koncepty:

1. **Kompilace** představují vše, co je třeba pro kompilaci programu v C#. Obsahuje všechny deklarované typy, členy nebo proměnné jako *symboly*. Stejně jako syntaktické stromy i kompilace jsou neměnné a pro jejich změnu je třeba pracovat s kopiemi.

2. **Symboly** reprezentují unikátní element deklarovaný ve zdrojovém kódu nebo naimportovaný z jiné knihovny. Každý jmenný prostor, typ, metoda, vlastnost, událost, parametr nebo lokální proměnná je reprezentována jako symbol. Každý typ symbolu je reprezentován jako podtřída třídy `Symbol` a nese informace nasbírané během kompilace (např. návratový typ metody).

3. **Sémantický model** představuje veškeré sémantické informace, obsažené ve zdrojovém souboru. Těmito informacemi jsou:

   (a) Symboly odkazované na dané pozici ve zdrojovém souboru.

   (b) Výsledný typ daného výrazu.

   (c) Chyby a varování.

   (d) Rozsah platnosti proměnných.

6.3 Práce s pracovními prostory

Pracovní prostor (angl. *workspace*) je reprezentací konceptu *řešení* (angl. *solution*), užívaného Visual Studiem. Pracovní prostor je typicky svázán s hostitelským prostředím, které se neustále mění v závislosti na uživatelských akcích prováděných nad řešením a projekty. Pro každou takovou akci je pak vyvolána patřičná událost.

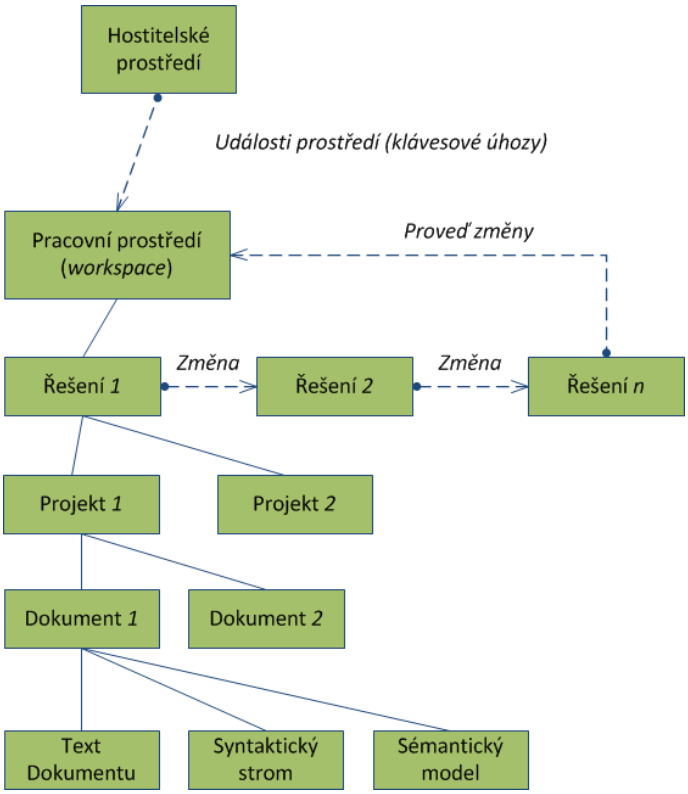
Pracovní prostor pracuje s následujícími koncepty:

1. **Solution** (řešení) je neměnný model obsahující projekty a v nich obsažené dokumenty.

2. **Project** (projekt) reprezentuje veškeré soubory obsahující zdrojový kód, nastavení kompilátoru a reference na assembly a jiné projekty.

3. **Document** (dokument) symbolizuje jeden zdrojový soubor. Poskytuje přístup k textovému obsahu tohoto souboru, jeho syntaktickému stromu a sémantickému modelu.

Obrázek schématicky znázorňuje model pracovního prostoru a způsob, jakým se propagují změny.



Obrázek 29: Model pracovního prostoru v Roslynu

7 Syntaktická a lexikální analýza Spark View Engine

V mém řešení jsem pro lexikální a syntaktickou analýzu zdrojového textu v šablonách Sparku využil generátoru Irony, který je popsaný v sekci 4.1. Výstupu lexikální analýzy je pak využito pro zvýrazňování syntaxe a výstup syntaktické analýzy je pak nezbytný pro implementaci doplňování slov.

Jelikož generování syntaktického a lexikálního analyzátoru v Irony je poněkud časově náročná operace (stovky milisekund), jejich definice a výstavba probíhá staticky—ve statickém konstruktoru třídy `SparkParsers`. To v důsledku znamená, že veškeré parsery jsou zkonstruovány pouze jednou v rámci běhu aplikace Visual Studia a nedochází tak ke zbytečnému zpomalování opětovnými výstavbami parserů.

7.1 Gramatiky pro Spark View Engine

Syntaxi sparku lze rozdělit do několika podmnožin, a sice:

1. **XML syntaxe sparku**—vlastní konstrukce definované Spark View Enginem.
2. **Inline kód**—deklarace jazyka C# uvozené znakem "#"a ukončené středníkem.
3. **Spark výrazy**—výrazy jazyka C# rozšířené o konstrukce Sparku. Jsou uzavřeny ve složených závorkách, které začínají znaky "$", "!" nebo "$!"Tyto výrazy pak mají speciální verzi, tzv. *boolean výrazy*, které jsou syntakticky totožné se Spark výrazy, s tím rozdílem, že boolean výrazy jsou uvozené znakem otazníku.

Při návrhu gramatiky jsem vycházel z těchto tří podmnožin a ke každé podmnožině jsem vytvořil gramatiku, která definuje pravidla syntaxe dané podmnožiny. Boolean výrazy a Spark výrazy mají pak gramatiku stejnou. Tyto podmnožiny pak tvoří hierarchii tříd, které obsahují definice gramatik pro Irony. Tato hierarchie je znázorněna třídím diagramem na obrázku 30. Jak je patrné z diagramu, na vrcholu této hierarchie je třída `Grammar` (viz pod-

Obrázek 30: Hierarchie tříd gramatik Sparku

sekce 4.1.5). Tuto třídu rozšiřuje třída `SparkExpressionGrammar`, která definuje gramatiku pro Spark výrazy. `SparkExpressionGrammar` je pak rozšířena o syntaxi inline kódu třídou `SparkStatementsGrammar`, jelikož syntaxe inline kódu je logické rozšíření syntaxe výrazů. Třídu `SparkExpressionGrammar` pak dále rozšiřuje gramatika definující XML syntaxi Sparku pojmenovaná `SparkGrammar`. Zde se sice úplně nejedná o logické rozšíření syntaxe inline kódu (ten uvnitř značek Sparku být nemůže), avšak XML Spark značky využívají určité konstrukty definované v inline kódu. Jako poslední je zde zadefinována gramatika pro intellisense `SparkIntellisenseGrammar`, která je logickým rozšířením gramatiky Sparku, ale mnohé konstrukty definované v jejím předku jsou zde předefinovány (bude popsáno dále). Při definici gramatik `SparkExpressionGrammar`

a `SparkStatementsGrammar` jsem vycházel z oficiální dokumentace gramatiky jazyka C #, kterou lze nalézt na [Microsoft(2010)].

7.1.1 Gramatiky Spark výrazů a inline kódu

`SparkExpressionGrammar` představuje pravidla syntaxe všech výrazů v jazyce C# s výjimkou anonymních metod a objektů a jiných typů výrazů, které nelze vyhodnotit jako textový řetězec. Tyto nejsou v gramatice zahrnuty, protože takové výrazy nejsou ve Sparku validní. Jsou zde však zadefinovány, protože jsou použity v gramatice pro inline kód. Dalším důležitým rozdílem oproti oficiální dokumentaci je způsob jakým je řešena priorita operátorů. Zatímco v oficiální dokumentaci je priorita řešena pomocí hierarchie neterminálů, v gramatice Spark výrazů je priorita řešena voláním metod Irony `RegisterOperators(int priority, string operator)`.

V této gramatice vznikl konflikt typu posun-redukce v situaci, kdy syntaktický analyzátor nemůže po přečtení symbolu "<"rozhodnout, jestli se jedná o operátor nebo uvození generického parametru. Tento konflikt je vyřešen přidáním speciální metody (`ResolveLessThanConflict()`), která projde vstupní text a podívá se na následující znaky. Pokud je nalezen symbol ">", který uzavírá generické parametry, je proveden posun. Pokud tento symbol nalezen není, jedná se o operátor a je provedena redukce. V gramatice je pak v neterminálu, který definuje generické parametry, zavolána tato metoda způsobem ilustrovaným na výpisu kódu 15.

```
1  genericLessThan.Rule = CustomActionHere(this.ResolveLessThanConflict) + this.ToTerm("<");
```

Výpis 15: Vyřešení konfliktu operátoru a uvození generického parametru

Gramatika `SparkStatementsGrammar` obsahuje definice pravidel pro deklarace v C# a to bez jakýchkoliv výjimek z oficiální dokumentace. Jsou zde tedy konstrukce pro:

- **přiřazování a deklarace proměnných**,
- **delegáty, anonymní metody a typy**,
- **podmínky, cykly, přepínače**,
- **bloky zachycující výjimky** (`try`, `catch` a `finally`),
- **speciální výrazy** (`yield return`, `checked`, `unchecked` atd).

V této gramatice vznikl další konflikt. Jedná se o tzv. konflikt "visícího else", který zmiňuji v podsekci 3.3.2. Vyřešení tohoto konfliktu je přímočaré—je preferována akce posunu. Prakticky to pak znamená zavolat metodu `PrefferShift()`, což ilustruje výpis kódu 16.

```
1  ifStatement.Rule = this.ToTerm("if") + LeftParenthesis + Expression + RightParenthesis +
       embededStatement | this.ToTerm("if") + LeftParenthesis + Expression + RightParenthesis +
       embededStatement + PreferShiftHere() + this.ToTerm("else") + embededStatement;
```

Výpis 16: Vyřešení konfliktu visícího else

7.1.2 Gramatiky pro XML kód Sparku a doplňování slov

Pro gramatiku Spark XML konstrukcí neexistuje oficiální specifikace a při její definici jsem vycházel z dokumentace na [DeJardin(2008)]. Tato gramatika nedefinuje kompletní strukturu Sparku v tom smyslu, že jednotlivé značky nejsou párovány a není zde ani řešeno zanořování značek. Jedná se tedy čistě o definici jednotlivých otevíracích, uzavíracích a samozavíracích značek definovaných Sparkem. K tomuto řešení jsem přistoupil ze dvou důvodů: LALR parser, který jsem v mém řešení použil, je poněkud citlivý na definici vstupní gramatiky (viz sekce 3.3). V případě Sparku by byl parser příliš náchylný ke konfliktům, jelikož Spark je vlastně kombinací dvou syntakticky dost odlišných jazyků. V důsledku by pak nemuselo být vůbec možné takovouto gramatiku zadefinovat. Druhý důvod je skutečnost, že definice takovéto gramatiky je jednodušší, přehlednější a čitelnější a tudíž se v ní lépe orientuje.

V takto zadefinované gramatice sice vzniknou situace, kdy v případě špatného spárování značek bude takový kód zpracován a nedojde k detekci chyby, avšak toto lze snadno ošetřit dodatečnou kontrolou struktury zdrojového kódu v šablonách Sparku.

Poznámka 7.1 Ve výrazech Sparku a v inline kódu vše funguje správně a např. neuzavřený blok `if` **bude** označen jako chybný.

V gramatice pro XML značky Spark jsou mimo jiné definovány terminály pro:

- **klíčová slova Sparku**—`for`, `if`, `macro`, `include`, `viewdata` ...
- **XML značky** (jména XML značek, jejich atributy a hodnoty atributů)—pro případ, kdy lze aplikovat některé značky Sparku na již existující XML nebo HTML značky. V dokumentaci Sparku jsou tyto označovány jako tzv. *smíšené značky* (angl.*mixed tags*).
- **XML prefixy**—pro označení jmenného prostoru dané značky.
- **XML komentáře**,
- **Spark výrazy**—hodnoty některých atributů některých Spark značek. Definice těchto neterminálů jsou poděděny ze třídy `SparkExpressionGrammar` přes třídu `SparkStatementsGrammar` (viz obrázek 30).

Poslední gramatika je gramatika pro doplňování slov, která je definována ve třídě `SparkIntellisenseGrammar`. Terminály pro Spark XML značky jsou poděděny ze třídy `SparkGrammar`, ovšem většina atributů těchto značek je zde předefinována. Toto předefinování jsem provedl proto, abych byl schopen rozlišit, o jaký atribut se přesně jedná. Ve třídě `SparkGrammar` jsou tyto atributy často definovány jako Spark výrazy, identifikátory apod., protože taková je jejich syntaxe. Pro potřeby doplňování slov je však třeba vědět, zda-li se jedná o atribut například globální či lokální proměnné. Takové atributy jsou syntakticky stejné, avšak pro nabídnutí validních doplnění je třeba je nějak odlišit. Tuto problematiku budu ilustrovat na příkladu značek `var` a `set`. Syntaxe a sémantika těchto značek je znázorněna výpisem kódu 17. Při deklarací proměnné značkou

`var` je atribut `jmenoPromenne` jakýkoliv text, který je validní jako XML atribut a stejně tak v případě značky `set`. Syntakticky jsou si tedy tyto atributy totožné. Sémanticky je zde rozdíl—atribut `jmenoPromenne` značky `set` může být pouze jméno proměnné, která je platná v daném rozsahu. Předefinováním terminálu pro `jmenoPromenne` značky `set` jej tedy sémanticky rozlišíme od atributu `jmenoPromenne` značky `var`.

```
1  <var jmenoPromenne="hodnotaPromenne"/>
2  <set jmenoPromenne="hdnotaPromenne"/>
```

Výpis 17: Syntaxe značky viewdata

Programově je toto předefinování řešeno tak, že ve třídě, ve které má být terminál předefinován, je zadefinován zcela nový terminál. Proměnná, specifikující neterminál generující terminál k předefinování je typu `protected`, a celé pravidlo gramatiky je pak v potomku definováno znovu, avšak s jiným terminálem. Tento terminál je pak potomkem třídy `Terminal` a jeho rozeznání lexikálním analyzátorem je pak naprogramováno ručně. Pro výše uvedený příklad to znamená, že neterminál generující značky *set* ve třídě `SparkGrammar`, je zadefinován jako člen této třídy s modifikátorem `protected`, jak ilustruje výpis kódu 18. Ve třídě `SparkIntellisenseGrammar` je neterminál `Set` předefinován jak ukazuje kód 19. Na řádku 6 je zadefinován nový terminál `VariableNameTerminal` a ten je pak použit pro předefinování pravidla na řádcích 9 a 10. Třída `VariableNameTerminal` je pak implementována podobným způsobem, jako v příkladě na obrázku 23 v kapitole 4.2. Je tu však jeden důležitý rozdíl—v případě, že terminál `VariableNameTerminal` byl identifikován, do parseru je uměle přidána chyba voláním `context.AddParserError("Syntax error, expected: variableType-terminal")`. Této chyby je pak využito pro identifikaci terminálu `VariableNameTerminal`. Důvody potřeby identifikace terminálů jsou detailněji popsány v podsekci 9.2.2.

```
1  [Language("Spark", "1.6", "Spark␣View␣Engine␣Grammar")]
2  public class SparkGrammar : SparkStatementsGrammar
3  {
4      protected NonTerminal SetTag;
5       public SparkGrammar()
6       {
7           ...
8          SetTag = new NonTerminal("set−tag");
9           ...
10         \\ Definice gramatickych pravidel znacky set
11     }
12 }
```

Výpis 18: Deklarace neterminálu pro značku `set`

```
1  [Language("Spark␣Intillisense", "1.6", "Spark␣View␣Engine␣Intellisense␣Grammar")]
2  public class SparkIntellisenseGrammar : SparkGrammar
3  {
4      public SparkIntellisenseGrammar()
5      {
```

```
6           VariableNameTerminal variableNameTerminal = new VariableNameTerminal("variable-
                name-terminal");
7           SetTag.Rule = this.ToTerm("set") + setTagAssignments;
8           setTagAssignments.Rule = this.MakeStarRule(setTagAssignments, setTagAssignment);
9           setTagAssignment.Rule = variableNameTerminal + equals + "\"" + Expression + "\""
10                                | variableNameTerminal + equals + "'" + Expression + "'";
11      }
12  }
```

Výpis 19: Předefinování značky `set` pro potřeby intellisense

8 Zvýrazňování syntaxe v šablonách Spark View Engine

Obecná implementace zvýrazňování syntaxe ve Visual Studiu 2012 je popsána v podsekci 5.6.2 a mnou implementované řešení je založeno principech popsaných v této podsekci.

Pokud chceme rozšířit Visual Studio o podporu Spark View Engine, je nejprve potřeba zadefinovat nový typ obsahu pro Spark. Způsob, jakým se obecně provádí definice obsahu, je popsán v podsekci 5.6.1. Protože Spark je ve své podstatě XML, definice obsahu Sparku je rozšířením již existujícího typu obsahu XML. Dále je nutno svázat soubory obsahující Spark kód s tímto nově definovaným obsahem (viz kapitola 5.6.1). Toto svázání je provedeno pro soubory s koncovkou *.spark*, ale také pro XML soubory, protože Spark šablony mohou být definovány i v souborech s koncovkou *.xml* (např. soubor *Bindings.xml*). Definice a svázání obsahu s příponami je provedeno ve třídě `SparkClassifierProvider`.

Výše popsané řešení není úplně ideální. Rozšíření typu XML totiž znamená, že musí být dodrženy veškeré konvence XML a kód Sparku tedy musí být validním XML. V případě, že tomu tak není, editor toto detekuje a zobrazí chybu. Problém vniká v situaci, kdy je v editoru validní Spark kód, nesplňující konvence XML, následkem čehož dojde k označení validního Spark kódu jako chybový. Příkladem takové situace je kód, obsahující více kořenových XML/Spark elementů nebo výskyt znaku "<"v XML/Spark atributu—v XML není povoleno více kořenových elementů, stejně tak se v atributech nesmí vyskytovat znak "<"; ve Sparku se však jedná o naprosto validní kód.

Podle oficiální dokumentace na stránkách Microsoftu je tato situace řešitelná užitím konceptu projekce (angl. *projection*). Dokumentace sice slovně zmiňuje jak použít projekci, avšak jedná se dle mého o poněkud vágní popis, z něhož jsem přes veškerou mou snahu nebyl schopen zjistit, jak konkrétně projekci použít v kódu. V jiných zdrojích jsem bohužel řešení tohoto dost specifického problému také nenašel.

8.1 Klasifikace značek Sparku

Třída `SparkClassifierProvider` exportuje (viz podsekce 5.3) objekty značek dvojího typu:

1. `ClassificationTag` (klasifikační značka)—nese informace o pozici a typu Spark značky,
2. `ErrorTag` (chybová značka)—nese informace o pozici chyby ve zdrojovém textu Spark šablony.

Tyto objekty jsou získávány ze třídy `SparkClassifier<T>`, kde generický typ `T` slouží pro identifikaci typu značky (klasifikační nebo chybová). Metoda `GetTags()` (viz. 5.6.2) pak v závislosti na hodnotě parametru `T` získává značky požadovaného typu. Tyto značky jsou získávány ze třídy `SparkTokenTagger`—opět voláním metody `GetTags()`—a to přes objekt `ITagAggregator<SparkToken> sparkTagAggregator` předaný v konstruktoru.

Ještě před vrácením je značkám přiřazen jejich klasifikační typ (v případě že se jedná o klasifikační značky). Tento typ je přiřazen v závislosti na tom, jaký typ je vrácen ze značkovače `SparkTokenTagger`; může být dvojího druhu:

1. typ definovaný Sparkem—nově zadefinovaný typ značek definovaných Sparkem.
2. Již existující typ.

Nové typy značek, které zavádí Spark, logicky vyplývají z definice tohoto jazyka. Také jsem se inspiroval alternativním řešením od firmy Microsoft—*ASP.NET MVC Razor*. Tyto typy jsou:

1. `SparkElement`—podbarvení všech Spark elementů,
2. `SparkElementBorder`—hraniční značky Spark elementů,
3. `SparkKeyword`—klíčová slova (`if`, `test`, `try`, `include` ...),
4. `SparkString`—textové řetězce,
5. `SparkNumber`—číselné konstanty,
6. `SparkText`—vše ostatní co nespadá do výše jmenovaných kategorií.

Již existující typy jsou použity ve značkách míchající syntaxi Sparku a XML; jedná se o typy pro:

1. `SparkXmlTagName`—názvy XML značek (*HTML Element Name*),
2. `SparkXmlAttributeName`—jména XML atributů (*HTML Attribute Name*),
3. `SparkXMLAttributeValue`—hodnoty XML atributů (*HTML Attribute Value*).

Nastavení vzhledu Spark typů je načítáno z konfigurace Visual Studia a lze jej kdykoliv změnit z menu Visual Studia. Názvy v závorkách jsou identifikátory typů, které používá Visual Studio. U Spark typů pak název typu přímo odpovídá identifikátoru ve Visual Studiu. Pod těmito názvy lze nalézt nastavení příslušného typu v menu Visual Studia. Způsob, jakým se tato nastavení provádí, je popsán návodu k použití, který se nachází v příloze B.

8.2 Víceřádkové značky

Rozhraní `ITagAggregator` definuje událost `TagsChanged`. Pokud dojde k vyvolání této události nad nějakým rozsahem textu, je zavolána metoda `GetTags()` a jsou vráceny značky vyskytující se v daném rozsahu. Ve třídě `SparkClassifier<T>` je událost `TagsChanged` obsloužena vyvoláním vlastní události `TagsChanged`, což v důsledku vede k zavolání metody `GetTags()` a následnému vrácení značek pro daný rozsah textu.

Ve třídě `SparkTokenTagger` je pak zadefinovaný posluchač události `ITextBuffer.Changed`. Tato událost je vyvolána, pokud je změněn obsah okna editoru. V posluchači této události se zjistí, jaká editace byla provedena a v závislosti na této události dojde k aktualizaci seznamu značek nacházejících se v editoru. Pokud tedy byla například změněna pozice značky, v seznamu značek se vyhledá značka, jejíž pozice byla změněna a tato pozice se aktualizuje v objektu reprezentujícím změněnou značku. Nakonec je vyvolána událost `TagsChanged` pro změněné pozice, v důsledku čehož je provedená změna propagována do okna editoru.

Celý tento poněkud komplexní mechanizmus je nutný, protože metoda `GetTags()` je volána jenom pro změněný text. Pokud by tedy například byla změněna značka, která se rozprostírá přes 3 řádky, a ke změně by došlo na třetím řádku této značky, metoda `GetTags()` by pak dostala pouze obsah třetího řádku této značky. Značka by tedy byla značně neúplná a nebylo by možno provést klasifikaci, poněvadž takto neúplná značka by neprošla syntaktickou analýzou. Jelikož však máme k dispozici aktuální seznam značek, není problém zjistit celý textový obsah značky a provést syntaktickou analýzu.

Značky v každém okně editoru jsou uchovávány v kolekci typu `SortedSet<SparkTokenTag>`. Jedná se o kolekci unikátních objektů značek, která je navíc vzestupně setříděná podle pozice značky. Tato kolekce je pak udržována v objektu `ITextBuffer _buffer`. Správce MEF vytváří tento objekt bufferu pro každé okno editoru. Buffer je pak možné dle potřeby importovat a pracovat s jeho obsahem. Aktuální seznam značek je pro každé okno editoru uchováván v tomto objektu bufferu a značky jsou tak dostupné v rámci celé aplikace—stačí pouze provést import bufferu v místě, kde chceme ke značkám přistupovat.

8.3 Parsování elementů

Veškeré elementy Sparku jsou představovány třídou `SparkTokenTag`. Objekty této třídy vznikají ve statické třídě `ElementLocator`, která obsahuje metody pro nalezení elementů Sparku z objektu `SnapshotSpan`. Tento objekt představuje verzi části obsahu okna editoru (snímek) a je předáván z metody `GetTags`. Vyhledávání probíhá zvláště pro

- **inline kód**—metoda `GetInlineCode()`,
- **XML tagy Sparku**—metoda `GetSparkTags()`,
- **výrazy Sparku**—metoda `GetExpressions()` a
- **boolean výrazy Sparku**—metoda `GetAttributeBooleanExpressions()`.

Syntaxe boolean a Spark výrazů je sice v podstatě stejná, avšak jejich vyhledávání je odděleno, protože boolean výrazy jsou validní pouze v atributech XML značek.

Samotná lokace elementů pak probíhá postupným procházením části textového obsahu objektu snímku editoru `SnapshotSpan`. Také je zde brán v úvahu možný předchozí nebo následující kód, který nemusí být obsažen ve snímku editoru (viz víceřádkové značky v kapitole 8.2). Vyhledávají se znaky, které by mohly indikovat přítomnost hledaného

elementu (např #, < apod.) a za asistence sady regulárních výrazů a ručního dohledání je lokalizována jejich poloha. Poté je vytvořen objekt značky `SparkToken`, který mimo jiné obsahuje:

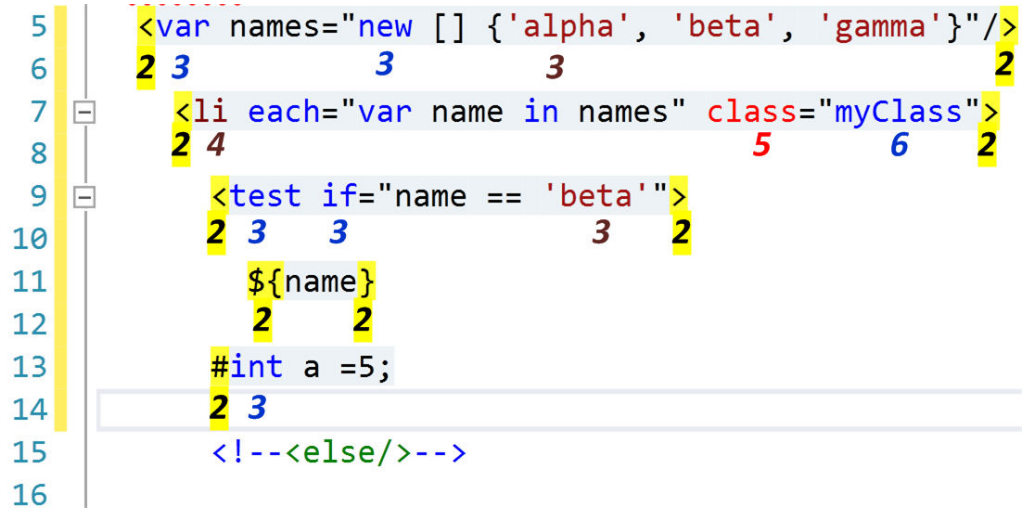
- **typ elementu**—inline kód, Spark boolean výraz, Spark výraz nebo XML značka,
- **rozsah elementu**—pozice značky ve snímku editoru,
- **textový obsah elementu** a
- **rozsah chyby elementu**—začátek a konec syntaktické chyby elementu (pokud nějakou obsahuje).

Jakmile jsou lokalizovány jednotlivé elementy nacházející se v okně editoru, lze přistoupit k jejich syntaktické analýze. Ve třídě `SparkTokenTagger` (metoda `GetInnerTokens()`) je volána metoda z Irony, která provádí parsování užitím patřičného objektu parseru ze třídy `SparkParsers`. Výsledkem tohoto volání je kolekce objektů typu `Token` (symbolů), které jsou obsaženy v daném elementu. Každý objekt `Token` má pak vlastnost `token.EditorInfo.Color`, která říká, o jaký typ symbolu se jedná. Na základě hodnoty vlastnosti `Color` je pak danému symbolu přiřazena hodnota z enumerace `SparkTokenType`, která definuje hodnoty popsané v 8.1. Rovněž je vypočtena a přiřazena pozice symbolu v editoru. Jestliže jsou při parsování objeveny nějaké chyby, jejich pozice je taktéž přepočítána do souřadnic editoru a přiřazena danému elementu.

Nakonec jsou jednotlivé symboly vráceny metodou klasifikátoru `GetTags()` do metody `GetTags()`, objektu třídy `SparkClassifier`. V závislosti na typu vráceného symbolu je pak voláním metody `typeService.GetClassificationType()` získán patřičný klasifikační typ daného symbolu. Ten je následně tomuto symbolu přiřazen a celý symbol je vrácen prostředí editoru Visual Studia jako objekt typu `ClassificationTag`. Výsledkem je zobrazení symbolu v patřičném formátu (barva textu, podbarvení apod.) ilustrované obrázkem 31. Čísla na obrázku označují jednotlivé klasifikační typy:

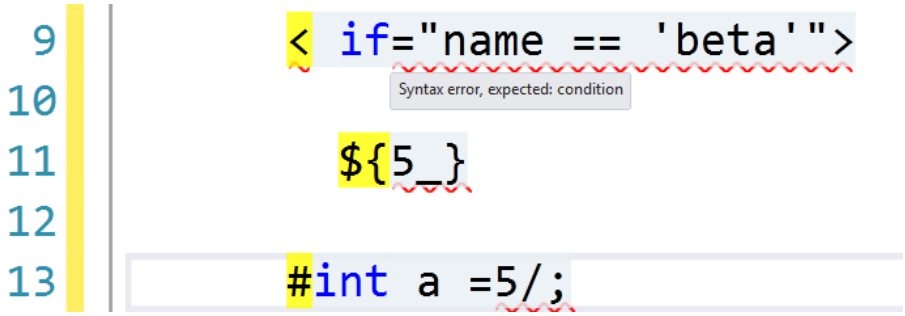
2. hranice elementů,
3. klíčová slova,
4. textové řetězce,
5. názvy XML značek, které nejsou definovány ve Sparku,
6. názvy atributů XML značek, které nejsou definovány ve Sparku,
7. hodnoty atributů, které nejsou definovány ve Sparku.

Zbylý černě zobrazený text odpovídá klasifikaci `SparkText`. Všechny Spark elementy jsou pak podbarveny světle modře—klasifikační typ `SparkElement`. XML komentáře a jiné XML značky, které nejsou součástí Sparku jsou ignorovány a zobrazeny dle nastavení Visual Studia.



Obrázek 31: Ukázka zvýrazňování syntaxe Spark Elementů

Co se podtržení chyb týče, celý proces je následovný:
pokud jsou objektem `SparkClassifier` vyžádány chybové značky, nedochází k opětovné syntaktické analýze kódu. V objektu bufferu (viz podsekce 8.2) jsou vyhledány všechny elementy obsahující chyby a ty jsou vráceny metodě v `GetTags()` objektu třídy `SparkClassifier`. Zde je pak pro každý chybový element vytvořen objekt typu `ErrorTag`, jenž je vrácen prostředí editoru Visual Studia. Následně je daný chybový rozsah textu zobrazen s červeným podtržením indikujícím syntaktickou chybu, což ilustruje obrázek 32. Po najetí kurzoru myši nad chybový kód (řádek 9), je ve formě bublinové nápovědy zobrazena chybová hláška vygenerovaná syntaktickým analyzátorem (řádek 10).



Obrázek 32: Ukázka zvýrazňování chyb Spark Elementů

9 Doplňování slov v šablonách Spark View Engine

Obecná implementace podpory doplňování slov ve Visual Studiu 2012 je popsána v podsekci 5.6.3 a mnou implementované řešení je založeno principech popsaných v této podsekci.

Nejprve popíšu implementaci naslouchání uživatelským akcím, které jsou relevantní pro doplňování slov. Tento mechanizmus je implementován ve třídě `VsTextViewCreationListener`, a především pak ve třídě `CompletionController`. Typ obsahu pro Spark šablony jsme již definovali v kapitole 8, nyní pouze stačí napojit posluchače uživatelských akcí na okna editoru obsahující Spark kód. Toto napojení ilustruje výpis kódu.

```
1 [Export(typeof(IVsTextViewCreationListener))]
2 [ContentType("spark")]
3 [TextViewRole(PredefinedTextViewRoles.Interactive)]
4 internal sealed class VsTextViewCreationListener : IVsTextViewCreationListener
5 {}
```

Výpis 20: Napojení intellisense na Spark obsah

Samotné svázání s obsahem ilustruje kód na druhém řádku. Ten říká správci MEF rozšíření Visual Studia, že má zachytávat veškeré uživatelské akce pro nově vytvořená okna editoru Visual Studia, jejichž obsah je typu "spark". Řádek 3 pak blíže specifikuje, že naslouchání bude prováděno na oknech, v nichž je možno provádět akce pomocí klávesnice a/nebo myši.

9.1 Zachytávání příkazů z rozhraní editoru

V konstruktoru třídy `VsTextViewCreationListener` je pak vytvořen objekt třídy `CompletionController`, který je následně specifikován jako obsluha uživatelských akcí. Tato třída je ve své podstatě filtr, který v závislosti na tom, jaká klávesa byla stisknuta, vyváří okno se slovy, jenž lze zapsat do editoru na dané pozici (angl. *statement completion window*). V implementaci metody `QueryStatus()` neustále probíhá kontrola, zda-li nebyla v uživatelském rozhraní provedena nějaká akce. Pokud je stisknuta klávesa relevantní pro doplňování slov, je tato akce zachycena. Obsluha akce pak probíhá v implementaci metody `Exec()`. V této metodě se zjišťuje, jaká akce byla vykonána, a v závislosti na této akci se provede:

- vytvoření sezení doplňování slov—voláním metody `StartSession()`,
- dokončení sezení—voláním metody `Complete()`,
- filtrování nabízených slov k dokončení—metoda `Filter()`,
- zrušení aktivního sezení—metoda `Cancel()`.

Sezení je vytvářeno pokud jsou zachyceny klávesové úhozy, které mají vyvolat okno nabízející slova k doplnění. V případě Sparku jsou to například znaky "<", "#", "{"apod.

Dokončení sezení, nebo-li potvrzení vložení vybraného slova, pak probíhá stisknutím klávesy *TAB*, nebo dvojklikem na požadované slovo. Sezení je také ukončováno, pokud je napsán znak, který uzavřel a tedy ukončil element (např znak ”>”).

Filtrování slov probíhá po napsání nebo smazání textového znaku a to tak, že v seznamu nabízených slov je označeno to slovo, které je nejvíce podobné již napsané části slova. Zrušení aktivního sezení je pak typicky prováděno po stisknutí klávesy *ESC*, nebo po kliknutí mimo okno s nabízenými slovy. Rozhodnutí o akci, která bude provedena, závisí na elementu, který se nachází poblíž kurzoru editoru.

9.2 Získávání slov pro doplnění

Poté co je aktivováno sezení, je potřeba získat slova, která se mají zobrazit. Tato slova poskytuje objekt třídy `SparkCompletionSource`, konkrétně jeho metoda `AugmentCompletionSession`, která je volána z objektu `CompletionController` při aktivaci sezení. Pokud Visual Studio již vytvořilo seznam slov pro doplnění, doplňovaná slova Sparku jsou přidána do tohoto existujícího seznamu. V takovém již existujícím seznamu jsou například uzavírací značky XML elementů—tento seznam je automaticky generován Visual Studiem, jelikož typ obsahu pro Spark rozšiřuje typ obsahu XML (viz sekce 8).

Slova pro doplnění jsou získávána v závislosti na tom, jestli na místě, kde se má zobrazit okno se slovy, byl rozpoznán element Sparku. Pokud zde žádný takový element není, je zobrazen seznam všech značek Sparku. Rovněž je třeba vzít v úvahu, zda-li sezení nebylo aktivováno ze souboru *Bindings.xml*, protože v tomto speciálním typu Spark souboru je validní jenom omezená množina značek. Získávání seznamu všech značek dále popisuje podsekce 9.2.1.

V případě, že v místě zobrazení okna se slovy existuje Spark element, je zobrazen seznam slov v závislosti na pozici kurzoru v tomto elementu. Existence či neexistence elementu je určena z aktuální pozice kurzoru a seznamu všech rozpoznaných značek. Tento seznam obsahuje objekty typu `SparkTokenTag` a je přístupný z naimportovaného objektu `ITextBuffer`(viz podsekce 8.2).

Poznámka 9.1 Při získávání slov k doplnění, je často potřeba analyzovat obsah jiných souborů obsažených v projektu. Nabízí se řešení načítat tento obsah ze souborového systému. Toto řešení má však jednu nevýhodu—pokud bude soubor, jehož obsah je potřeba načíst, otevřený v editoru a budou v něm neuložené změny, tento soubor bude obsahově odlišný od toho, který je uložený na pevném disku. Z tohoto důvodu je v mém řešení nejprve ověřeno, zda-li soubor, jehož obsah je třeba načíst, je otevřen v okně editoru. Pokud ano, je získán obsah, který se nachází v okně editoru včetně neuložených změn a nikoliv na souborovém systému. Pokud soubor není otevřený v editoru, je přistoupeno na souborový systém.

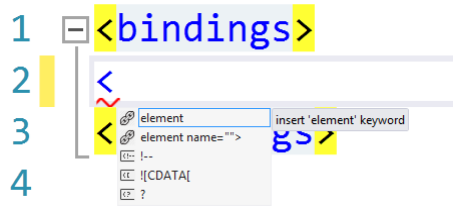
9.2.1 Získávání seznamu všech značek

Kontextově závislé značky Při získávání seznamu všech Spark značek je třeba mít na paměti, že Spark View Engine obsahuje značky, které nejsou validní v každém kontextu. Patří zde:

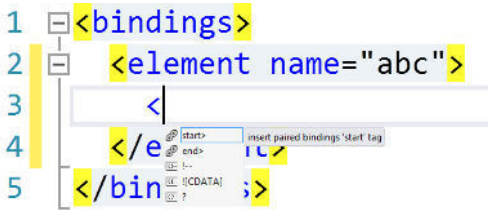
- značky pro podmínky `else`, `elseif` a `else if`,
- a značky v souboru *Bindings.xml*: `element`, `start` a `end`.

V praxi to pak znamená, že při nabízení doplnění všech značek Sparku, je nejprve nutno zjistit, která z výše jmenovaných značek se může zobrazit v aktuálním kontextu. Toto zjištění má na starosti třída `CompletionProvider`. Metody této třídy přijímají jako parametr seznam značek, jenž se nacházejí před aktuální pozici kurzoru. Z názvů a pořadí předcházejících značek je pak určeno, které kontextově závislé značky mají být v aktuálním kontextu zobrazeny.

Obrázky 33 a 34 ilustrují funkci kontextově závislých doplnění uvnitř značky `bindings`. Z obrázku 33 je patrné, že jediné validní doplnění v tomto kontextu je značka `element`. Obrázek 34 pak ukazuje, že uvnitř značky `element` lze doplnit pouze značky `start` a `end`.

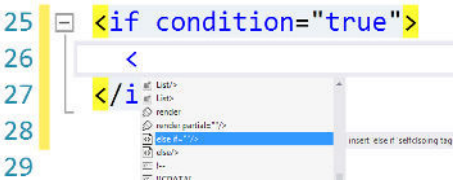


Obrázek 33: Nabízená doplnění uvnitř značky `bindings`

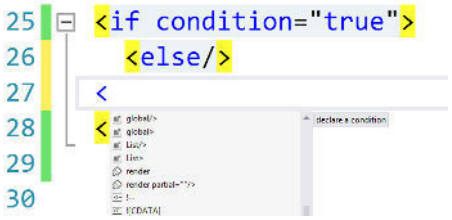


Obrázek 34: Nabízená doplnění uvnitř značky `element`

Jiným příkladem zobrazení doplnění, jejichž přítomnost je závislá na kontextu, jsou doplnění uvnitř značky `if`. Obrázek 35 ukazuje, že uvnitř značky `if`, která neobsahuje žádné značky `else`, lze doplnit značku `else`. Pokud však uvnitř značky `if` již existuje značka `else`, nelze tuto značku doplnit, což ilustruje obrázek 36.



Obrázek 35: Nabízená doplnění uvnitř značky `if`



Obrázek 36: Nabízená doplnění uvnitř značky `if` obsahující značku `else`

Značka `viewdata` Tato značka je svou sémantikou poněkud specifická, jelikož v jejích atributech mohou být pouze názvy proměnných předané do šablony Sparku (*view*) v objektu *controlleru*. Controller je objekt známý z konceptu *Model-View-Controller*, který řídí celý proces zobrazení webové stránky. Obsahuje metody akcí, které určují, jaký view (webová stránka) bude pro danou akci zobrazena. Při implementaci nabízení značky `viewdata`, je tedy nutno mít k dispozici informaci o proměnných, které se mohou být předávány z controlleru.

Dle konvence jsou šablony Sparku uloženy ve složce *Views*, která obsahuje podsložky, jejichž název je zároveň názvem view. Controller patřící danému view je pak prefixován názvem složky tohoto view. Budeme-li tedy například mít šablonu Sparku, jenž bude mít cestu `/Views/Home/Index.spark`, controller šablony `Index.spark` bude pojmenován `HomeController`. Z této konvence také vychází mnou implementované řešení získávání platných názvů proměnných, které mohou být v atributu značky `viewdata`.

Získání proměnných pro značku `viewdata` probíhá v několika krocích. Pro analýzu kódu controllerů je využito Roslynu (viz sekce 6). Nejprve je získán textový obsah třídy controlleru právě editované šablony. Z tohoto obsahu je pak vytvořen objekt dokumentu Roslynu (viz kapitola 6.3). Následně jsou procházeny jednotlivé metody definované v objektu controlleru. Dále je z návratového typu každé metody zjištěno, zda-li se jedná o akční metodu a pokud ano, je analyzováno její tělo. V těle metody jsou pak vyhledána všechna přiřazení proměnných do objektu `Viewdata`, který slouží jako nosič dat mezi view a controllerem. Pro každé přiřazení je uložen název proměnné (užitím syntaktické analýzy Roslynu) a její typ (užitím sémantické analýzy Roslynu). Z těchto dat je pak složen obsah nabízených `viewdata` značek. Celý tento proces probíhá za pomoci metod, definovaných ve třídě `ViewDataRetriever`.

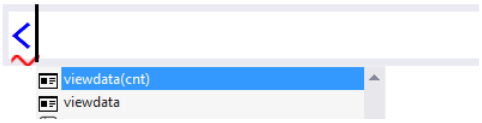
Doplňování značky `viewdata` ilustrujeme na příkladu. Mějme třídu `HomeController` a v ní akční metodou `About()` definovanou na kódem obrázku 37. V této metodě je zadefinována proměnná `count`, která je typu `int`. Tato proměnná je následně přiřazena do kolekce `ViewData` s identifikátorem `cnt`. Nakonec je vrácen view `About`, obsahující tuto proměnnou. Pokud pak vyvoláme doplnění v souboru *Views/Home/About.spark*, do-

```
5   public class HomeController : Controller
6   {
7       public ActionResult About()
8       {
9           int count = 10;
10          ViewData["cnt"] = count;
11          return View("Data/About");
12      }
13  }
```

Obrázek 37: Definice a předání proměnné do `viewdata`

staneme nabídku zobrazenou na obrázku 38. Označený řetězec `viewdata(int)` představuje doplnění pro proměnnou `cnt` předanou do view `About.spark` z controlleru

`HomeController`. Pokud bude stisknuta klávesa *TAB*, do editoru bude doplněna značka znázorněná na obrázku 39, a to včetně správného datového typu (`int`).



Obrázek 38: Nabízená doplnění značky `viewdata`

```
16  <viewdata cnt="int" />
```

Obrázek 39: Vložený kód pro doplnění označené kurzorem na obrázku 38

Značky partial souborů Spark View Engine obsahuje koncept tzv. *partial* souborů. Ve své podstatě se jedná o soubory prefixované znakem "_", které mají koncovku ".spark" případně ".xml". Pokud se v projektu nachází takový soubor, je automaticky vytvořena definice nové značky, jejíž název je shodný s názvem partial souboru bez úvodního podtržítka. Takové značky jsou rovněž zahrnuty v seznamu všech nabízených značek. Získání názvu těchto značek je přímočaré—projdou se všechny soubory aktivního projektu s koncovkou ".spark"nebo ".xml", od jejich názvu se odebere znak "_"a tento upravený název je pak přidán do seznamu všech nabízených značek.

9.2.2 Získávání slov pro existující element

Analýza kódu v aktuálním kontextu Celý proces získávání slov pro doplnění uvnitř existujícího elementu začíná v metodě `AddTagCompletions()`. Zde se provádějí následující analýzy:

1. nalezení naimportovaného obsahu
2. Zjištění, zda-li se kurzor nachází v cyklu `foreach`.
3. Nalezení proměnných, které jsou platné v aktuálním kontextu.
4. Zjištění, která makra jsou platná v daném kontextu.
5. Nalezení značek typu `using`.
6. Nalezení zahrnutých *assembly*.

Ad 1) Spark definuje značky, pomoci jichž lze do šablony zahrnout obsah a to v závislosti na hodnotě atributu dané značky. Těmito značkami jsou `include file`, `use file`, `use import` a `include href`. Všechny tyto značky jsou nalezeny a je uchován seznam značek, které se nalézají v obsahu souboru, na nějž značky odkazují. Toto vyhledávání probíhá rekurzivně, jelikož soubor odkazovaný danou značkou může rovněž obsahovat značky zahrnující obsah, a v tomto obsahu mohou opět být nějaké značky zahrnující obsah atd. Do výsledného seznamu značke jsou také přidány značky ze souboru "_global.spark", jehož obsah je platný v celém projektu. Vyhledávání a sběr značek probíhá v metodě `GetIncludedFilesTags` ve třídě `UseFileIncludeRetriever`.

Ad 2) Při vyhledávání cyklů jsou procházeny značky, nacházející se před aktuální pozicí kurzoru. Každá nespárovaná značka `for` je uchována, a následně je pro ni vygenerován ekvivalentní C# kód. Nespárované značky `for` je nutno mít k dispozici, protože deklarují nové proměnné s omezených rozsahem platnosti.

K lokaci nespárovaných značek (jakéhokoliv typu, nejen `for`) je využito struktury zásobníku. Kód šablony se prochází od první značky a nalezené otevírací značky požadovaného typu jsou postupně vkládány na vrchol zásobníku. Pokud je nalezena uzavírací značka stejného typu, z vrcholu zásobníku je jedna značka vyjmuta. Nakonec jsou v zásobníku pouze ty značky, které nejsou spárované.

Ad 3) V této fázi jsou procházeny všechny značky získané v bodě 1, jakožto i značky nacházející se před aktuální pozicí kurzoru. Nejprve jsou nalezeny proměnné deklarované značkami `viewdata`, `global`, `content` a `var`, které jsou platné v aktuálním kontextu. Pro každou proměnnou je uchován její název a typ. Toto probíhá v metodě `GetVariablesInScopeContents()` třídy `VariableRetriever`. Do seznamu proměnných jsou rovněž přidány proměnné deklarované ve značkách `render partial`, `render segment` a `segment`. Nakonec je pro každou proměnnou vygenerován kód v C#, představující deklaraci této proměnné. Pokud lokální proměnná nemá nastaven typ, je deklarována jako `var`. Netypované globální proměnné jsou deklarovány s typem `object`. Vyhledání a generování kódu proměnných probíhá v metodě `GatherVariables()` třídy `CompletionSource`.

Ad 4) Platná makra jsou rovněž vyhledávána ve značkách předcházejících aktuální pozici kurzoru a ve značkách nalezených v bodě 1. Pro každé nalezené makro je vygenerován C# kód ve formě statické metody, s parametry pojmenovanými a typováními ve shodě s definicí makra. Za vyhledání a generování kódu maker je zodpovědná metoda `GetMacrosDefinitionsFromTags` definovaná ve třídě `BindingsRetriever`.

Ad 5) Vyhledání značek `using` probíhá ve stejném obsahu jako v předchozích bodech. Pro každou značku `using` je rovněž vygenerován kód jazyka C#, který odpovídá syntaxi direktivy `using` definované v C#. Vyhledání a tvorba kódu direktiv `using` je realizováno v metodě `GetUsings()` ve třídě `UsingDirectiveRetriever`.

Ad 6) Značky `use assembly` jsou hledány ve stejném obsahu jako v předchozích bodech. Assembly nejsou v .NET odkazovány přímo z kódu, a tudíž pro ně není generován kód v C#. Zde se uplatní třída `AssemblyCache`. Tato třída poskytuje funkcionalitu spojenou s mechanismem assembly v .NET. Nachází se zde metoda `QueryAssemblyInfo()`, která získává informace o `assembly`. Metoda `QueryAssemblyInfo()` je volána pro všechny zahrnuté assembly, a pro každou assembly je uložena její cesta. Tím vznikne seznam cest všech zahrnutých assembly.

Získávání slov k doplnění pomocí Roslynu Jako další krok je volána metoda `GetCompletions()` a to ve třídě `CompletionSource`. V závislosti na typu rozpoznaného elementu, nacházejícího se na aktuální pozici, je zvolena jedna ze tří alternativ:

- doplnění pro inline kód,
- doplnění pro Spark výrazy (včetně boolean výrazů) nebo
- doplnění uvnitř Spark značky.

Pro všechny tři typy doplnění probíhají následující kroky.

Jako první je získán inline kód předcházející aktuální pozici kurzoru. Poté je volána metoda `GetCompletionItems()` třídy `CSharpCompletionSource`; při tomto volání je předán veškerý C# kód vygenerovaný ve výše popsaných bodech 1–6.

Třída `CSharpCompletionSource` poskytuje slova pro doplnění, která jsou získávána z knihovny Roslyn. Proces získání doplnění probíhá následovně:

- nejprve je získán objekt `DTE`, jenž představuje právě aktivní instanci Visual Studia, tedy instanci, ve které běží sezení doplňování slov.
- Z tohoto objektu je vytvořen objekt Roslynu `ISolution` představující aktivní solution.
- Ze stejného `DTE` objektu je dále získán aktivní projekt, a ten je přiřazen do objektu solution z předchozího bodu.
- Pokud byly předány nějaké cesty assembly souborů, reference na tyto assembly jsou přidány do právě vytvořeného projektu.
- Nakonec je vytvořen virtuální soubor třídy s názvem `CodeCompletion.cs`. Obsah této třídy je závislý na zdroji volání doplnění, nicméně třída vždy obsahuje předaný C# kód. Tato třída je pak přidána do projektu z předchozího bodu.

Do vygenerované třídy je však navíc přidána metoda `Main()`, v jejímž těle jsou zadefinovány patřičné lokální proměnné.

Poznámka 9.2 Načítání solution a projektu se z výkonových důvodů provádí pouze pokud je to nutné. Ve většině případů pouze dochází k aktualizaci již načteného projektu přidáním vygenerované třídy `CodeCompletion.cs`.

Budeme-li mít kurzor na pozici naznačené na obrázku 40, třída `CodeCompletion.cs` bude mít obsah ilustrovaný obrázkem 41.

Posledním krokem je získání samotných slov pro doplnění. Toto ilustruje kód na obrázku 42. Objekt `_completionSourceClass` na řádku 105 zde představuje vygenerovaný soubor třídy `CodeCompletion.cs` popsaný výše. Voláním metody `GetCompletionItemGroups()` jsou pak získána samotná slova pro doplnění. První parametr metody metody představuje pozici ve zdrojovém kódu, pro níž mají být vrácena slova k doplnění. Pokud pro předanou pozici existují nějaká doplnění (řádek 109),

```
13  <macro name ="myMacro" code="int">
14      ${code}
15  </macro>
16  <use namespace="System.Collections.Generic"/>
17  #string s ="string";
18  <viewdata items="IList[[object]]"/>
19  <global globVar="'hello'"/>
20  <var localVar ="5"/>
21  <for each="var item in items">
22
23  </for>
```

Obrázek 40: Příklad kódu Sparku

```
using System.Collections.Generic;
class Program
{
    public static IList<object> items;
    public static object globVar = "hello";
    public static void myMacro(int code){}
    public static void Main()
    {
        var localVar = 5;
        string s = "string";
        foreach (var item in items)
        {

        }
    }
}
```

Obrázek 41: Vygenerovaný C# kód pro Spark šablonu na obr. 40

jsou postupně procházeny kategorie slov pro doplnění (řádky 111–117), a pro každou kategorii jsou tato slova uložena do jejich výsledného seznamu (řádky 113–116).

Pokud bylo vyžádáno doplnění pro inline kód, získání tohoto doplnění probíhá podle právě popsaného algoritmu. Pokud je požadavek na doplnění Spark výrazu, je do těla metody `Main()` navíc vložen výraz "`var expression =`", a pozice ve vygenerovaném kódu je posunuta za tento výraz.

Získání slov k doplnění uvnitř Spark značky Jestliže se kurzor nachází uvnitř značky Spark a je vyžádán seznam doplnění, je volána metoda `GetTagCompletions()`. Této metodě je jako parametr předán obsah značky od jejího začátku, až po aktuální pozici kurzoru v okně editoru. Následně proběhne syntaktická analýza tohoto neúplného kódu značky. Tuto analýzu provádí parser, jenž je sestrojen z gramatiky `SparkIntellisenseGrammar`. Z parseru je vrácen řetězec obsahující očekávané ter-

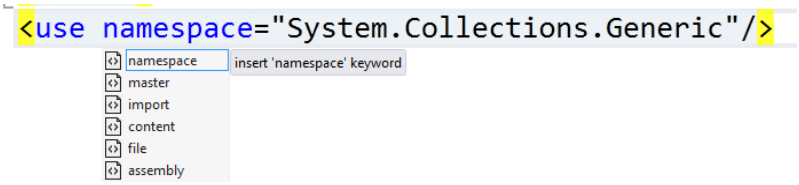
```
104         IEnumerable<CompletionItemGroup> completionGroups =
105             _completionSourceClass.GetCompletionItemGroups(position-1 + _completionTextLength,
106                                                            CompletionTriggerInfo
107                                                                .CreateInvokeCompletionTriggerInfo
108                                                                ());
109         if (completionGroups != null)
110         {
111             foreach (var completionItemGroup in completionGroups)
112             {
113                 foreach (var completionItem in completionItemGroup.Items)
114                 {
115                     completionItems.Add(completionItem);
116                 }
117             }
118         }
```

Obrázek 42: Získání slov pro doplnění pomocí Roslynu

minály, a to ve formě popisu chyby vniklé při parsování. Z tohoto řetězce jsou získány jednotlivé terminály, které jsou dvojího typu:

1. symboly validní uvnitř značky Sparku na dané pozici (kupř. `assembly`, `file`, `before`, `href`, ...) a
2. řetězce ve tvaru "nazev-teminalu-terminal".

První typ terminálů představuje validní slova k doplnění na dané pozici ve značce. Tato slova jsou hned vrácena a zobrazena. Obrázek 43 ukazuje příklad doplnění pro první typ terminálu.



Obrázek 43: Příklad doplnění přímo získaného ze syntaktického analyzátoru

Terminály určující druh slov k doplnění Druhý typ terminálů je ručně přidán do definice gramatiky `SparkIntellisenseGrammar` (viz podsekce 7.1.2) a slouží k identifikaci typu slov pro doplnění na dané pozici značky. V závislosti na typu značky a pozice kurzoru ve značce, mohou být syntaktickým analyzátorem intellisense vráceny řetězce, jejichž název je v prvním sloupci tabulky 9. Sloupec *značka* představuje název značky Sparku, pro kterou může být daný terminál vrácen. Sloupec *atribut* pak indikuje název atributu značky, pro jehož hodnotu může být vrácen daný terminál. Názvy atributů v hranatých závorkách značí, že daný atribut nabývá různých hodnot.

Slova k doplnění pro terminál `expression-terminal` jsou získána stejným způsobem, jako doplnění pro Spark výrazy.

terminál	atribut	značka
expression-terminal	`each` (za symbolem `in`)	`for`
	`key`	`cache`
	[název proměnné]	`set`
variableType-terminal	`each` (před názvem proměnné)	`for`
	`type`	`var`, `viewdata`,`default`, `global`, `def`
	`model`	`viewdata`
cache-expires-terminal	`expires`	`cache`
viewdata-parameter-terminal	`viewdata`	
use-content-terminal	`content`	`use`
namespace-terminal	`namespace`	`use`
use-file-terminal	`file`	`use`
include-href-terminal	`href`	`include`
render-segment-terminal	`render`	`segment`
partial-file-terminal	`partial`	`render`
use-assembly-terminal	`assembly`	`use`
variable-name-terminal	[hodnota atributu]	`set`

Tabulka 9: Terminály `SparkIntellisenseGrammar`

Pokud je syntaktickým analyzátorem vrácen terminál `variableType-terminal`, je vygenerován kód, podobný tomu, který je generován pro Spark výrazy, s tím rozdílem, že místo vložení výrazu "`var expression =`", je vložen výraz "`typeof(`", a za něj je posunut kurzor. Roslyn pak vrátí všechny datové typy validní v daném kontextu.

Jelikož v atributu `expires` značky `cache` jsou validní pouze návratové hodnoty členů knihovních tříd `System.DateTime` a `System.TimeSpan`, je při generování kódu pro neterminál `cache-expires-terminal` zahrnuta statická třída pojmenovaná `DateTimeWrapper`. Obsah této třídy znázorňuje výpis kódu 21. Do generované metody `Main()` je pak vložen výraz "`DateTimeWrapper.`", který umožňuje získat správná doplnění.

```
1 class DateTimeWrapper
2 {
3     public static DateTime DateTime;
4     public static TimeSpan TimeSpan;
5 }
```

Výpis 21: Třída použitá pro terminál cache-expires-terminal

Hodnoty pro doplnění jmen atributů značky `viewdata` jsou získány stejným způsobem, jako je popsáno v podsekci 9.2.1.

Pro terminál `use-content-terminal` jsou procházeny všechny značky v aktuálním souboru, a to včetně těch zahrnutých pomocí jiných značek. Pro všechny nalezené značky typu `content` jsou zjištěny názvy obsahu, který tyto značky definují, a tyto názvy jsou nabídnuty k doplnění.

Jestliže je z parseru vrácen terminál `namespace-terminal`, vygenerovaná třída obsahuje pouze naimportované assembly a výraz `use`. Doplnění jsou poté získána z pozice za tímto výrazem.

Je-li vrácen terminál `use-file-terminal`, jsou prohledány všechny soubory v aktivním projektu. Z těchto souborů jsou vybrány ty, které lze zahrnout do aktuálního souboru. Názvy těchto souborů jsou poté zobrazeny. Podobný sled akcí taktéž probíhá pro terminál `include-href-terminal`.

Návrat terminálu `partial-file-terminal` pak znamená prohledání všech validních lokací vzhledem k lokaci právě editovaného souboru, a jejich prohledání na partial soubory. Takto nalezené názvy partial souborů jsou pak vráceny a nabídnuty k doplnění.

Pro terminál `use-assembly-terminál` je volána metoda `QueryAssemblyInfo()` ze třídy `AssemblyCache`. V závislosti na pozici v atributu `assembly` je pak nabídnuta patřičná hodnota—plně kvalifikované jméno assembly, veřejný klíč, verze atp.

A konečně, pokud parser vrátí terminál `variable-name-terinal`, dojde k vygenerování C# kódu všech aktuálně platných proměnných (viz body 1 a 3 v podsekci 9.2.2). Poté je na vygenerovaném kódu provedena syntaktická a sémantická analýza za asistence Roslynu (metoda `GetVariableNames()` ve třídě `VariableRetriever`). Výsledkem těchto analýz jsou názvy všech proměnných, které jsou platné v aktuálním kontextu. Tyto názvy jsou pak zobrazeny k doplnění.

10 Závěr

Za asistence knihoven a řešení třetích stran jsem naimplementoval podporu zvýrazňování syntaxe a doplňování slov ve Spark View Engine šablonách. Výsledné řešení do značné míry urychluje práci se zdrojovým kódem psaným za užití Spark View Engine. Programátor píšící zdrojový kód ve Sparku teď díky zvýrazněné syntaxi snadno a rychle rozpozná, o jaký úsek kódu šablony se jedná. Navíc jsou při psaní kódu automaticky nabízena slova, která jsou jak syntakticky, tak sémanticky validní v aktuálním kontextu, a programátor tak nemusí zdlouhavě vypisovat všechny znaky cílového slova. Snad každý, kdo vyvíjí webové stránky pomocí Sparku, rozpozná přínos mnou naimplementovaného doplňku pro Visual Studio 2012. Instalace je snadná, použití je intuitivní a do jisté míry flexibilní, jelikož je uživateli umožněno provést vlastní definici zvýrazňování syntaxe.

Jako téměř každý (ne-li každý) softwarový produkt, ani tento není dokonalý, a má určité drobné nedostatky. Za hlavní nedostatek považuji skutečnost, že některé části kódu, které jsou validním kódem šablon Spark View Engine, se zobrazují jako chybové. Důvod přítomnosti tohoto nedostatku je detailně popsán v sekci 8.

Další nedostatek, který se mi přes veškerá má snažení nepodařilo odstranit, je způsob, jakým je potvrzován výběr nabízených slov ve Spark XML značkách. Při práci s uživatelským rozhraním intellisense je běžné, že slovo, které má být vloženo do editoru, se potvrzuje stisknutím klávesy *ENTER* nebo *TAB*. V případě Spark XML značek, je však potvrzování realizováno pouze stiskem klávesy *TAB*. Stisk klávesy *ENTER* povede ke zrušení okna intellisense a odřádkování, namísto očekávané akce potvrzení doplnění. Důvodem tohoto nestandardního chování je opět způsob zadefinování obsahu Sparku, který je popsaný v sekci 8.

Druhým nedostatkem je občasná pomalost získávání slov pro doplnění. Toto zpomalení je způsobeno ve voláních knihovny Roslyn. Jelikož Roslyn zatím není ve své finální verzi, lze předpokládat, že v dalších verzích této knihovny dojde ze strany společnosti Microsoft ke zrychlení problematických úseků kódu.

I přes mou snahu vytvořit bezchybný kód, se s velkou pravděpodobností v některých jeho částech vyskytují různé malé chyby a ”bugy”. Jejich přítomnost by se však neměla významně podepsat na použitelnosti celého řešení, a v případě nalezení, by jejich odstranění nemělo být nijak složité či časově náročné.

Naprogramovaný doplněk by se nepochybně dalo rozšířit o další funkcionalitu. Především by bylo užitečné mít k dispozici detekci sémantických chyb. V současném řešení například není žádná typová kontrola, a tím pádem je snadné se dopustit chyb v typování proměnných. Další typ chyb, které nejsou detekovány, jsou chyby v párování značek. Také je možno se odkazovat na neexistující soubory nebo značky, aniž by na takové chyby byly detekovány a zobrazovány editorem.

V neposlední řadě by se dalo vyžít dalších možností rozšiřování Visual Studia. Například při vytváření projektu Spark View Engine, je vždy nutno vytvořit projekt typu MVC, a následně provést dodatečné úpravy a nastavení, které umožní použít Spark View Engine. Přidání nového typu projektu pro Spark by odstranilo nutnost provádět dodatečné konfigurace při vytváření projektu, používajícího Spark View Engine.

Další vlastností, jejíž přidání by dále zlepšilo a zefektivnilo práci se Sparkem, je pod-

pora tzv. *rychlé nápovědy* (angl. *quick info*). Po najetí kurzoru myši nad nějaký symbol zapsaný v editoru, by byly uživateli poskytnuty dodatečné informace o tomto slově. Takováto dodatečná funkcionalita by mohla být užitečná v šablonách Sparku. Především by bylo užitečné mít k dispozici rychlou nápovědu pro symboly převzaté z knihoven .NET (např. názvy metod a objektů).

11 Reference

[Aho et al.(2006)Aho, Sethi,, Ullman] AHO, A. V. – SETHI, R. – ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools*. Vyd. 2. Pearson Education, Inc, 2006. ISBN 0-321-48681-1.

[Baxter(2010)] BAXTER, I. *What is the difference between LR, SLR, and LALR parsers?* [online]. 10 2010. [cit. 23. 4. 2013]. Dostupné z: http://stackoverflow.com/questions/2676144/what-is-the-difference-between-lr-slr-and-lalr-parsers.

[Beneš(1999)] BENEš, M. *Překladače*. VŠB-TU Ostrava, 1999.

[Chang(2009)] CHANG, R. *Mid-Rule Actions and The Dangling Else* [online]. 10 2009. [cit. 23. 4. 2013]. Dostupné z: http://www.csee.umbc.edu/~chang/cs431/dangling-else.shtml.

[DeJardin(2008)] DEJARDIN, L. *Spark View Engine* [online]. 7 2008. [cit. 28. 4. 2013]. Dostupné z: http://sparkviewengine.com/documentation.

[Habiballa(2005)] HABIBALLA, H. *Regulární a bezkontextové jazyky II*. Ostravská univerzita v Ostravě, 2005. Dostupné z: www1.osu.cz/home/habibal/publ/rabj2.pdf.

[Ivantsov(2008)] IVANTSOV, R. *Irony - .NET Compiler Construction Kit* [online]. 1 2008. [cit. 24. 4. 2013]. Dostupné z: http://www.codeproject.com/Articles/22650/Irony-NET-Compiler-Construction-Kit.

[Jančar(2007)] JANčAR, P. *Teoretická Informatika*. Vyd. 1. Ediční středisko VŠB-TUO, 2007. ISBN 978-80-248-1487-2.

[Microsoft(2010)] MICROSOFT. *C. Grammar* [online]. 2010. [cit. 27. 4. 2013]. Dostupné z: http://msdn.microsoft.com/en-us/library/aa664812(v=vs.71).aspx.

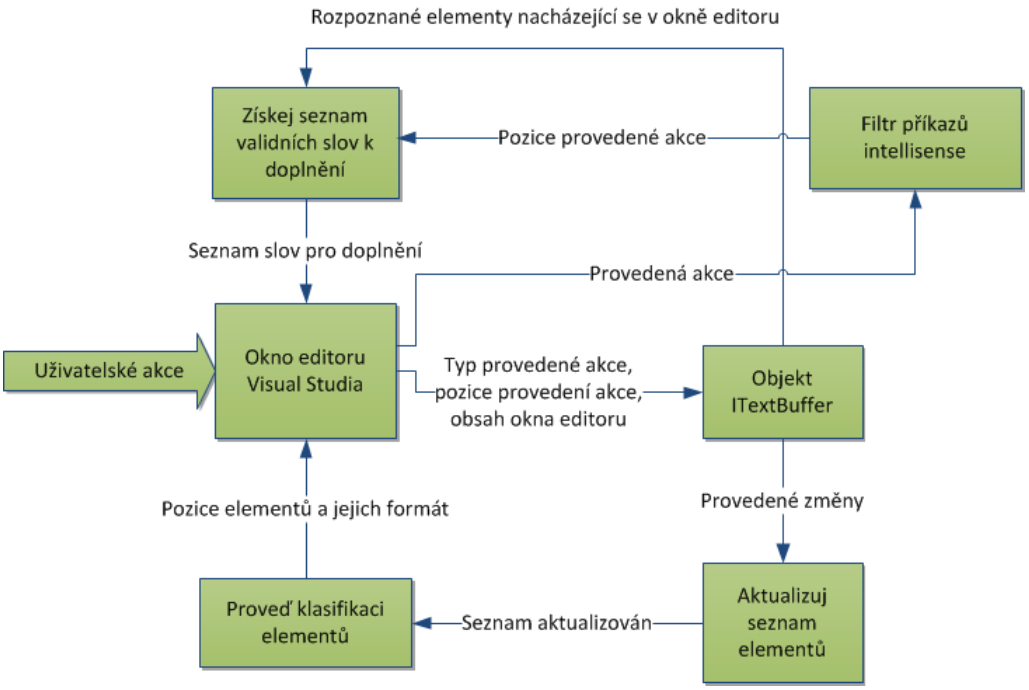
[Microsoft(2013a)] MICROSOFT. *Iterators (C# and Visual Basic)* [online]. 2013a. [cit. 25. 4. 2013]. Dostupné z: http://msdn.microsoft.com/en-us/library/dscyy5s0(v=vs.110).aspx.

[Microsoft(2013b)] MICROSOFT. *Managed Extensibility Framework (MEF)* [online]. 2013b. [cit. 26. 4. 2013]. Dostupné z: http://msdn.microsoft.com/en-us/library/dd460648.aspx.

[Microsoft(2013c)] MICROSOFT. *PackageManifest Element (Root Element, VSX Schema)* [online]. 2013c. [cit. 26. 4. 2013]. Dostupné z: http://msdn.microsoft.com/en-us/library/vstudio/dd393754.aspx.

[Sloane – Verity(2008)Sloane, Verity] SLOANE, T. – VERITY, D. *COMP332 Lecture Notes 5: Shift-reduce parsing (Scott 2.2.6)* [online]. 8 2008. [cit. 20. 4. 2013]. Dostupné z: `http://comp.mq.edu.au/units/Archive2008S2/comp332/lectures/5%20Shift-reduce%20parsing.htm`.

A Schématické znázornění provázání komponent



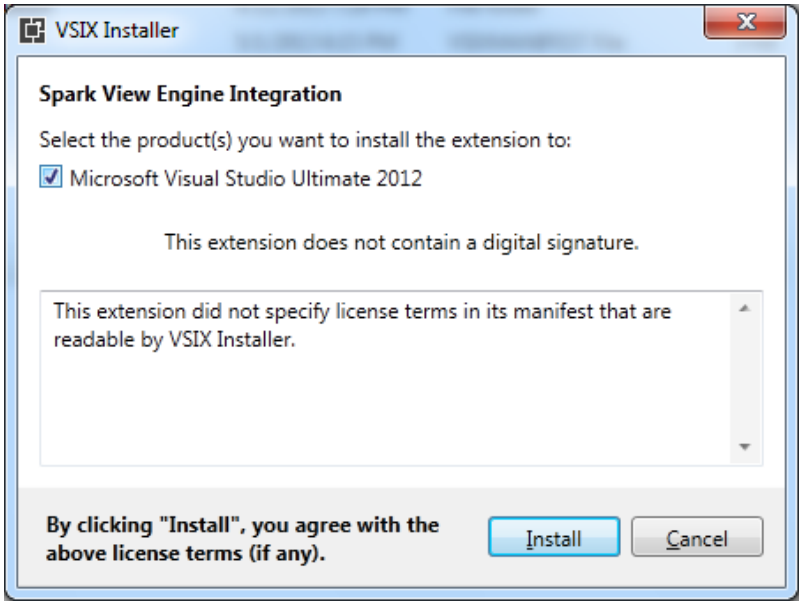
Obrázek 44: Schématické znázornění provázání komponent zodpovědných za zvýrazňování syntaxe a nabízení slov k doplnění

B Návod k instalaci a použití

Instalace

Pro úspěšnou instalaci rozšíření je třeba provést následující kroky:

1. Nainstalovat Visual Studio 2012 verze *Professional* a vyšší. Při instalaci je nutno při výběru instalovaných komponent zatrhnout položku *Web Developer Tools*.
2. Nainstalovat .NET Framework 4.5. Instalační soubor frameworku .NET 4.5 se nachází na přiloženém DVD ve složce *install/dotnetfx45_full_x86_x64.exe*.
3. Nainstalovat Microsoft Roslyn. Instalační soubor se také nachází na přiloženém DVD, a to ve složce *install/RoslynSetup.exe*.
4. Pokud je požadavek zkompilovat přiložené zdrojové kódy, je nutno mít nainstalované Visual Studio 2012 SDK. Instalační soubor SDK se nachází přiloženém DVD ve složce *install/vssdk_full.exe*.
5. Nainstalovat samotný doplněk. Ten se nachází na přiloženém DVD ve složce *install/SparkIntegration.vsix*. Spuštění instalace se provádí dvojklikem na .vsix soubor. Poté se zobrazí okno, které ukazuje obrázek 45. Následně stačí kliknout na tlačítko *Install* a začne instalace.



Obrázek 45: Instalace VSIX doplňku

Použití

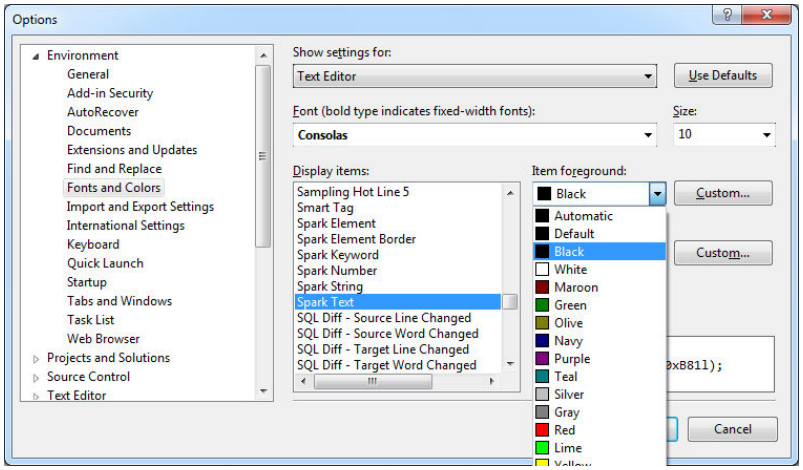
Pro použití doplňku je nutné mít ve Visual Studiu načtený Spark projekt. Tento projekt lze:

1. vytvořit od začátku podle návodu na `http://sparkviewengine.com/usage/project-from-scratch`, nebo
2. otevřít poskytnutou solution s již vytvořeným Spark projektem. Tato solution se nachází na přiloženém DVD, a to ve složce *SparkExperiments_2012/SparkExperiments.sln*.

Pokud uživatel používá *bílé* barevné schéma, je navíc potřeba nastavit černou barvu písma pro typ *Spark Text*. Postup je následovný:

1. v menu Visual Studia kliknout na **tools** a vybrat položku **options**.
2. V nastavení **Environment** vybrat **Fonts and Colors**.
3. V podokně nadepsaném **Display Items** vyhledat položku **Spark Text**.
4. V comboboxu nadepsaném **Item Foreground** zvolit barvu **Black**.
5. Potvrdit nastavení stisknutím tlačítka **OK**.

Pro lepší názornost je na obrázku 46 snímek obrazovky ilustrující tento postup.



Obrázek 46: Nastavení barvy pro *Spark Text* v menu Visual Studia

Veškerá potřebná nastavení jsou provedena. Teď již stačí otevřít nějaký soubor Spark šablony z načteného Spark projektu, a začít editovat zdrojový kód.