VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

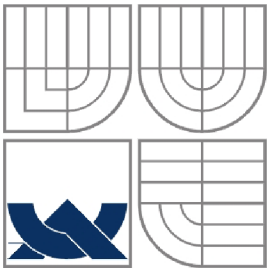
MODERNÍ EDITOR ZDROJOVÉHO KÓDU PRO ZADANÝ JAZYK

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

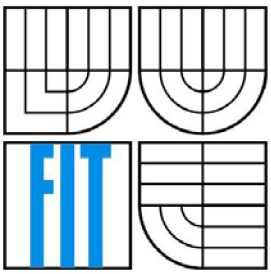
AUTOR PRÁCE
AUTHOR

PAVOL SRNA

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

MODERNÍ EDITOR ZDROJOVÉHO KÓDU PRO ZADANÝ JAZYK

ADVANCED SOURCE CODE EDITOR FOR A GIVEN LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE
AUTHOR

PAVOL SRNA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MILOŠ EYSSELT, CSc.

BRNO 2008

Zadanie bakalárskej práce

1. Seznamte se s rysy moderních editorů zdrojových kódů, prostudujte možnosti rozšíření editoru některého vývojového prostředí (např. Eclipse) pro nový programovací jazyk.

2. Seznamte se se zadaným jazykem, jeho gramatikou a vlastnostmi.

3. Pro vybrané vývojové prostředí navrhněte a implementujte editor daného jazyka, který bude mít hlavní rysy moderních editorů (např. syntax coloring, syntax checking, code completion, nápověda/tipy podle kontextu).

4. Rozšíření musí být přeložitelné pod systémy typu Windows i UNIX a musí být licencováno GNU Lesser General Public License (LGPL) verze 2.1.

5. Záležitosti spojené s textovými editory konzultujte s Ing. Petrem Gotthardem, ANF DATA, Brno.

6. Zhodnoťte výsledky své práce a diskutujte možnosti dalšího rozšíření projektu.

Licenčná zmluva

Licenčná zmluva je uložená v archíve Fakulty informačných technológií Vysokého učení technického v Brně.

Abstrakt

Hlavným cieľom tejto práce je zoznámiť sa s aplikačným rozhraním vývojového prostredia eclipse a vhodne rozšíriť jeho editor zdrojového kódu tak, aby pre jazyk SIT mal rysy moderných textových editorov. Práca ukazuje vytvorenie parsra a lexra z gramatiky ANTLR. Vysvetľuje spôsob písania zásuvných modulov do platformy eclipse so zameraním na editor zdrojového kódu. Implementovaný plugin editoru má podporu zvýrazňovania a kontroly syntaxe, automatického dopĺňania kódu a nápovedy podľa kontextu.

Kľúčové slova

eclipse, plugin, editor zdrojového kódu, parser, lexer, ANTLR

Abstract

Aim of this BSc Thesis is to acquaint with application programming interface of eclipse development environment and to extend appropriately its source code editor to obtain the characteristics of the advanced text editors in the language SIT. The Thesis describes the creation of the parser and lexer from ANTLR grammar. It explains the way of writing plugins to the eclipse platform by focusing on the source code editor. The implemented editor plugin has the support of syntax highlighting, a syntax checking, an automatic code completion and a context information.

Keywords

eclipse, plugin, source code editor, parser, lexer, ANTLR

Citácia

Pavol Srna: Moderní editor zdrojového kódu pro zadaný jazyk. Brno, 2008, bakalářská práce, FIT VUT v Brně.

Moderní editor zdrojového kódu pro zadaný jazyk

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením Ing. Miloša Eysselta, CSc.
Ďalšie informácie mi poskytol Ing. Petr Gotthard.
Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

……
Pavol Srna
30. 4. 2008

Poďakovanie

Rád by som poďakoval vedúcemu práce Ing. Milošovi Eysseltovi, CSc. za odbornú pomoc. Taktiež by som chcel poďakovať Ing. Petrovi Gotthardovi za poskytnutie informácií a čas venovaný konzultáciám. Moje poďakovanie patrí aj priateľke Zuzke za podporu a gramatickú korekciu textu.

Obsah

Úvod

Sme svedkami masívnej produkcie softvéru. Zvýrazňovanie a kontrola syntaxe, automatické dopĺňanie kódu, nápoveda podľa kontextu a ďalšie funkcie pre podporu editácie zdrojového kódu veľmi zefektívňujú a zrýchľujú jeho zaobstaranie. Rýchlosť a efektivita získania kódu má nasledovne priamy vplyv na cenu vyvíjanej aplikácie. Tento fenomén tlačí na výrobcov vývojových prostredí a ženie ich k tvorbe dokonalejších nástrojov, ktoré majú spomínané rysy. Neproduktivita a časová náročnosť stavať ten istý základ editora viackrát pre niekoľko druhov aplikácií vedie k pojmu *Tool Integration*. Tento pojem znamená integráciu viacerých nástrojov do jedného softvéru, ktorú ocenia aj užívatelia, ktorí sú zvyknutí pracovať s rozličnými technológiami v jednom vývojovom prostredí. Eclipse je platforma, ktorá umožňuje formou zásuvných modulov pridávať integrovanému vývojovému prostrediu nové funkcionality. Štruktúra tejto platformy dovoľuje vývojárovi plugin-u[1] používať už implementovaný základ IDE[2], ako napríklad okno workbench s jednotlivými pohľadmi, open-close-save model a mnoho ďalších, čo mu dovoľuje sústrediť sa na špecifické vlastnosti zásuvného modulu, ktorý vytvára.

Spoločnosť ANF DATA (Siemens AG Österreich) v rámci aplikovaného výskumu vyvíja moderné vývojové prostredie pre špeciálne jazyky používané vo firme Siemens. Jedným z týchto jazykov je aj jazyk SIT. Cieľom tejto bakalárskej práce je zoznámiť sa s aplikačným rozhraním eclipse, a vhodne rozšíriť jeho editor zdrojového kódu tak, aby pre jazyk SIT mal spomínané rysy moderných textových editorov. Problematika, ktorou sa táto práca zaoberá, je aktuálna a doposiaľ k nej bolo vydaných málo knižných publikácií, preto väčšina zdrojov pochádza z internetu. V práci používam anglické pojmy, ktoré skloňujem, pretože nie sú zaužívané ich ustálené slovenské ekvivalenty.

Dokument je logicky členený do niekoľkých kapitol. V teoretickej časti je čitateľ oboznámený s BNF[3] a EBNF[4]. Taktiež, je mu vysvetlená funkčnosť parsra a lexra. Tieto informácie sú potrebné na pochopenie textu v nasledujúcich kapitolách. V kapitole *2* venujem pozornosť práci s nástrojom ANTLR, vysvetľujem techniky tvorby gramatiky jazyka a opisujem časté problémy pri práci s týmto nástrojom. Poznatky z tejto časti sú potrebné v kapitole *4*, v ktorej sa na nich spätne odkazujem. Čitateľ tak dostáva potrebný teoretický základ k jednoduchšiemu pochopeniu návrhu a implementácie. V kapitole *3* rozoberám platformu eclipse, pričom sa špeciálne venujem mechanizmu zásuvných modulov. Návrh a implementáciu zásuvného modulu vysvetľujem v kapitole *4*. V závere sú zhodnotené dosiahnuté ciele a diskutované možnosti ďalšieho pokračovania práce.

---

[1] Plugin - zásuvný modul.

[2] Integrated Development Environment - integrované vývojové prostredie.

[3] BNF - backus naurova forma.

[4] EBNF - rozšírená backus naurova forma, z anglického: Extended Backus Naur Form.

1 Teoretický úvod

1.1 BNF

Backus Naurova Forma je metasyntax používaná na vyjadrenie bezkontextovej gramatiky. Je to spôsob popisu formálneho jazyka. John Backus a Peter Naur vyvinuli bezkontextovú gramatiku, aby mohli definovať syntax programovacieho jazyka na základe dvoch množín pravidiel: t.j. lexikálne a syntaktické pravidlá [1].

Princíp BNF je podobný matematickej hre: začínate štartovacím symbolom a potom nasleduje súbor pravidiel, za použitím ktorých môžete štartovací symbol nahradiť. Jazyk definovaný BNF je množinou všetkých reťazcov, ktoré vyprodukujete použitím istých pravidiel. Tieto pravidlá sa nazývajú produkčné pravidlá a vyzerajú nasledovne:

```
symbol := alternative1 | alternative2 ...                    (1)
```

Produkčné pravidlo jednoducho vyjadruje, že symbol na ľavej strane znaku := musí byť nahradený jednou z alternatív na pravej strane. Alternatíva môže byť buď symbolom alebo TERMINÁLOM. Pre terminály neexistujú produkčné pravidlá, preto ukončujú produkčný proces. Symboly sa často nazývajú neterminály (NON-TERMINALS) [2]. Príklad BNF je ukázaný v (2).

```
START := '-' FractionalNumber | FractionalNumber
FractionalNumber := DigitList | DigitList '.' DigitList
DigitList := Digit | Digit DigitList
Digit := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' |
'8' | '9'                                                    (2)
```

Správne vety jazyka popísaného touto gramatikou sú čísla, ktoré môžu byť desatinné alebo záporné desatinné. Ak chceme vygenerovať číslo z tejto gramatiky, začneme v štartovacom symbole START. Uvažujme, že chceme číslo, ktoré je záporné a desatinné, napríklad číslo -3.14.

```
START                                                        (3)
```

Výraz (3) nahradíme jednou z jeho alternatív, v našom prípade '-' FractionalNumber.

```
'-' FractionalNumber                                         (4)
```

'-' je terminálom, takže v ďalšom kroku nahradíme len symbol FractionalNumber. Chceme vygenerovať číslo -3.14, takže jediná prípustná alternatíva teda bude DigitList '.' DigitList. Dostávame teda:

```
'-' DigitList '.' DigitList                                    (5)
```

V ďalších krokoch postupujeme podobným spôsobom. Vždy symbol nahradíme podľa produkčného pravidla jednou z jeho alternatív. Dospejeme k nasledujúcemu záveru:

```
'-' Digit '.' DigitList
'-' '3' '.' DigitList
'-' '3' '.' Digit DigitList
'-' '3' '.' Digit Digit
'-' '3' '.' '1' Digit
'-' '3' '.' '1' '4'                                            (6)
```

Výsledkom je nami hľadané číslo -3.14 [2].

V (2) som musel použiť v pravidle DigitList rekurziu. Týmto sa stáva gramatika BNF menej prehľadná. EBNF rieši tento problém elegantným spôsobom, keď rozširuje BNF o nasledujúce operátory [2]:

? Symbol alebo skupina symbolov uzavretá v zátvorkách naľavo od tohto operátora je voliteľná. Môže sa vyskytnúť maximálne raz.

* Symbol alebo skupina symbolov uzavretá v zátvorkách naľavo od tohto operátora sa môže opakovať mnohonásobne, prípadne sa môže vynechať.

+ Symbol alebo skupina symbolov uzavretá v zátvorkách naľavo od tohto operátora sa musí vyskytnúť minimálne raz, alebo niekoľko krát.

Príklad gramatiky BNF uvedený v (2) vyzerá v EBNF nasledovne:

```
START := '-'? Digit+ ('.' Digit+)?
Digit := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' |
'8' | '9'                                                      (7)
```

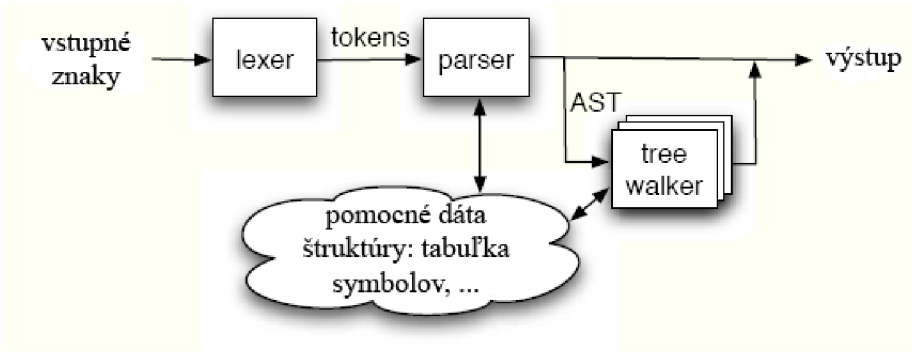
Pomocou EBNF sa dajú definovať gramatiky programovacích jazykov. To má radu výhod, ako napríklad, že syntax daného jazyka je jednoznačne určená. Taktiež je jednoduché z gramatiky EBNF

vygenerovať parser a lexer. Ja použijem v tejto bakalárskej práci nástroj ANTLR, pomocou ktorého vygenerujem z gramatiky EBNF lexer a parser implementovaný v jazyku JAVA.

1.2 Lexer a parser

Lexikálna analýza je proces, pri ktorom prevádzame vstupnú sekvenciu znakov na sekvenciu tokenov. Programy, ktoré vykonávajú lexikálnu analýzu, sa nazývajú lexre. Token je kategorizovaný blok textu, ktorý ma pridelený význam [10].

Syntaktická analýza je proces rozboru sekvencie tokenov na určenie gramatickej štruktúry v súvislosti s danou formálnou gramatikou. Parser je program, ktorý vykonáva syntaktickú analýzu, často je súčasťou interpreta alebo prekladača [11].



Obrázok 1 [3]. Ilustrácia práce lexra a parsra.[5]

LL parser je postavený na metóde zhora dolu. Vstup číta z **L**ava do prava a vytvára **L**avú deriváciu. Parsrovanie zhora dolu je stratégia analýzy neznámych vzťahov dát, takým spôsobom, že zo vstupu vytvárame štruktúru parsrového stromu (parse tree), a zvažujeme, či získaná štruktúra zodpovedá predpokladu tvoreného z pravidiel gramatiky. Jednoduchšie povedané, LL parser začne v štartovnom produkčnom pravidle, a pokúša sa zhodnúť sa s jednou z jeho alternatív. Istý problém predstavuje gramatika, ktorá obsahuje ľavú rekurziu. Pretože jednoduché implementácie LL parsrov nedokážu úspešne ukončiť parsrovanie. Toto je možné obísť za použitia backtrackingu, ale takýto parser má už exponenciálnu časovú zložitosť.

LL parser nazveme LL(k) parsrom, ak pri parsrovaní "vidí" na vzdialenosť k tokenov. Ak pre niektorú gramatiku exituje takýto parser a dokáže parsrovať vstup bez použitia backtrackingu, potom sa takáto gramatika nazýva LL(k) gramatikou. Veľmi obľúbenými gramatikami sú gramatiky LL(1), pretože takémuto parsru stačí len nasledujúci token na jednoznačné rozhodnutie [12]. Uvediem príklad LL(3) gramatiky. Majme LL(3) gramatiku zapísanú pomocou EBNF:

[5] Niektoré z pojmov sú v anglickom jazyku, pretože nie je ustálený ich slovenský ekvivalent.

```
declaration    :=  'int' ID '=' INT ';'     //napr. "int x = 3;"
               |   'int' ID ';'             //napr. "int x;"          (8)
```

V tomto prípade potrebuje parser 3 tokeny na jednoznačné určenie, či sa jedná o deklaráciu s inicializáciou alebo len o deklaráciu. Kapitola *The Nature of computer languages* [3] predkladá zaujímavý pohľad na túto problematiku.

2 ANTLR

ANTLR[6] je program, ktorý zo súboru obsahujúceho popis gramatiky dokáže vygenerovať prekladač, kompilátor, interpret implementovaný v jednom z podporovaných cieľových programovacích jazykoch. Tento nástroj používa LL(*) parsrovaciu stratégiu. Spôsob, akým táto stratégia funguje, je veľmi dobre popísaný v kapitole *Understanding Predicated-LL(*) Grammars* [3]. V nasledujúcich odstavcoch sa budem zaoberať povahou práce s týmto nástrojom, ktorá bude potrebná na pochopenie textu v kapitole 4.

2.1 Gramatika

ANTLR používa takzvanú kombinovanú gramatiku, v ktorej spája pravidlá pre lexer s pravidlami pre parser. Aby bolo možné od seba tieto princípy odlíšiť, tak sa lexikálne pravidlá píšu s veľkým začiatočným písmenom a parsrovacie s malým. Všetky gramatiky majú jednu a tú istú základnú štruktúru:

```
/** This is a document comment */
grammarType grammar name;
«optionsSpec»
«tokensSpec»
«attributeScopes»
«actions»
/** doc comment */
rule1 : ... | ... | ... ;
rule2 : ... | ... | ... ;                                          (9)
```

[6] ANTLR - ANother Tool for Language Recognition.

Poradie jednotlivých sekcii musí byť zachované ako je ukázané vyššie, pričom definície pravidiel sa nachádzajú vždy až na konci za všetkými sekciami.

Z gramatiky *T*, ANTLR vygeneruje rozoznávač (recognizer) s menom identickým jeho úlohe. Ak používame ako cieľový jazyk JAVU, ANTLR vytvorí TLexer.java a TParser.java. ANTLR vždy vygeneruje súbor obsahujúci zoznam použitých tokenov. Tento súbor môžu použiť iné gramatiky, aby boli synchronizované s hlavnou gramatikou, v tomto prípade gramatikou T [3].

Nasledujúci príklad jednoduchej (kombinovanej) gramatiky ukazuje základné črty gramatík ANTLR:

```
grammar T;
options {
    language=Java;
}
@members {
    String s;
}

r : ID '#' {s = $ID.text; System.out.println("found "+s);};
ID: 'a'..'z' + ;
WS: (' ' |'\n' |'\r' )+ {skip();} ; //ignore whitespace          (10)
```

Gramatika *T* prijíma všetky identifikátory, za ktorými nasleduje kríž. V akcii vytvorí lokálnu premennú, do ktorej priradí text identifikátora a vypíše ho na štandardný výstup. V príklade môžeme pozorovať ešte jednu akciu, tento krát sa jedná o akciu lexra, ktorá informuje lexer, aby preskočil všetky tokeny typu WS a pokračoval v hľadaní ďalšieho tokenu.

Nasledujúci hlavný program ukazuje ako preloženú gramatiku použiť, aby parsrovala dáta zo štandardného vstupu:

```
import org.antlr.runtime.*;

public class Test {
public static void main(String[] args) throws Exception {
    ANTLRInputStream input = new ANTLRInputStream(System.in);
    TLexer lexer = new TLexer(input);
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    TParser parser = new TParser(tokens);
    parser.r();
}
}                                                                (11)
```

Parser začne pracovať, ak kontrolný program *Test* vyvolá jednu z metód generovaných z pravidla gramatiky. V tomto prípade *parser.r()* vyvolá pravidlo **r** ako štartovný symbol [3].

Sekcia «optionsSpec» uvedená v (9) musí mať takúto štruktúru:

```
options {
    key1 = value1 ;
    key2 = value2 ;
    ...
}                                                    (12)
```

Oddiel options gramatiky ANTLR zobrazený v (12) umožňuje určiť skupinu priradení kľúč = hodnota, ktoré upravujú spôsob, akým ANTLR generuje kód. Tieto možnosti ovplyvňujú globálne všetky prvky obsiahnuté v gramatike, pokiaľ ich v niektorom z pravidiel nepredefinujeme. V nasledujúcej časti rozoberiem len úplne základne možnosti nastavenia gramatiky ANTLR, ktoré sa najčastejšie používajú a ktoré sú potrebné k pochopeniu textu v ďalších kapitolách. Všetky možnosti sú dopodrobna vysvetlené v [3].

- **Backtrack** - ak je nastavené na **true**, indikuje, že ak zlyhá analýza LL(*), ANTLR má použiť backtracking na získanie deterministického rozhodnutia. Často sa používa s možnosťou memoize. Základné nastavenie je **false** [3].

- **Memoize** - pamätá si čiastkové parsrovacie výsledky, čím zabezpečí, že počas backtrackingu sa nebude parsrovať ten istý vstup v rovnakom pravidle viac ako raz. Tento prístup poskytuje lineárnu rýchlosť parsrovania na úkor nelineárnej pamäte. Defaultne je nastavené na **false** [3].

Špecifikácie tokenov (Tokens Specifikation) sa používajú v gramatike ANTLR na zavedenenie nového tokenu, alebo priradenie lepšieho názvu literálu tokenu. Oddiel «tokensSpec» uvedený v (9) má nasledujúcu formu:

```
tokens {
    token-name1 ;
    token-name2 = 'string-literal' ;
    ...
}                                                    (13)
```

Tento prístup umožňuje vytvoriť imaginárne tokeny, ktorých názov ešte nie je asociovaný so žiadnymi vstupnými znakmi [3].

2.2 Lexikálne pravidlá

Lexikálne pravidlá sa odlišujú od parsrovacích v niekoľkých smeroch, aj napriek tomu, že ich syntax je takmer identická. Azda každé lexikálne pravidlo je aj názov tokenu a musí začínať s veľkým začiatočným písmenom. V porovnaní s gramatikou parsru, gramatika lexra neobsahuje žiadne štartovné začiatočné pravidlo. Táto gramatika obsahuje len zoznam tokenov, ktoré môže lexer identifikovať kedykoľvek vo vstupnom prúde [3].

Často je veľmi výhodné rozdeliť veľké pravidlá do menších, zrozumiteľnejších. Pretože ANTLR predpokladá, že všetky lexikálne pravidlá sú platné tokeny, je nutné pred takzvané "pomocné pravidlá" uviesť kľúčové slovo **fragment**. Token, ktorý vznikne z takéhoto pravidla, nie je poslaný ďalej k parsru. Nasledujúce pravidlo určuje syntax unicode znaku za použitia fragment pravidla, ktoré identifikuje aktuálne hexadecimálne číslice [3].

```
UNICODE_CHAR
: '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
;
fragment
HEX_DIGIT
: '0'..'9' |'a'..'f' |'A'..'F'
;                                                    (14)
```

Jednou z najťažších lexikálnych záležitostí je vyrovnať sa s paradoxom, že parser musí ignorovať komentáre a mnoho krát aj biele znaky. Avšak, v tom istom čase musí poskytnúť prístup k takýmto tokenom počas prekladu. K vyriešeniu tohto problému, dovoľuje ANTLR, aby každý token objekt mohol existovať na rozličnom kanále (channel) podobným spôsobom, ako existujú rozličné rádiové stanice. Parser môže "naladiť" jeden z kanálov, a teda tokeny, ktoré nie sú na tomto kanále (mimo-kanálové tokeny) budú ignorované. Objekty, ktoré implementujú rozhranie *TokenStream*, ako napríklad *CommonTokenStream*, poskytujú prístup k takýmto mimo-kanálovým tokenom z dôvodu použitia v akciách (**actions**). Nasledujúca definícia tokenu identifikuje biele znaky a za použitia akcie určí, že takýto token má byť umiestnený na Štandardný skrytý kanál (HIDDEN_CHANNEL) [3]:

```
WS : (' ' |'\t' |'\r' |'\n' )+ {$channel=HIDDEN;};   (15)
```

2.2.1 Ako emitovať viac ako jeden token

Informácie, ktoré som získal z [3], sú podané v tejto kapitole. Lexikálne pravidlo môže donútiť lexer, aby emitoval viac ako jeden token počas vyvolania pravidla manuálnym zavolaním metódy *emit()*. Tento spôsob rieši veľmi zložité problémy, napríklad vkladanie 'imaginárnych tokenov'[7]. Najlepším príkladom je lexikálna analýza pythonu. Pretože python používa odsadzovanie na indikovanie blokov kódu, neexistujú žiadne explicitné *pre* a *za* tokeny, ktoré by zlúčili príkazy v rámci jedného bloku. Napríklad, v nasledujúcom kóde pythonu je príkaz **if** a volanie funkcie **g()** na rovnakej vonkajšej úrovni. Príkaz **print** a volanie metódy **f()** sú na spoločnej vnútornej, vnorenej úrovni.

```
if foo:
  print "foo is true"
  f()
g()                                                    (16)
```

Bez *pre* a *za* tokenov predstavuje parsrovanie takéhoto vstupu pre parser problém pri snahe o zlúčenie jednotlivých príkazov. Lexer potrebuje emitovať imaginárne **INDENT**[8] a **DEDENT**[9] tokeny pre vyznačenie začiatku a konca kódu bloku. Sekvencia tokenov bude nasledujúca:

```
IF ID : NL INDENT PRINT STRINGLITERAL NL ID ( ) NL DEDENT
ID ( ) NL                                              (17)
```

Parsrovacie pravidlo na identifikovanie nasledujúceho bloku kódu by vyzeralo nasledovne:

```
block : INDENT statement+ DEDENT ;                     (18)
```

Lexer môže emitovať INDENT token, keď vidí biely znak, ktorý je viac zanorený ako biele znaky pred predchádzajúcim príkazom. Avšak, neexistuje vstupný znak, ktorý by vyvolal DEDENT token. V podstate musí lexer emitovať DEDENT token, keď vidí menšie zanorenie ako v predchádzajúcom príkaze. Dokonca musí emitovať niekoľko DEDENT tokenov v závislosti na tom, o koľko viac sa vynoril od predchádzajúceho príkazu. Pravidlo INDENT by mohlo vyzerať nasledovne:

```
INDENT
: // turn on rule only if at left edge
  {getCharPositionInLine()==0}?=>
```

[7] Tokeny, pre ktoré neexistuje korešpondujúci vstup.

[8] Token, ktorý indikuje zanorenie, vstup do bloku.

[9] Token, ktorý indikuje vynorenie, koniec jedného bloku.

```
(' ' |'\t' )+ // match whitespace
{
if ( «indentation-bigger-than-before» ) {
// can only indent one level at a time
emit(«INDENT-token»);
«track increased indentation»
}
else if ( «indentation-smaller-than-before» ) {
int d = «current-depth » - «previous-depth »;
// back out of d code blocks
for (int i=1; i<=d; i++) {
emit(«DEDENT-token»);
}
«reduce indentation»
}
};
```
(19)

2.3 Akcie

Akcie sú bloky kódu písané v cieľovom jazyku (ten, v ktorom bude vygenerovaný výsledný parser a lexer). Akcie je možné použiť v mnohých miestach v rámci gramatiky, pričom syntax je vždy rovnaká: ľubovoľný text ohraničený zloženými zátvorkami [3].

ANTLR generuje metódu pre každé pravidlo v gramatike, ktorá je zapuzdrená v triede. ANTLR poskytuje pomenované akcie, ktoré umožňujú vložiť vlastný kód (lokálnu metódu, premennú) do vygenerovaného kódu. Syntax je nasledovná:

```
@action-name { ... }
@action-scope-name::action-name { ... }
```
(20)

Nasledujúci príklad definuje premennú a metódu za použitia akcie **members**:

```
grammar T;
@members {
  int n;
  public void foo() {...}
}

a : ID {n=34; foo();} ;
```
(21)

Ak chceme generovaný kód umiestniť do príslušného JAVA balíčku, tak použijeme akciu **header**:

```
grammar T;
@header {
   package org.antlr.test;
}                                                                    (22)
```

Ak tvoríme kombinovanú gramatiku, ktorá obsahuje lexikálne a syntaktické pravidlá, musíme mať možnosť nastaviť **members** a **header** ako pre parser, tak aj pre lexer. Stačí pridať predponu **lexer** poprípade **parser** ako action-scope[10] akcie [3].

```
grammar T;

@header {import org.antlr.test;} //not auto-copied to lexer
@lexer::header{import org.antlr.test;}
@lexer::members{int aLexerField;}                                    (23)
```

Okrem akcií popísaných v predchádzajúcej časti existujú ešte akcie, ktoré sú zabudované do pravidiel gramatiky ANTLR. Na vykonanie akcie pred všetkým ostatným v pravidle a na inicializáciu lokálnej premennej sa používa akcia **init**. Podobne na vykonanie nejakého kódu potom, čo bola vykonaná alternatíva pravidla a tesne pred tým než pravidlo vráti návratovú hodnotu (defaultne void), sa používa akcia **after**. Napríklad majme pravidlo **r**, ktoré inicializuje návratovú hodnotu na nulu predtým, ako vykoná nejakú z jeho alternatív. Po vykonaní jednej z alternatív vypíše návratovú hodnotu na štandardný výstup [3]:

```
r returns [int n]
@init {
   $n=0; // init return value
}
@after {
   System.out.println("returning value n="+$n);
}
: ... {$n=23;}
| ... {$n=9;}
| ... {$n=1;}
;                                                                    (24)
```

[10] Akcia definovaná pre konkretnú oblasť. Napríklad výhradne pre lexer, alebo pre parser. Syntax je nasledovná: *scope::action*. V preklade *oblasť*::*akcia*.

2.4 Atribúty

Na vykonanie dobrého prekladu musia akcie odkazovať na vstupné symboly. Tokeny a odkazy na pravidlá majú oba preddefinované atribúty s nimi spojené, ktoré sú užitočné počas prekladu. Napríklad, chcem pristúpiť k textu rozpoznanom pre prvky pravidla:

```
decl: type ID ';'
      {System.out.println("var "+$ID.text+":"+$type.text+";");}
     ;
type : 'int' | 'float' ;                                          (25)
```

kde *text* je preddefinovaný atribút. *$ID* je objekt tokenu a *$type* je zoskupenie dát, ktoré obsahuje vlastnosti pre konkrétny odkaz na pravidlo **type**. Ak zadáme na vstup *int x;*, prekladač vypíše na štandardný výstup: *var x:int;*. Ak sú odkazy na prvky pravidiel unikátne, akcie môžu použiť *$elementName* na prístup k združeným atribútom. Ak máme viac odkazov na prvok pravidla, *$elementName* je nejasný, preto musíme označiť jednotlivé prvky na vyriešenie jednoznačnosti. Ak máme napríklad dva identifikátory na jednom riadku a chceme pristupovať k obidvom tokenom *ID*, musíme ich označiť a v akcii sa odkazovať na ich štítky (štítky sme určili pri značení).

```
decl: t=ID id=ID ';'
      {System.out.println("var "+$id.text+":"+$t.text+";");}
     ;                                                            (26)
```

Ak pravidlo rozpoznáva prvky opakovane, prekladač bežne potrebuje zostaviť zoznam takýchto prvkov. Ako obvykle, ANTLR poskytuje `+=` operátor, ktorý automaticky pridá združené prvky do zoznamu typu `ArrayList`. Nasledujúce pravidlo zoskupí všetky identifikátory do zoznamu *ids* [3].

```
decl: type ids+=ID (',' ids+=ID)* ';' ;
//ids is list of ID tokens                                        (27)
```

Všetky tokeny rozpoznané parsrom a lexrom obsahujú kolekciu preddefinovaných read-only atribútov. Tieto atribúty zahrňujú užitočné vlastnosti samotných tokenov ako typ tokenu, text rozpoznaný pre daný token a ďalšie. Akcie môžu pristupovať k týmto atribútom cez *$label.attribute,* kde *label* označuje odkaz na daný token.

Atribút	Typ	Popis
text	String	Text rozpoznaný pre daný token
type	int	Typ tokenu. Typ daného tokenu môžeme nájsť v súbore *gramatika.tokens* ktorý vygeneroval ANTLR.
line	int	Číslo riadku, na ktorom sa nachádza daný token, počítajúc od 1.
pos	int	Pozícia prvého znaku tokenu v rámci jedného riadku, počítajúc od 0.
...		

Tabuľka 1 [3]. Najpoužívanejšie atribúty tokenov.

2.5 Sémantický predikát

Zvážte problém rozoznania prvku nie viac ako štyri krát. Prekvapujúco je toto zložité definovať pomocou syntaktických pravidiel. Za použitia čistej bezkontextovej gramatiky (inými slovami bez sémantických akcií a predikátov) je potrebné vypísať všetky možné kombinácie:

```
data:  BYTE BYTE BYTE BYTE
|      BYTE BYTE BYTE
|      BYTE BYTE
|      BYTE
;                                                    (28)
```

Ak by sme potrebovali takýto prvok rozoznať niekoľkonásobne viac krát, tak by sa toto riešenie jednoducho zrútilo. Jednoduchšie riešenie je rozoznať toľko BYTE tokenov, koľko sa na vstupe v danom momente nachádza a potom v akcii overiť, či ich nie je príliš veľa:

```
data: ( b += BYTE )+ {if ( $b.size()>4 ) «error»;};          (29)
```

Alebo je možné použiť formálny ekvivalent, ktorý poskytuje ANTLR nazývaný *overovanie sémantických predikátov*. Overovanie sémantických predikátov vyzerá ako akcia, za ktorou nasleduje otáznik:

```
data : ( b += BYTE )+ {$b.size()<=4}?
     ;                                               (30)
```

Overovanie sémantických predikátov sú booleovské výrazy, ktoré rozoznávač (recognizer) vyhodnocuje počas behu aplikácie. Ak výraz zlyhá, neuspeje aj sémantický predikát, a rozoznávač vyhodí `FailedPredicateException`, čiže výnimku [3].

Špeciálnym typom sémantických predikátov sú synchronizované sémantické predikáty. Tieto predikáty vyzerajú ako *{...}?=>* a uzatvárajú booleovské výrazy, ktoré sú vyhodnocované za behu. Synchronizovaný sémantický predikát určuje, či môže rozoznávač vybrať danú alternatívu. Ak predikát zlyhá, je alternatíva pre recognizer neviditeľná. Uvediem príklad založený na (30). Ak miesto `FailedPredicateException` výnimky chceme generovať syntaktickú chybu, použijeme synchronizované sémantické predikáty [3].

```
data
@init {int n=1;} //n becomes a local variable
: ( {n<=4}?=> BYTE {n++;} )+ //enter loop only if n<=4
;                                                          (31)
```

2.6 Zhrnutie

V kapitole venovanej ANTLR som sa snažil zachytiť podstatné vlastnosti a možnosti tohto nástroja, ktoré som využíval pri implementácií. Je to, samozrejme, len zlomok toho, čo tento nástroj skutočne dokáže. Táto malá časť by však mala viesť k pochopeniu praktík používaných v kapitole zaoberajúcej sa návrhom a implementáciou. Záujemcov o podrobné informácie k ANTLR odkazujem na zdroj [3], v ktorom nájdu kompletného sprievodcu tohto nástroja. Dozvedia sa mimo iného aj ako používať *návrátové hodnoty pravidiel*, ako vytvoriť *AST*[11] *gramatiku*, ako funguje mechanizmus rozpoznávania jazykových elementov, ako využívať *scopes* na komunikáciu medzi jednotlivými pravidlami a mnoho ďalšieho.

3 Platforma Eclipse

Eclipse je javová open source vývojová platforma. V skutočnosti je to framework (skelet), ktorý poskytuje sadu služieb na vytváranie vývojového prostredia z plugin komponentov. Eclipse je možné stiahnuť štandardne ako súhrn pluginov, vrátane Java Development Tools.

Eclipse obsahuje PDE[12], ktoré zaujíma predovšetkým vývojárov, ktorí mienia rozšíriť eclipse, odvtedy čo im to dovoľuje vytvárať nástroje jednoliato integrované s prostredím eclipse. Pretože

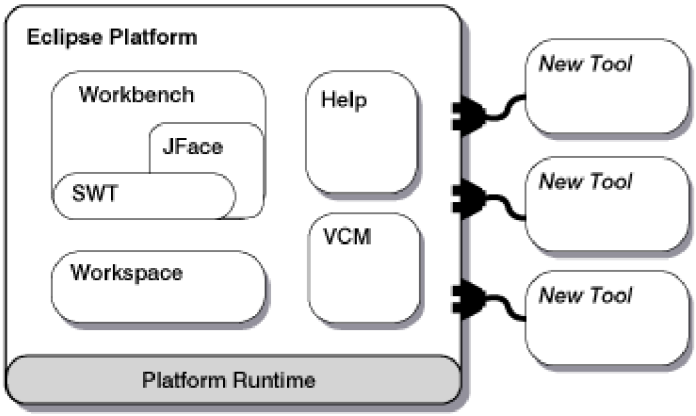
[11] Abstract Syntax Tree - abstraktný syntaktický strom.

[12] Plugin Development Environment - vývojové prostredie pluginov.

v eclipse je všetko plugin, všetci vývojári majú hracie pole rôznych úrovní pre poskytnutie rozšírení a ponúknutie konzistentných, zjednotených integrovaných vývojových prostredí pre užívateľov [4].

3.1 Architektúra

Architektúra platformy eclipse pozostáva z niekoľkých hlavných častí: *Platform run-time*, *Workspace*, *Workbench*, *Version and Configuration Management (VCM) system*, a *Help*. Informácie som získal z [4].



Obrázok 2 [6]. Architektúra eclipse.

Platform run-time je jadro, ktoré pri štarte eclipse zistí, ktoré zásuvné moduly sú nainštalované a vytvorí o nich register informácií. Na zníženie času štartu a použitých zdrojov nebude načítaný žiaden plugin, pokiaľ nie je aktuálne potrebný. Okrem jadra je všetko implementované formou plugin-u.

Workspace je plugin zodpovedný za spracovanie užívateľských zdrojov. To zahrňuje projekty, ktoré užívateľ vytvára - súbory v jednotlivých projektoch, zmeny súborov a ďalšie zdroje. Workspace informuje ostatné plugin-y o zmenách v zdrojoch - ako vytvorenie súboru, zmazanie, zmena.

Workbench poskytuje eclipse užívateľské rozhranie. Je postavený na základe Standard Widget Toolkit (SWT), Swing-u a JFace

Help komponenta ide paralélne so samotným rozšírením platformy eclipse. Takým spôsobom ako plugin pridáva funkcionalitu eclipse, help poskytuje rozširujúcu navigačnú štruktúru, ktorá dovoľuje nástrojom pridať dokumentáciu vo forme HTML súboru.

3.2 Plugin model

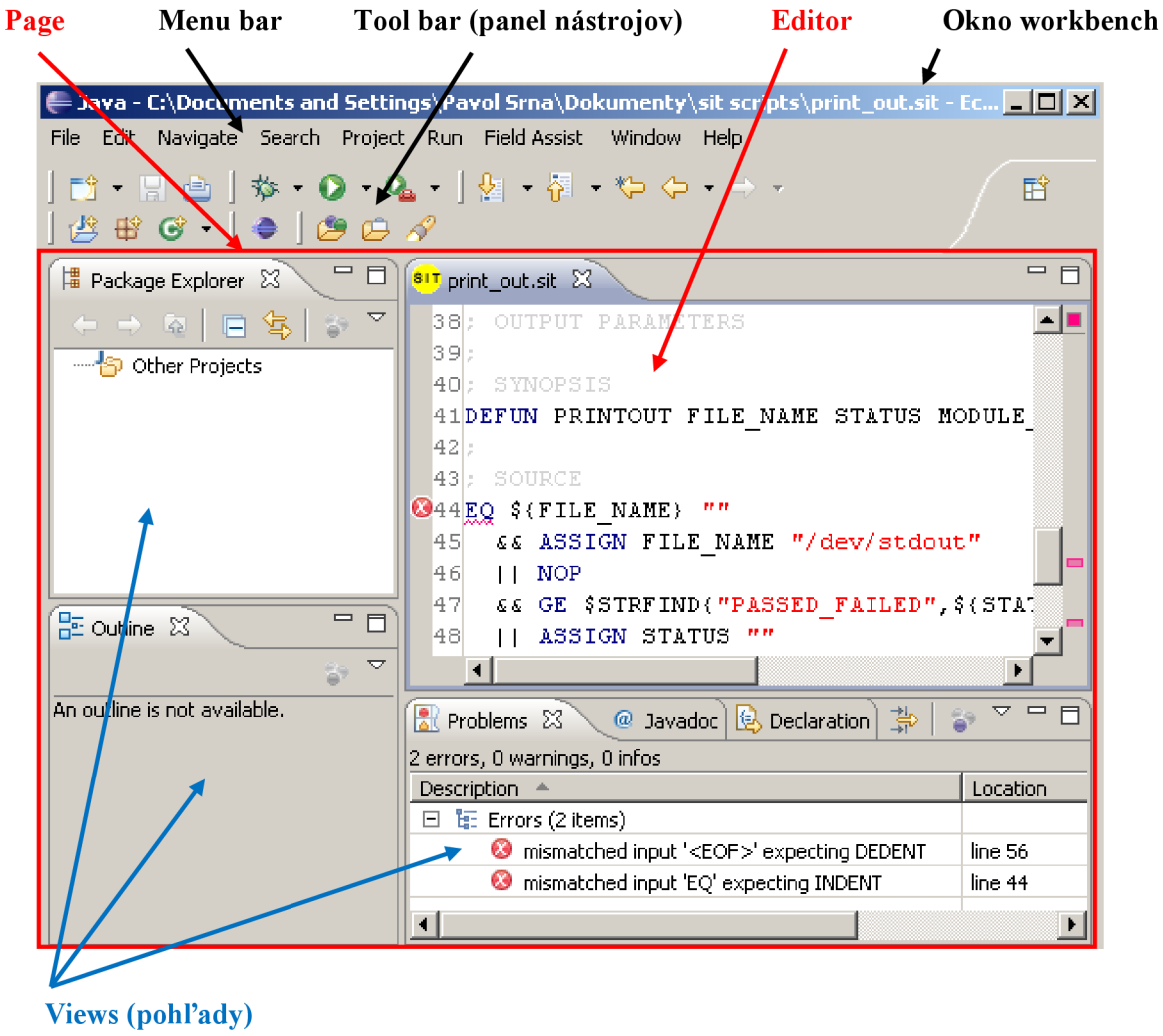
Plugin je najmenšia jednotka funkcie, ktorá môže byť vyvíjaná a dodávaná oddelene. Často je malý nástroj písaný ako zásuvný modul a funkčnosť komplexného nástroja je rozdelená naprieč mnohých plugin-ov. Zásuvné moduly pre eclipse sú písané v programovacom jazyku JAVA. Každý plugin má vlastný manifest súbor **(plugin.xml)**, ktorý deklaruje prepojenie na iné plugin-y. Model prepojenia je jednoduchý: plugin deklaruje nejaký počet bodov rozšírenia **(extension points)** a nejaký počet rozšírení **(extensions)** k jednému alebo viacerým bodom rozšírení v iných plugin-och. Bod rozšírenia jednoducho hovorí: "Mám tu pre Teba slot, aby si mi poskytol nové správanie." a rozšírenie hovorí: "Tu je nové správanie, ktoré si žiadal.". Zásuvné moduly môžu zastupovať obidve role - deklarovanie bodu rozšírenia a poskytovanie k tomuto bodu rozšírenie [6].

Definovanie bodu rozšírenia je podobné definovaniu nejakého API[13]. Jediný rozdiel spočíva v tom, že bod rozšírenia je deklarovaný za použitia XML, namiesto signatúry kódu. Hlavným cieľom tohto spôsobu je, že užívateľ neplatí pamäťové a výkonnostné penalty za plugin-y, ktoré sú nainštalované, ale sa nepoužívajú. Vysvetľujúca podstata platformového modelu rozšírení dovoľuje jadru rozhodnúť, ktoré rozšírenia a body rozšírenia sú poskytované plugin-om bez toho, aby daný zásuvný modul bolo potrebné spustiť. Preto môžu byť nainštalované mnohé plugin-y, ale žiaden z nich nebude spustený dovtedy, kým funkcia poskytovaná zásuvným modulom nebude požadovaná podľa aktivity užívateľa. Toto je dôležitá vlastnosť v poskytovaní mohutnej platformy [5].

[13] API - application programming interface. V preklade: aplikačné programové rozhranie.

3.3 Workbench

Workbench poskytuje rozsiahlu množinu tried a rozhraní pre budovanie komplexných užívateľských rozhraní. Na obrázku č. 3 je zobrazené okno, ktoré sa otvorí po štarte platformy.



Obrázok 3. Komponenty okna platformy eclipse.

Užívateľské rozhranie platformy eclipse je založené na editoroch, pohľadoch (views) a perspektívach. Z pohľadu užívateľa okno workbench vizuálne pozostáva z niekoľkých pohľadov a editorov. Perspektívy sa prejavujú vo výbere a usporiadaní editorov a pohľadov viditeľných na displeji.

Editor umožňuje užívateľovi otvárať, editovať a ukladať objekty a súbory. Pohľad poskytuje informácie o objekte, s ktorým užívateľ pracuje vo workbench-i. Pohľad by mal pomáhať editoru poskytovať informácie o dokumente, ktorý sa edituje. Okno workbench môže mať viacero perspektív, ale iba jedna môže byť aktívna v danom momente. Perspektíva usporiada skupinu editorov pre vzhľad na obrazovke [6].

Často pri programovaní plugin-ov pridávame vizuálne komponenty do workbench-u, je preto nevyhnutné sa rozhodnúť, či chceme implementovať pohľad (view) alebo editor. Bližšie informácie sú uvedené v [5].

- **Pohľad** sa zvyčajne používa na prechádzanie hierarchie informácií, na otvorenie editora, zobrazenie vlastnosti aktívneho editora. Napríklad pohľad *project explorer* umožňuje prechádzanie hierarchiou pracovného priestoru (workspace). Pohľad *properties* a *outline* ukazujú informácie o objekte v aktívnom editore. Každá zmena, ktorá môže byť vykonaná v pohľade (ako napríklad zmena nejakej vlastnosti), je uložená okamžite.

- **Editor** sa zvyčajne používa na editovanie alebo prezeranie dokumentu alebo vstupného objektu. Zmeny uskutočnené v rámci editoru nasledujú open-save-close model podobne ako v externých editoroch súborových systémov.

3.4 Možnosti eclipse

Platforma eclipse je navrhnutá a postavená na spĺňanie týchto základných požiadaviek:

- Podpora vytvárania rozmanitých nástrojov určených na vývoj ďalších aplikácií.
- Podpora neobmedzeného množstva poskytovateľov nástrojov, vrátane nezávislých predajcov softvéru.
- Podpora nástrojov na manipuláciu ľubovoľných typov obsahu.
- Uľahčenie integrácie nástrojov v rámci a naprieč rozličných typov obsahu a poskytovateľov nástrojov.
- Beh na veľkom množstve operačných systémov, vrátane operačného systému Linux, Mac OS X a Windows.
- Využitie popularity programovacieho jazyka JAVA na písanie nástrojov.

Eclipse poskytuje pre vývojárov plugin-ov sofistikovaný framework, ktorý im umožňuje vytvárať textové editory s pokročilými funkciami ako

- zvýrazňovanie syntaxe,
- kontrola syntaxe,
- automatické dopĺňanie kódu,

- automatické formátovanie kódu,
- podtrhávanie syntaktických chýb (error marking),
- ponúkanie nápovedy podľa kontextu,
- správa poznámok nad dokumentom,
- detektor hypertextových odkazov,
- ...

Mechanizmus bodov rozšírenia umožňuje pridávať vlastné wizardy, dialogové okná a príkazy. Je možné pridávať položky do panela nástrojov i do jednotlivých ponúk menu. Dokonca je možné za využitia eclipse API implementovať aj vlastný debugger.

Z môjho pohľadu predstavuje eclipse API takmer neobmedzené možnosti. Nie je to jednoduché aplikačné programovacie rozhranie na pochopenie, ale zato veľmi mohutné.

4 Návrh a implementácia

Cieľom tejto bakalárskej práce je vytvoriť editor zdrojového kódu pre programovací jazyk SIT. Tento jazyk je vyvinutý firmou Siemens. Používa sa na testovanie telekomunikačných zariadení interne v rámci firmy Siemens, ktoré pracujú nad SIP protokolom. Dokumentácia k jazyku, ako aj ostatné záležitosti spojené s týmto jazykom, spadajú pod firemné tajomstvo a nie sú prístupné verejnosti. Editor implementujem formou zásuvného modulu do eclipse. Táto kapitola sa bude zaoberať návrhom a implementáciou tohto plugin-u. Hlavná pozornosť bude venovaná dvom častiam IDE: Jadru a UI[14].

Hoci sa dnes dostupné vývojové prostredia (IDE) od seba líšia a sú preplnené množstvom funkcií, skoro každé z nich zdieľa jednu a tú istú základnú architektúru. Vývojové prostredie je všeobecne rozdelené do vrstiev a modulov závisiacich na funkčnosti jeho odlišných častí [7].

Dve z najviac bežných modulov, ktoré obsahuje každé IDE sú:

- **JADRO**, základná vrstva každého IDE, pozostávajúca z parsru/lexru, ktorý rozoznáva jazykové elementy.
- **UI**: Kód spojený s užívateľským rozhraním, ako napr. syntax highlighting, content assistant, content outline. (tieto pojmy budú neskôr v texte ozrejmené).

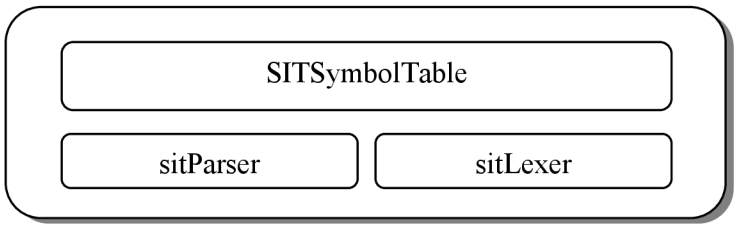
[14] UI - user interface. V preklade: užívateľské rozhranie.

Ostatné bežné moduly IDE, ktorým ale nebudem v texte venovať pozornosť:

- **DEBUG**, kód spojený s ladiacim nástrojom.
- **DOC**, dokumentácia v HTML, alebo inom formáte.
- **BUILD**, zostavovací modul, systém.

4.1 Jadro

Jadro je spodnou vrstvou vývojového prostredia (IDE), takže všetko ostatné v rámci IDE závisí priamo, či nepriamo na ňom. Preto ak máme v jadre chybu, môže sa to nepriaznivo prejaviť pri písaní kódu UI. Vedomosti o tejto problematike som získal v [7].



Obrázok 4. Štruktúra jadra.

Skoro všetky jadrá pozostávajú zo základných elementov uvedených v obrázku 4. Parser a Lexer sú potrební na rozpoznávanie zdrojového kódu, ktorý zadáva užívateľ. Tabuľka symbolov (v tomto prípade SITSymbolTable) je štruktúra, ktorá uchováva informácie o zdrojovom kóde. Ostatné časti IDE prechádzajú tabuľku symbolov a získané informácie interpretujú. Parser a lexer obvykle beží na pozadí a udržuje tabuľku symbolov aktuálnu.

Mnoho krát má tabuľka symbolov formu AST stromu. Ja som pre túto bakalársku prácu zvolil jednoduchšiu formu postačujúcu pre tento účel. Štruktúra SITSymbolTable pozostáva z niekoľkých hashovacích tabuliek, v ktorých sú uložené definované funkcie i parametre funkcií, a zo zoznamu (List) chýb objavených počas parsrovania.

Rozhodnutie, ako dobre je jadro navrhnuté, záleží na jednom faktore - rýchlosti. Rýchlosť sa stáva našou najväčšou výzvou. Výkon celého IDE závisí pravé na rýchlosti jadra. Ak nie je jadro dosť rýchle, nie je možné implementovať všetky pomyslené prvky, pretože by boli na používanie príliš pomalé. Prečo je rýchlosť jadra pri dnešných 3GHZ viacjadrových procesoroch tak dôležitá? Na vývojové prostredia (hlavne, ich editory v tomto prípade) sa kladú špeciálne požiadavky, ktoré v bežných aplikáciách nenájdeme. Určite ste si počas používanie nejakého moderného vývojového prostredia všimli, že sa počas písania kódu do jeho editora objavujú v texte červené zvlnené značky,

ktoré označujú chyby. V postrannom paneli môžete mnoho krát pozorovať hierarchiu kódu. Ak stlačíte **ctrl+space** zobrazí sa automatické dopĺňanie kódu. Všetko toto sa deje skoro v reálnom čase počas toho, ako píšete. Nepozorujete žiadne zdržanie v závislosti na tom, koľko riadkov kódu ste napísali.

Toto je miesto, v ktorom do hry vstupuje rýchlosť. IDE musí aktualizovať všetky dátové štruktúry každým zlomkom sekundy počas procesu písania. Tento prístup stojí príliš veľa výpočtového výkonu a pre užívateľa to môže znamenať, že počas písania bude pozorovať na obrazovke oneskorenie. Väčšina vývojových prostredí používa nasledujúci prístup na riešenie tohto problému:

- Parser/Lexer beží v samostatnom vlákne na pozadí.
- Vlákno beží po uplynutí nejakej pevnej doby (povedzme 200 ms) a spustí parser/lexer, aby aktualizoval tabuľku symbolov.
- UI beží vo vlastnom vlákne.

Najjednoduchší spôsob na vytvorenie parsra a lexra je zúžitkovať nejaký parser/lexer generátor. Ja použijem ANTLR, ktorému som v tejto práci venoval celú jednu kapitolu. Na rýchlosť jadra vplýva aj fakt, či je parser inkrementálny. Inkrementálny parser zaručuje, že pri každom ďalšom spustení sa nebude parsrovať celý vstup, ale iba úsek kódu, ktorý bol od posledného spustenia zmenený. ANTLR v súčasnej verzii 3.0.1 inkrementálny parser nepodporuje. V nasledujúcej verzii je už táto vlastnosť ohlásená. Eclipse API má podporu pre obidva prístupy (neinkrementálny aj inkrementálny), čo umožňuje v budúcnosti vygenerovať za pomoci ANTLR nový parser/lexer a nahradiť ten pôvodný.

4.1.1 Zákonitosti jazyka SIT

Špecifickosť jazyka SIT mi skomplikovala prácu pri tvorbe gramatiky ANTLR, preto tu chcem zdôvodniť netradičné postupy, ktoré som použil.

Z kapitoly *2.1* vieme rozoznať syntaktické pravidlá od lexikálnych. Ak sa podrobnejšie pozrieme na ANTLR gramatiku jazyka SIT, tak zistíme, že obsahuje len malý počet lexikálnych pravidiel. Nenájdeme tu žiadne lexikálne pravidlo, napríklad pre identifikátor. Mnohé časti zdrojového kódu jazyka SIT sú vyhodnotené až za behu, čo mi znemožnilo pri lexovaní zdrojového kódu presne určiť typ tokenu. Pre kompilované programovacie jazyky, akým je napríklad jazyk C, môžeme hneď po lexikálnej analýze jednoznačne povedať, na ktorom mieste v zdrojovom kóde sa nachádza reťazec alebo identifikátor, atď. Reťazec jazyka C je jednoznačne určený lexikálnym

pravidlo, ktoré hovorí, že takýto reťazec musí začínať aj končiť úvodzovkami. Ukážme si príklad lexikálneho pravidla pre identifikátor jazyka C:

```
ID   : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*
     ;                                                          (32)
```

Vidíme, že identifikátor v jazyku C nesmie začínať numerickým znakom. Prečo ale nesmie začínať numerickým znakom? Odpoveď je jednoduchá. Pretože existuje ďalšie lexikálne pravidlo *DIGIT*, (33) a lexer by nemohol jednoznačne určiť typ tokenu.

```
DIGIT  : '0'..'9'
       ;                                                        (33)
```

Ak by pravidlo pre identifikátor umožňovalo, aby začínal numerickým znakom, tak pre vstup, povedzme *345*, by lexer nedokázal určiť typ tokenu - či sa jedná o ID alebo DIGIT.

Reťazec jazyka SIT môže obsahovať akékoľvek znaky okrem medzery, avšak nemusí, ale môže byť uzavretý v úvodzovkách. Tento fakt predstavuje pre lexer problém, pretože nedokáže deterministicky určiť typ tokenu. Takýchto prípadov nájdeme v jazyku SIT viac. Preto gramatika obsahuje len veľmi málo základných lexikálnych pravidiel. Rozhodnúť, či sa jedná o reťazec alebo nejaký identifikátor, je možné až na úrovni syntaktickej analýzy. Býva zvykom, že lexer umiestni biele znaky na skrytý kanál (viď kapitola *2.2*). Túto zásadu nemôžeme v gramatike SIT uplatniť. Biele znaky, ktoré nasledujú bezprostredne za sebou zoskupíme do jedného tokenu lexikálnym pravidlom *WS* (WhiteSpace) a ponecháme na štandardnom kanále, ktorý je pre parser viditeľný. Povedzme, že máme nejakú preddefinovanú funkciu jazyka SIT (sit action), ktorá má určené parametre. Tieto parametre sú typu reťazec. To znamená, že jednotlivé parametre je možné od seba rozoznať len, ak je medzi nimi medzera. Preto aj syntaktické pravidlo musí vo svojej definícií obsahovať lexikálne pravidlo medzery, ktoré oddeľuje jednotlivé parametre. Ukážeme si jeden jednoduchší príklad akcie jazyka SIT, ktorý má jeden nepovinný parameter:

```
nop  : 'NOP' (WS nop_val)? WS? NEWLINE suite?
     ;
nop_val : eval_expr
        ;                                                       (34)
```

Všimnime si, že je nevyhnutné, aby syntaktické (parsrovacie) pravidlo *nop* obsahovalo lexikálne pravidlá *WS*. Takýto spôsob riešenia zahmlieva gramatiku, je však potrebný, aby sme dokázali vymedziť, na ktorom mieste sa v zdrojovom kóde nachádza parameter *nop_val*.

4.1.2 Bloky

Jazyk SIT začleňuje jednotlivé príkazy do blokov podľa toho, ako sú príkazy voči sebe odsadené. Túto stratégiu používa aj programovací jazyk Python, ktorého ANTLR gramatiku môžeme nájsť v [9]. Techniku odsadzovania som využil z gramatiky Pythonu a upravil vlastným potrebám.

Celá metóda spočíva v tom, že pri lexikálnej analýze podrobíme biele znaky testu. Ak sa biely znak nachádza na začiatku riadka, overíme to pomocou synchronizovaného sémantického predikátu (viď kapitola *2.5*), lexer emituje tokeny typu *LEADING_WS*[15]. Inak sa emituje klasický token WS[16]. Po dokončení lexikálnej analýzy získame stream tokenov. Do tohto streamu je ešte potrebné vložiť na správne miesto INDENT a DEDENT tokeny, ktoré budú určovať jednotlivé bloky. Túto úlohu zabezpečuje trieda `sitTokenSource`.

Našou úlohou je zistiť, či token LEADING_WS predstavuje zanorenie do ďalšieho bloku alebo vynorenie z bloku, alebo sa jedná o ten istý blok. Trieda `sitTokenSource` obsahuje zásobník, na ktorý ukladáme jednotlivé úrovne zanorenia. Na začiatku je na zásobníku jediná hodnota rovná nule. Pripomeňme si kapitolu *2.4*, kde atribút *Text* predstavuje text tokenu rozpoznaný počas lexikálnej analýzy. Ak metóda `nextToken()` triedy `sitTokenSource` vráti token LEADING_WS, vezme sa atribút *Text* tohto tokenu, ktorý predstavuje veľkosť zanorenia, a porovná sa s vrcholom zásobníka. Ak sa zhoduje, nič sa nestane. Ak je väčší ako hodnota na vrchole zásobníka, vloží sa na zásobník hodnota atribútu *Text* a vygeneruje sa INDENT token. Ak je hodnota atribútu *Text* menšia ako vrchol zásobníka, vyberú sa zo zásobníka všetky hodnoty, ktoré sú väčšie ako hodnota atribútu *Text* a pre každú vybranú hodnotu sa vygeneruje DEDENT token. Ak narazíme na koniec súboru (EOF), tak sa pre každú ostávajúcu hodnotu väčšiu ako nula vygeneruje jeden DEDENT token.

Takto upravený stream tokenov predáme ďalej k parseru. Parser už pracuje na základe svojich pravidiel a vykonáva syntaktickú analýzu. Rád by som ešte ukázal syntaktické pravidlo ***suite***, ktoré určuje kde sa musí nachádzať nový blok:

```
suite  :     INDENT (branch)+ DEDENT
       ;                                                    (35)
```

4.1.3 Pripojené súbory

Jednou z požiadaviek na editor bolo aj automatické dopĺňanie identifikátorov premenných, funkcií a štruktúr, ktoré boli definované v pripojených súboroch. Tabuľka symbolov teda musí byť naplnená aj údajmi z týchto súborov, ktoré musíme podrobiť lexikálnej a syntaktickej analýze.

---

[15] Token, ktorý reprezentuje vedúci biely znak, alebo skupinu vedúcich bielych znakov.

[16] Token, ktorý reprezentuje biele znaky, ktoré sa nenachádzajú na prvej pozícií v riadku.

Lexer dostane na vstup zdrojový kód sit skriptu vo forme prúdu znakov a začne vykonávať lexikálnu analýzu podľa pravidiel gramatiky. Ak narazí na postupnosť znakov, ktorá značí direktívu *include*, získa názov pripojeného súboru a pokúsi sa ho otvoriť. Predtým však musí pôvodný vstupný prúd uložiť na zásobník a zapamätať si, kde skončil. Po úspešnom otvorení pripojeného súboru pokračuje s lexikálnou analýzou v tomto súbore a vytvára zoznam tokenov. Ak narazí na EOF tohto súboru, vyberie zo zásobníka uložený prúd a pozíciu a pokračuje s lexikálnou analýzou nad týmto prúdom. V akcii *@lexer::members* som definoval zásobník typu SaveStruct, do ktorého ukladám prúd a pozíciu, ak lexer narazí na direktívu *include*. Taktiež, som tu redefinoval metódu `nextToken()` ktorá zaistí správne obnovenie prúdu zo zásobníka, ak lexer narazí v pripojenom súbore na token *EOF_TOKEN*. Akcia lexikálneho pravidla *INCLUDE* zaistí, že v prípade správneho otvorenia pripojeného súboru sa predchádzajúci prúd uloží vhodne na zásobník. Inak sa to interpretuje ako chyba. Výsledkom je celkový zoznam tokenov zo všetkých vstupov, ktorý predáme parsru. Ten vykonáva syntaktickú analýzu, ako keby pracoval v rámci jedného súboru.

4.1.4 Oznamovanie chýb

Štandardné nastavenie parsra vygenerovaného pomocou ANTLR je také, že chyby syntaktickej analýzy vypisuje na štandardný chybový výstup. Takéto riešenie je v praxi nepoužiteľné. V triede *SITSymbolTable* som vytvoril zoznam *errorTable,* ktorého prvky sú typu *errorTableElement*. Ten obsahuje informácie o chybe, t.j. poloha chyby v zdrojovom kóde a text chyby. Užívateľské rozhranie použije tento zoznam a jeho obsah vhodne umiestni do pohľadu Problems.

Ostáva teda vyriešiť, ako budeme tento zoznam plniť. Redefiníciou funkcie parsa, *public void displayRecognitionError (String[] tokenNames, RecognitionException e){...}*, môžeme tento výsledok dosiahnuť. Keďže nestačí len vypísať číslo a pozíciu v riadku, na ktorom chyba nastala, ale aj názov súboru (pretože chyba môže byť v pripojenom súbore), je potrebné upraviť lexer tak, aby ku každému tokenu pridal informáciu, ktorá udáva, ku ktorému súboru daný token patrí. Na základe toho som musel vytvoriť vlastný typ tokenu, ktorý vznikol dedením od *CommonToken*, do ktorého som pridal atribút *sourceName*. Tento atribút predstavuje názov súboru, ku ktorému token patrí. Problémom však je, že v metóde *public void displayRecognitionError (String[] tokenNames, RecognitionException e){...},* je možné pristúpiť k tokenu cez parameter e.token, ale tento token je spätne pretypovaný na *CommonToken* a ten neobsahuje atribút *sourceName*. Takže najjednoduchšou cestou, ako sa dostať k atribútu *sourceName* , je redefiníciou metódy *ToString()*, v ktorej vrátim informácie o tokene. Činnosť v rámci metódy *displayRecognitionError(...)* je nasledovná:

- Získať názov súboru, v ktorom nastala chyba.

- Vytvoriť text chyby zavolaním vstavanej metódy *public String getErrorMessage (RecognitionException e, String[] tokenNames).*
- Vytvoriť prvok typu *errorTableElement*, ktorý bude obsahovať získané informácie o chybe a tento prvok následne vložiť do zoznamu *errorTable*.

4.2 UI

Aká je prvá vec, ktorá Vám napadne pri pomyslení na užívateľské rozhranie vývojového prostredia (IDE)? Editor! V podstate všetko týkajúce sa užívateľského rozhrania IDE sa točí okolo editora. Toto je obzvlášť pravda pri zásuvnom module eclipse, pretože základná funkčnosť UI - workbench, tool bars, atď - je už implementovaná. Ostáva sa už len špecializovať na veci spojené s Vašim IDE [8].

Pozrieme sa na architektúru textového editora patriaceho do Eclipse. Eclipse rozdeľuje koncept textového editora do dvoch častí: *document* a *viewer* [8].

- DOCUMENT - udržuje obsah editora, nestará sa o zobrazenie na display. Napríklad parser/lexer beží v samostatnom vlákne a komunikuje iba s dokumentom. Dokument je definovaný rozhraním `org.eclipse.jface.text.IDocument`.
- VIEWER - zobrazuje obsah dokumentu na display. Napríklad Syntax highlighter komunikuje iba s *viewer*, o obsah dokumentu sa nestará. Viewer je definovaný rozhraním `org.eclipse.jface.text.ITextViewer`. Trieda `TextViewer` je navrhnutá ako všeobecná implementácia, ktorá si poradí s každým druhom textu. Eclipse poskytuje špeciálnu triedu `SourceViewer`, ktorá je podedená od `TextViewer` a je navrhnutá na zobrazenie textu vo forme zdrojového kódu. `SourceViewer` používa triedu `SourceViewerConfiguration` na selektívne pripojenie prispôsobených UI komponentov.

4.2.1 Textový editor

Trieda `org.eclipse.ui.editors.text.TextEditor` zväzuje document a viewer dohromady a pripája špecifickú funkčnosť eclipse. Editor je prepojený s triedou `IEditorInput`, ktorá definuje protokol pre vstup editora. Pre potreby tejto bakalárskej práce stačí v eclipse plugin-e vytvoriť triedu, ktorá je podedená od triedy `TextEditor` a redefinovať metódy, ktoré chceme prispôsobiť potrebám tohto zásuvného modulu.

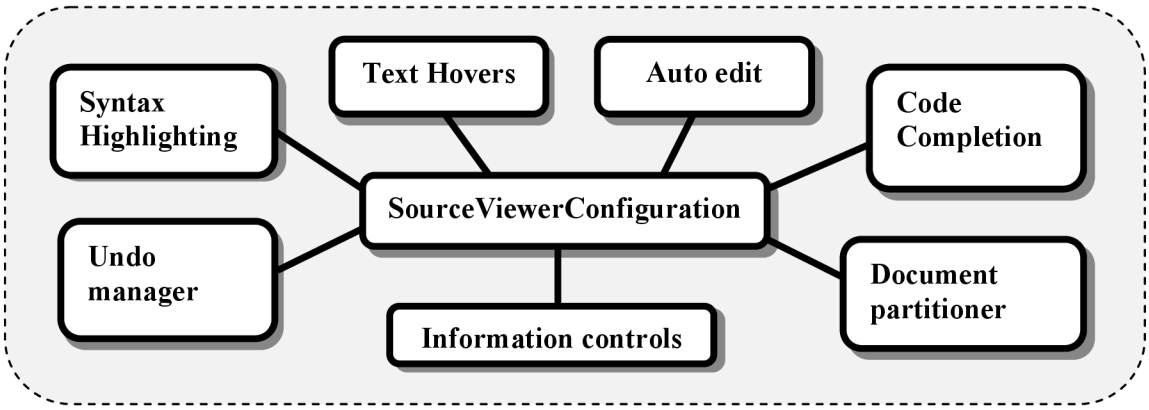
Aby však bolo možné používať textový editor v zásuvnom module, je nevyhnutné určiť podtriedu triedy `TextEditor` v súbore *plugin.xml* za použitia bodu rozšírenia `org.eclipse.ui.editors` [8]. Časť súboru *plugin.xml*:

```
<extension point="org.eclipse.ui.editors">
 <editor
       name="SIT Editor"
       extensions="sit"
       icon="icons/sample.gif"
       contributorClass="sit.SITActionContributor"
       class="sit.SITEditor"
       id="sit.SITEditor">
 </editor>
</extension>                                                    (36)
```

Za zmienku stojí hlavne atribút *class*, ktorý v tomto prípade určuje, že podtriedou triedy `TextEditor` je trieda `SITEditor` z balíčku sit. Ďalším dôležitým atribútom je *extensions*, ktorý zabezpečí, že súbory s koncovkou *sit* sa budú editovať v nami implementovanom editore. Atribút *id* je unikátne meno nášho editora. Pomocou atribútu *icon* môžeme priradiť ikonu, ktorá sa zobrazí pri jednotlivých pohľadoch vo workbench-i.

4.2.2 SourceViewerConfiguration

`SourceViewerConfiguration` je trieda, ktorá zlučuje viaceré UI komponenty dohromady. Je možné si ju predstaviť ako nejaký centrálny hub editora, do ktorého sa napojí väčšina vlastností užívateľského rozhrania [8].



Obrázok 5. Trieda SourceViewerConfiguration.

4.2.3 Rozdelenie dokumentu (Document partitioning)

Dokument si môžeme predstaviť ako reprezentáciu skutočného súboru v pamäti. Po otvorení dokumentu ho eclipse framework rozdelí na jednotlivé oddiely, štandardne na jeden základný. Jednotlivé oddiely sa nesmú prekrývať a každý oddiel patrí odlišnému typu obsahu. Povedzme, že chceme farebne zvýrazňovať syntax zdrojového kódu. Ako ale zabezpečiť, aby sa nezvýrazňovali kľúčové slová v rámci komentárov? Práve na takéto prípady slúžia oddiely. Definujeme jeden nový oddiel, ktorý bude patriť obsahu typu komentár a ponecháme ten štandardný na všetky ostatné časti zdrojového kódu. Túto funkčnosť zabezpečuje trieda `SITPartitionScanner`, ktorá dedí vlastnosti od triedy `RuleBasedPartitionScanner`. V triede `SITPartitionScanner` je definované toto pravidlo:

```
new EndOfLineRule(";", singlelinecomment);                    (37)
```

Hovorí, že ak sa v dokumente vyskytne znak `";"`, celý ostatok riadku klasifikuje ako komentár. Zavolaním rodičovskej metódy *setPredicateRules(...)* zariadime, aby sa toto pravidlo používalo pri rozlišovaní oddielov. Najskôr je však nutné pripojiť objekt triedy `SITPartitionScanner` k dokumentu. Učiníme tak v triede `SITDocumentSetupParticipant`. Ešte je potrebné redefinovať nasledujúce metódy v triede `SourceViewerConfiguration`, a oznámiť tým vlastné členenie dokumentu:

```
public String getConfiguredDocumentPartitioning(ISourceViewer sourceViewer){
     return SITEditor.SIT_PARTITIONING;
}

public String[] getConfiguredContentTypes(ISourceViewer sourceViewer) {
     return new String[] {
            IDocument.DEFAULT_CONTENT_TYPE,  SITPartitionScanner.SIT_COMMENT
     };
}                                                              (38)
```

Rozdelenie dokumentu (document partitioning) je vhodné použiť v nasledujúcich prípadoch:

- Syntax highlighting.
- Content assistant.
- Error marking.
- Formatting support.

4.2.4 Zvýrazňovanie syntaxe (Syntax highlighting)

Ak správne pochopíme rozdeľovanie dokumentu, sme na dobrej ceste k pochopeniu zvýrazňovania syntaxe. Syntax highlighting zahrňuje rozdelenie oddielu (document partition) na tokeny. Každý token má vlastné atribúty na zobrazovanie textu, podľa ktorých sa naformátuje text v editore. Eclipse poskytuje vlastnú implementáciu tokenov a pravidiel a eclipse API ponúka skener, ktorý podľa definovaných pravidiel rozčlení dokument na jednotlivé tokeny. Tento prístup veľmi zefektívni prácu, pretože nie je potrebné implementovať lexer. Hovoríte si, isteže, ale mi už lexer implementovaný máme, tak prečo ho nepoužiť. V tomto momente sa odvolám na kapitolu *4.1.1 Zákonitosti jazyka SIT*, podľa ktorej by na zvýraznenie syntaxe nestačil implementovaný lexer, ale museli by sme využiť abstraktný syntaktický strom (AST) a tree walker[17], ktorého implementácia je nad rámec tejto práce.

Spomínaný skener je implementovaný triedou `SITCodeScanner`. Pri pohľade na štruktúru tejto triedy vidíme definície tokenov a pravidiel. Metóda triedy `SourceViewerConfiguration`, ktorá sa zaoberá zvýrazňovaním syntaxe je `IPresentationReconciler getPresentation-Reconciler (ISourceViewer sourceViewer)`. Nasledujúca ukážka kódu demonštruje implementáciu tejto metódy v SIT editore:

```
public IPresentationReconciler getPresentationReconciler (ISourceViewer
     sourceViewer) {

     PresentationReconciler reconciler= new PresentationReconciler();

     DefaultDamagerRepairer dr= new DefaultDamagerRepairer(getSITCodeScanner());
     reconciler.setDamager(dr, IDocument.DEFAULT_CONTENT_TYPE);
     reconciler.setRepairer(dr, IDocument.DEFAULT_CONTENT_TYPE);
     ...
     return reconciler;
}                                                                    (39)
```

`IPresentationReconciler` je zodpovedný za sledovanie zmien v danom dokumente. Pre každý typ oddielu si udržuje referenciu na inštanciu `IPresentationDamager` a `IPresentation-Repairer`. `IPresentationDamager` a `IPresentationRepairer` sú rozhrania, ktoré implementuje trieda `DefaultDamagerRepairer` a vykonáva tak funkciu ako ničiteľa (damager), tak aj opravára (repairer). Ak je vykonaná zmena v dokumente, je poslané oznámenie ničiteľovi a ten vytvorí `IRegion`, ktorý označuje ovplyvnenú oblasť dokumentu, ktorú je potrebnú obnoviť. Obnovenie oblasti má na starosti repairer.

[17] Tree walker je program, ktorý prechádza štruktúru AST a vykonáva definované akcie.

4.2.5 Eclipse reconciler

Eclipse poskytuje framework nazývaný Eclipse *reconciler*, ktorý umožňuje umiestniť parser/lexer do samostatného vlákna. Toto vlákno zaznamenáva všetky zmeny textu v dokumente a je vyvolávané periodicky. Po tom, čo sa pripojí reconciler (nemýliť si s PresentationReconciler) k textovému editoru, vytvorí sa fronta, do ktorej sa zaznamenávajú všetky zmeny textu. Každá zmena je reprezentovaná objektom `DirtyRegion` a všetky tieto objekty sú pridané do fronty `DirtyRegionQueue`. `DirtyRegion` obsahuje nasledujúce prvky:

- Dĺžka oblasti (region).
- Offset kde začína oblasť.
- Text, ktorý bol zmenený.
- Typ obsahu oblasti (Content type of the region).

Skvelou vlastnosťou je, že ak sa vyskytne niekoľko editácií dokumentu sekvenčne za sebou, reconciler ich dokáže zlúčiť do jedného objektu `DirtyRegion` [8].

Reconciler je definovaný rozhraním `IReconciler`, ktoré obsahuje hlavne jednu metódu na prístup k `IReconcilingStrategy`. `IReconcilingStrategy` je abstrakcia eclipse na akceptovanie akéhokoľvek typu reconciler-a alebo skenera. Trieda `SITReconcilingStrategy` implementuje rozhranie `IReconcilingStrategy`. Toto rozhranie nám ponúka dve preťažené metódy *reconcile(...)*, jednu v prípade použitia inkrementálneho parsra a druhú v prípade neinkrementálneho.

Dôležitým bodom je redefinícia metódy `public IReconciler getReconciler (ISourceViewer sourceViewer)` v triede `SourceViewerConfiguration`, ktorá vytvorí objekt reconciler-a a nainštaluje ho na source viewer. Nasledujúca ukážka kódu znázorňuje tento proces:

```
@Override
public IReconciler getReconciler(ISourceViewer sourceViewer){

    MonoReconciler reconciler = new MonoReconciler(new SITReconcilingStrategy(),
    false);

    reconciler.install(sourceViewer);
    return reconciler;
}                                                                          (40)
```

Booleovská hodnota *false* značí, že reconciler beží v neinkrementálnom móde. V kapitole *4.1* som sa zmienil, že ANTLR nemá podporu inkrementálneho parsra. V novej verzii je táto vlastnosť ohlásená. Stačí teda zmeniť hodnotu *false* na *true* a kód parsra/lexra presunúť do druhej preťaženej metódy *reconcile(...)* v triede `SITReconcilingStrategy`.

4.2.6 Automatické dopĺňanie kódu (Code completion)

Jednou z veľmi obľúbených funkcií vo vyspelých moderných vývojových prostrediach je automatické dopĺňanie kódu podľa kontextu. Samozrejme, aj eclipse poskytuje rozhranie, za pomoci ktorého je možné pomerne jednoduchým spôsobom túto vlastnosť docieliť. Musíme si však uvedomiť, že implementovať skutočne inteligentný procesor na automatické doplňovanie kódu je pomerne zdĺhavý postup. Ja som na ilustráciu v tejto bakalárskej práci implementoval automatické dopĺňanie kľúčových slov a funkcií, ktoré už boli alebo sú definované v pripojených súboroch.

Trieda `ContentAssistant` implementuje rozhranie `IContentAssistant`, ktoré poskytuje podporu interaktívneho dopĺňania kódu. Objekt tejto triedy je umiestnený v redefinovanej metóde `public IContentAssistant getContentAssistant (ISourceViewer sourceViewer)` triedy `SourceViewerConfiguration`. ContentAssistant je `ITextViewer` add-on, ktorého cieľom je ponúkať, zobrazovať a vkladať možnosti kódu do dokumentu na pozíciu kurzora. Zoznam možností získa za použitia `IContentAssistProcessor`-a. Trieda `SITCompletionProcessor` implementuje rozhranie `IContentAssistProcessor`. Práve v tejto triede je ukrytá celá logika a inteligencia celého mechanizmu automatického dopĺňania kódu. V tejto triede je určené, kedy sa má zobraziť ponuka na dopĺňanie kódu. Napríklad, po stlačení kombinácie klávesov **ctrl+space** sa ContentAssistProcessor najprv uistí, či sa nachádza pred kurzorom kľúčové slovo *FUN*[18], v takomto prípade sa ponúkne zoznam definovaných funkcií. Ak sa pred kurzorom toto kľúčové slovo nenachádza, znamená to, že užívateľ sa v kóde nesnaží zavolať už definovanú funkciu. Preto sa v takomto prípade ponúkne zoznam kľúčových slov a preddefinovaných akcií jazyka SIT.

4.2.7 Podtrhávanie syntaktických chýb (Error marking)

Reconciler zabezpečí, že sa pri každej zmene textu na pozadí vykonáva syntaktická analýza. Ak nastane počas analýzy chyba, informácie o tejto chybe sa uložia v zozname chýb v štruktúre SITSymbolTable. Viď kapitola *4.1.4*. Je nevyhnutné tieto informácie interpretovať užívateľovi vizuálne.

Značky (markers) sú všeobecný mechanizmus ako asociovať poznámky a meta-data so zdrojmi. Sú poskytované rozhraním `IMarker`. Eclipse poskytuje 5 základných preddefinovaných značiek:

- `org.eclipse.core.resources.marker`,
- `org.eclipse.core.resources.taskmarker`,
- `org.eclipse.core.resources.problemmarker`,

[18] FUN je kľúčové slovo jazyka SIT na zavolanie definovanej funkcie.

- `org.eclipse.core.resources.bookmark`,
- `org.eclipse.core.resources.textmarker`.

Implementoval som triedu `SITMarkingErrorHandler`, ktorá preberá dáta zo zoznamu chýb a z informácií o chybe vytvorí značku typu `problemmarker`. Takúto značku stačí za pomoci metódy *createMarker* vytvoriť nad pracovným priestorom (workspace) plugin-u a daná značka sa nám zobrazí ako chyba v pohľade Problems okna workbench.

Moderné vývojove prostredia dokážu navyše chyby zobraziť priamo v kóde červeným zvĺneným podtrhnutím. Ani toto nie je v eclipse žiaden problém. Rozhranie `IAnnotationModel` definuje model na spravovanie poznámok (annotations). Tento model získame zavolaním metódy *getAnnotationModel()*. Nasledujúci úsek kódu ukazuje ako pridať poznámku do tohto modelu:

```
model.addAnnotation(new SimpleMarkerAnnotation(marker), position);
                                                                    (41)
```

Parameter *marker* v konštruktore *SimpeMarkerAnnotation* je typu `problemmarker`, ktorý zariadi, že sa poznámka zobrazí ako chyba a text sa na pozícií *position* červeno vlnovito podtrhne.

5 Záver

Hlavným cieľom tejto bakalárskej práce bolo zoznámiť sa s aplikačným rozhraním vývojového prostredia eclipse a vhodne rozšíriť jeho editor zdrojového kódu tak, aby mal rysy moderných textových editorov. Naštudovaním materiálov uvedených v *Literatúre* som získal cenné znalosti, ktoré som využil pri návrhu a implementácií zásuvného modulu editora zdrojového kódu pre jazyk SIT v prostredí eclipse.

Jazyk SIT je komplexný natoľko, že by bolo výhodné umiestniť moduly jadra a užívateľského rozhrania požadovaného editora do samostatných projektov a pracovať na nich oddelene a tímovo. Mojim cieľom však bolo ukázať tvorbu moderných editorov a prácu s vývojovým prostredím eclipse, nie implementovať do detailov plne prepracované IDE. Priemernú väčšinu praktickej časti práce som sa venoval návrhu a implementácii gramatiky jazyka SIT, z ktorej som za pomoci nástroja ANLTR vygeneroval parser a lexer. Pri tvorbe gramatiky som vychádzal z poskytnutých vzorových skriptov od spoločnosti ANF DATA a z dokumentácie jazyka. Tieto materiály sú však pre širokú verejnosť neprístupné. Výsledná gramatika ANTLR jazyka SIT nie je úplne kompletná, avšak implementuje taký rozsah jazyka, ktorý pre účely tejto bakalárskej práce stačí nad rámec.

Plugin, ktorý som implementoval, dokáže farebne rozlišovať a kontrolovať syntax. Syntaktické chyby zobrazuje užívateľovi v pohľade Problems okna workbench. Miesto v zdrojovom kóde, kde nastala chyba je vyznačené červeným zvlneným podtrhnutím. Zásuvný modul poskytuje automatické dopĺňanie kľúčových slov, identifikátorov funkcií, ktoré boli definované, alebo sú definované v pripojených súboroch. Mojou prácou som postavil kvalitné základy, z ktorých by v budúcnosti mohlo vyrásť vysoko funkčné IDE a zjednodušiť tak prácu pri písaní skriptov v jazyku SIT.

Napriek všetkým týmto vlastnostiam, je tu stále priestor na budúce rozšírenie. Najpodstatnejším prínosom by bolo implementovanie stromovej gramatiky (z anglického *tree grammar*). Štruktúra abstraktného syntaktického stromu (AST) by sa využila na prepracovanejšie zvýrazňovanie syntaxe. Veľmi výhodne by bolo implementovať sémantickú kontrolu, ktorá je pri tomto type jazyka skoro nevyhnutná. ANTLR scopes poskytuje možnosť na vytváranie lokálnych premenných v rámci bloku kódu, ktoré sú z ostatných blokov neviditeľné. Prospešné by bolo túto vlastnosť využiť a prepracovať ju spolu s inteli-sense (ContentAssistProcessor-om) na dosiahnutie kvalitnejších a presnejších možností automatického dopĺňania kódu.

Literatúra

[1] Backus – Naur Form. Wikipedia, 2008, dokument dostupný na URL http://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form (apríl, 2008)

[2] Garshol, Lars Marius: BNF and EBNF: What are they and how do they work? 2003, dokument dostupný na URL http://www.garshol.priv.no/download/text/bnf.html (apríl, 2008)

[3] Parr, Terence: The Definitive ANTLR Reference: Buidling Domain-Specific Languages. Norh Carolina, The Pragmatic Bookshelf, 2007.

[4] Gallardo, David: Getting started with the Eclipse Platfrom: Using Eclipse plug-ins to edit, compile, and debug your app. IBM Corporation, 2002, dokument dostupný na URL http://www.ibm.com/developerworks/opensource/library/os-ecov/ (apríl, 2008)

[5] Eclipse: Eclipse Documentation-Stable build. Dokument dostupný na URL http://help.eclipse.org/stable/ (apríl, 2008)

[6] Erickson, Mark: Working the Eclipse Platform. IBM Corporation, 2001. http://www.ibm.com/developerworks/opensource/library/os-plat/ (apríl, 2008)

[7] Deva, Prashant: Create a Commrecial-quality Eclipse IDE, Part 1: The Core. IBM Corporation, 2006, dokument dostupný na URL http://www.ibm.com/developerworks/edu/os-dw-os-ecl-commplgin1.html (apríl, 2008)

[8] Deva, Prashant: Create a Commercial-quality Eclipse IDE, Part2: The user interface. IBM Corporation, 2006, dokument dostupný na URL http://www.ibm.com/developerworks/edu/os-dw-os-ecl-commplgin2.html (apríl, 2008)

[9] ANTLRv3: Grammat List. Dokument dostupný na uRL http://www.antlr.org/grammar/list (apríl, 2008)

[10] Lexical analysis. Wikipedia, 2008, dokument dostupný na URL http://en.wikipedia.org/wiki/Lexical_analysis (apríl, 2008)

[11] Parsing. Wikipedia, 2008, dokument dostupný na URL http://en.wikipedia.org/wiki/Syntactic_Analysis (apríl, 2008)

[12] LL Parser. Wikipedia, 2008, dokument dostupný na URL http://en.wikipedia.org/wiki/LL_parser (apríl, 2008)

[13] Zoio, Phil: Building an Eclipse Text Editor with JFace Text. Realsolve Developer Corner, 2006, dokument dostupný na URL http://www.realsolve.co.uk/site/tech/jface-text.php (apríl, 2008)

Zoznam príloh

Príloha 1. DVD s manuálom a zdrojovými textami.