\Chapter{Szintaktikai elemzés}

% TODO: Formális nyelvekkel, fordítóprogramokkal kapcsolatos könyvek hivatkozásai.

% http://www.informatik.uni-bremen.de/agbkb/lehre/ccfl/Material/ALSUdragonbook.pdf

Ennek a célja az, hogy megállapítsa, hogy érvényes programról van-e szó, illetve hogy majd egyszerűen fa struktúrába rendezhető legyen a program. (AST és CST problémaköre)

Ide kerül a szintaktikai elemzés elemzés elterjedt módjainak a bemutatása. Elvi szinten

\subsection{Példa az összehasonlításhoz}

Az alábbiakban a programozási nyelv nyelvtanának felírása történik meg Extended Backus-Naur From és szintaxis diagram segítségével.

Program ::= Class+

Class ::= "Create" Space Identifier Space "(" Identifier+ ")" ("ex" Identifier)? LineBreak ((Tabulator Assignment LineBreak) | (Tabulator Function LineBreak))+

Function ::= "Funct" "(" Identifier+ ")" LineBreak (Tabulator (Assignment | Instruction | Selection | Choose | Loop | EndLoop) LineBreak)+

EndLoop ::= "Loop" LineBreak (Tabulator (Assignment | Instruction) LineBreak)+ Condition

Loop ::= "Loop" Space Condition LineBreak (Tabulator (Assignment | Instruction) LineBreak)+

Choose ::= "Switch" Space Identifier LineBreak (Tabulator Identifier ":" LineBreak (Tabulator (Instruction | Assignment) LineBreak))+ Tabulator "Def" ":" LineBreak Tabulator (Instruction | Assignment) LineBreak

Selection ::= "If" Space Condition LineBreak (Tabulator (Assigment | Instruction))+ ("EIf" Space Condition LineBreak (Tabulator (Assigment | Instruction))+)\* ("Else" Space Condition LineBreak (Tabulator (Assigment | Instruction))+)?

Condition ::= Identifier ("=" | ">=" | "<=" | ">" | "<") (Identifier | Character | String | Digit)

Instruction ::= Identifier "=" (Identifier Operator Identifier)+

Assignment ::= Identifier "=" (Character | String | Digit)+

Identifier ::= Character (Character | Digit)+

String ::= '"' Character+ '"'

Character ::= [a-zA-Z]+

Digit ::= [0-9]+

Space ::= " "

LineBreak ::= "\n" | "\r" | "\r\n"

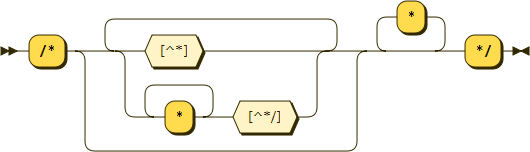
Tabulator ::= "\t"

Operator ::= "+" | "-" | "\*" | "/"

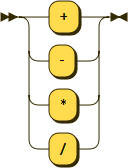
Comment ::= '/\*' ( [^\*] | '\*'+ [^\*/] )\* '\*'\* '\*/'

A szintaxis diagrammal történő felírás

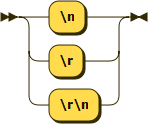
Comment



Operator



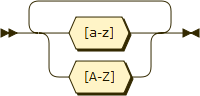
LineBreak



Digit



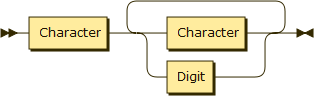
Character



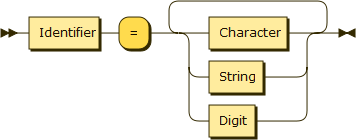
String



Identifier



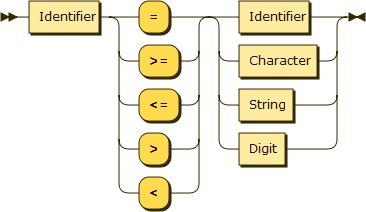
Assignment



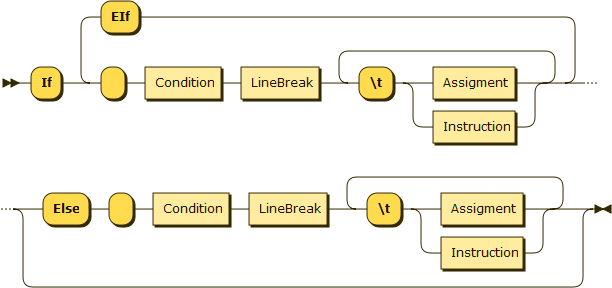
Instruction



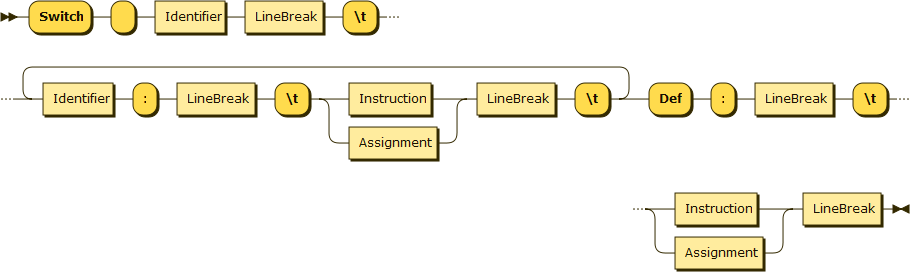
Condition



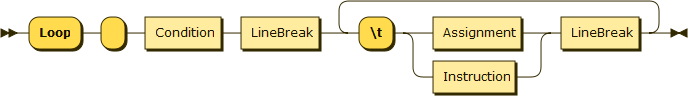
Selection



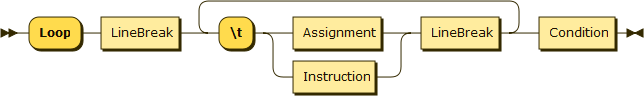
Choose



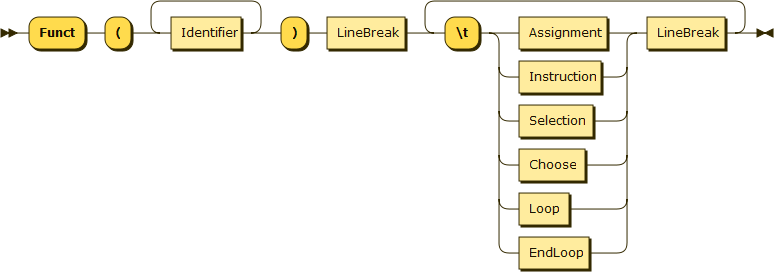
Loop



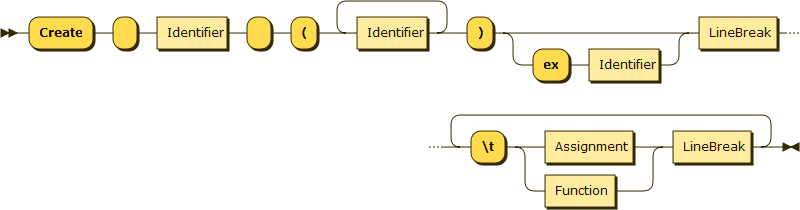
EndLoop



Function



Class



Program



Egy példa forráskód részlet , melyben egy osztály és a benne lévő elemek láthatók

Create NewLangExample(elsoParam, masodikParam)

harmadikParam = 10

funct addAllParam()

elsoParam + masodikParam + harmadikParam

Az alábbiakban a fenti osztály példányosítása és használatának példája látható

pelda = NewLangExample(1, 3)

pelda.setElsoParam(5)

szam = pelda.addAllParam

\section{Java parser generátorok}

Az interneten sokféle parser generátor található, mely segítségével a szintaktikai elemzés könnyebbé válik, megoldható.

A feladat megoldásához olyan parser generátorra van szükség, mely Java nyelvű, mivel a fordítóprogram ezen a nyelvek kerül megírása.

Emellett a parser generátorok feldolgozás szempontjából is sokfélék, és jelen feladathoz olyan generátort kellett keresni, mely reguláris nyelvvel képes működni.

Az alábbiakban a legelterjedtebb ilyen generátorokat vizsgáljuk meg.

\subsection{AnnoFlex}

Az Annoflex egy Java alapú parser generátor, mely szabadon letölthető és használható, sőt módosítható is. Az AnnoFlex implementálható mind az Eclipse, mind a InteliJ fejlesztőpi környezetekbe, külső eszközként, így bármikor használható lesz.

AZ AnnoFlex megalkotáskor is a használata minél egyszerűbbé tétele volt a fő szempont, legegyszerűbb esetben az alábbi kódot kell megírnunk:

/\*\*

\* **@option** methodName = getNextToken

\* **@option** statistics = enabled

\*/

**public** **class** Example\_Annoflex {

/\*\* **@expr** [0-9]+ \*/ String createNumber() { **return** "number"; }

/\*\* **@expr** [a-zA-Z]+ \*/ String createIdentifier() { **return** "identifier"; }

/\*\* **@expr** [ \n\r\t\f]+ \*/ String createWhitespace() { **return** "whitespace"; }

//%%LEX-MAIN-START%%

//%%LEX-MAIN-END%%

}

Az AnnoFlexnek meg kell adni a beállításokat, méghozzá a megvalósítása szerint az osztálynév előtt és komment formájában kell megadni őket, valamint fontos, hogy a @option annotáció előzze meg őket. Példánkban a metódus nevét állítottuk be, illetve azt, hogy a generálás során statisztikát jelenítsen meg a program.

Ezután az osztályon belül meg kell adni a kifejezéseket és a metódusokat. Itt találkozhatunk több megkötéssel is az AnnoFlex tekintetében. A megadáskor mindenképpen @expr kifejezéssel kell kezdeni, melyet szintén kommentben kell elhelyezni.

A kifejezés után csak és kizárólag reguláris kifejezés állhat. A komment után magát a metódust kell megírni, melynél szintén van megkötés. A metódusok nem lehetnek static módosítóval ellátva, visszatérési értékük nem lehet csak primitív típus vagy String, de mindenképpen az összes így megadott metódusnak ugyanolyan visszatérési értékkel kell bírnia, egyetlen kivétellel, ami a void. Void visszatérési érték állhat más visszetérési érték mellett. További megkötés, hogy a metódusoknak nem lehet paraméterük, csak visszatérési értékük.

Minden egyes metódust csak egy darab reguláris kifejezés előzhet meg, ha több reguláris kifejezés is kellene, hogy ott álljon, akkor a reguláris kifejezések uniójával oldható ez meg, melyhez a | operátor használható.

Ez után következik egy tagek által határolt üres rész, ezt mindig a //%%LEX-MAIN-START%% és //%%LEX-MAIN-END%% határolja, és ide kerül legenerálásra a tulajdonképpeni kód. Ebben ad nagy segítséget az AnnoFlex.

Amennyiben elkészültünk a kifejezések megírásával, akkor a fejlesztőkörnyezetben lefuttathatjuk a AnnoFlex programot, mely eredménye a következő lesz:

/\*\*

\* **@option** methodName = getNextToken

\* **@option** statistics = enabled

\*/

**public** **class** Example\_Annoflex {

/\*\* **@expr** [0-9]+ \*/ String createNumber() { **return** "number"; }

/\*\* **@expr** [a-zA-Z]+ \*/ String createIdentifier() { **return** "identifier"; }

/\*\* **@expr** [ \n\r\t\f]+ \*/ String createWhitespace() { **return** "whitespace"; }

/\*\* **@expr** [^] \*/ String createMisc() { **return** "misc"; }

//%%LEX-MAIN-START%%

//================================================

// \_ \_\_\_\_\_ \_

// / \ \_ \_\_ \_ \_\_ \_\_\_ | \_\_\_| | \_\_\_ \_ \_\_

// / \_ \ | \_ \| \_ \ / \_ \| |\_ | |/ \_ \ \/ /

// / \_\_\_ \| | | | | | | (\_) | \_| | | \_\_/> <

// /\_/ \\_\\_| |\_|\_| |\_|\\_\_\_/|\_| |\_|\\_\_\_/\_/\\_\

//

//================================================

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Generation Statistics \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* \*

\* Rules: 4 \*

\* Lookaheads: 0 \*

\* Alphabet length: 4 \*

\* NFA states: 15 \*

\* DFA states: 5 \*

\* Static size: 308 Bytes \*

\* Instance size: 24 Bytes \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//=================

// Table Constants

//=================

/\*\*

\* Maps Unicode characters to DFA input symbols.

\*/

**private** **static** **final** **byte**[] ***CHARACTER\_MAP*** = *createCharacterMap*(

"\0\t\3\2\0\1\3\2\0\22\3\1\0\17\1\n\0\7\2\32\0\6\2\32");

/\*\*

\* The transition table of the DFA.

\*/

**private** **static** **final** **byte**[][] ***TRANSITION\_TABLE*** = *createTransitionTable*(

"\5\1\2\1\3\1\4\1\0\1\2\1\0\2\0\2\3\1\0\1\0\3\4\1\0\4");

/\*\*

\* Maps state numbers to action numbers.

\*/

**private** **static** **final** **byte**[] ***ACTION\_MAP*** = *createActionMap*(

"\0\1\1\1\2\1\3\1\4\1");

//===============

// String Fields

//===============

/\*\*

\* The current string to be scanned.

\*/

**private** String string = "";

//===============

// Region Fields

//===============

/\*\*

\* The start of the scan region.

\*/

**private** **int** regionStart;

/\*\*

\* The end of the scan region.

\*/

**private** **int** regionEnd;

//============

// Dot Fields

//============

/\*\*

\* The start position of the next scan.

\*/

**private** **int** dot;

//==============

// Match Fields

//==============

/\*\*

\* The start of the last match.

\*/

**private** **int** matchStart;

/\*\*

\* The end of the last match.

\*/

**private** **int** matchEnd;

//===============

// Table Methods

//===============

/\*\*

\* Creates the character map of the scanner.

\*

\* **@param** characterMapData The compressed data of the character map.

\* **@return** The character map of the scanner.

\*/

**private** **static** **byte**[] createCharacterMap(String characterMapData) {

**byte**[] characterMap = **new** **byte**[123];

**int** length = characterMapData.length();

**int** i = 0;

**int** j = 0;

**while** (i < length) {

**byte** curValue = (**byte**)characterMapData.charAt(i++);

**for** (**int** x=characterMapData.charAt(i++);x>0;x--) {

characterMap[j++] = curValue;

}

}

**return** characterMap;

}

/\*\*

\* Creates the transition table of the scanner.

\*

\* **@param** transitionTableData The compressed data of the transition table.

\* **@return** The transition table of the scanner.

\*/

**private** **static** **byte**[][] createTransitionTable(String transitionTableData) {

**byte**[][] transitionTable = **new** **byte**[5][4];

**int** length = transitionTableData.length();

**int** i = 0;

**int** j = 0;

**int** k = 0;

**while** (i < length) {

**byte** curValue = (**byte**)((**short**)transitionTableData.charAt(i++) - 1);

**for** (**int** x=transitionTableData.charAt(i++);x>0;x--) {

transitionTable[j][k++] = curValue;

}

**if** (k == 4) {

k = 0;

j++;

}

}

**return** transitionTable;

}

/\*\*

\* Creates the action map of the scanner.

\*

\* **@param** actionMapData The compressed data of the action map.

\* **@return** The action map of the scanner.

\*/

**private** **static** **byte**[] createActionMap(String actionMapData) {

**byte**[] actionMap = **new** **byte**[5];

**int** length = actionMapData.length();

**int** i = 0;

**int** j = 0;

**while** (i < length) {

**byte** curValue = (**byte**)((**short**)actionMapData.charAt(i++) - 1);

**for** (**int** x=actionMapData.charAt(i++);x>0;x--) {

actionMap[j++] = curValue;

}

}

**return** actionMap;

}

//================

// String Methods

//================

/\*\*

\* Sets the string to be scanned. The scan region is set to the entire

\* string.

\*

\* **@param** string The new string to be scanned.

\*/

**public** **void** setString(String string) {

**this**.string = string != **null** ? string : "";

regionStart = 0;

regionEnd = **this**.string.length();

dot = 0;

matchStart = 0;

matchEnd = 0;

}

/\*\*

\* Returns the current string to be scanned.

\*

\* **@return** The current string to be scanned.

\*/

**public** String getString() {

**return** string;

}

//================

// Region Methods

//================

/\*\*

\* Sets the scan region. The dot is clipped to the region if necessary.

\*

\* **@param** start The start of the scan region (inclusive).

\* **@param** end The end of the scan region (exclusive).

\* **@throws** IllegalArgumentException If the region is out of range

\*/

**public** **void** setRegion(**int** start, **int** end) {

**if** ((start < 0) || (end > string.length()) || (start > end)) {

**throw** **new** IllegalArgumentException("region offsets out of range");

}

regionStart = start;

regionEnd = end;

**if** (dot < start) {

dot = start;

}

**if** (dot > end) {

dot = end;

}

}

/\*\*

\* Returns the start of the scan region.

\*

\* **@return** The start of the scan region.

\*/

**public** **int** getRegionStart() {

**return** regionStart;

}

/\*\*

\* Returns the end of the scan region.

\*

\* **@return** The end of the scan region.

\*/

**public** **int** getRegionEnd() {

**return** regionEnd;

}

//=============

// Dot Methods

//=============

/\*\*

\* Sets the position at which the next scan starts.

\*

\* **@param** dot The new position at which the next scan starts.

\* **@throws** IllegalArgumentException If the position is out of range

\*/

**public** **void** setDot(**int** dot) {

**if** ((dot < regionStart) || (dot > regionEnd)) {

**throw** **new** IllegalArgumentException("dot out of range");

}

**this**.dot = dot;

}

/\*\*

\* Returns the position at which the next scan starts.

\*

\* **@return** The position at which the next scan starts.

\*/

**public** **int** getDot() {

**return** dot;

}

//===============

// Match Methods

//===============

/\*\*

\* Returns the start (inclusive) of the last match.

\*

\* **@return** The start (inclusive) of the last match.

\*/

**public** **int** getMatchStart() {

**return** matchStart;

}

/\*\*

\* Returns the end (exclusive) of the last match.

\*

\* **@return** The end (exclusive) of the last match.

\*/

**public** **int** getMatchEnd() {

**return** matchEnd;

}

/\*\*

\* Returns the length of the last match.

\*

\* **@return** The length of the last match.

\*/

**public** **int** getMatchLength() {

**return** matchEnd - matchStart;

}

/\*\*

\* Returns the text of the last match.

\*

\* **@return** The text of the last match.

\*/

**public** String getMatchText() {

**return** string.substring(matchStart,matchEnd);

}

/\*\*

\* Returns a character relative to the start of the last match.

\*

\* **@param** index The index of the character relative to the last match.

\* **@return** The character at the specified position.

\* **@throws** IndexOutOfBoundsException If the specified index is invalid

\*/

**public** **char** getMatchChar(**int** index) {

**int** stringIndex = matchStart + index;

**if** ((stringIndex < regionStart) || (stringIndex >= regionEnd)) {

**throw** **new** IndexOutOfBoundsException("match character not available");

}

**return** string.charAt(stringIndex);

}

//==============

// Scan Methods

//==============

/\*\*

\* Performs at the current position the next step of the lexical analysis

\* and returns the result.

\*

\* **@return** The result of the next step of the lexical analysis.

\* **@throws** IllegalStateException If a lexical error occurs

\*/

**public** String getNextToken() {

**if** (dot < regionEnd) {

// find longest match

**int** curState = 0;

**int** iterator = dot;

**int** matchState = -1;

**int** matchPosition = 0;

**do** {

**char** curChar = string.charAt(iterator);

curState = ***TRANSITION\_TABLE***[curState][curChar >= 123 ?

0 : ***CHARACTER\_MAP***[curChar]];

**if** (curState == -1) {

**break**;

}

**if** (***ACTION\_MAP***[curState] != -1) {

matchState = curState;

matchPosition = iterator;

}

} **while** (++iterator < regionEnd);

// match found, perform action

**if** (matchState != -1) {

**int** endPosition = matchPosition + 1;

matchStart = dot;

matchEnd = endPosition;

dot = endPosition;

**switch**(***ACTION\_MAP***[matchState]) {

**case** 0: **return** createNumber();

**case** 1: **return** createIdentifier();

**case** 2: **return** createWhitespace();

**case** 3: **return** createMisc();

}

}

// no match found, set match values and report as error

matchStart = dot;

matchEnd = dot;

**throw** **new** IllegalStateException("invalid input");

}

// no match found, set match values and return to caller

matchStart = dot;

matchEnd = dot;

**return** **null**;

}

//%%LEX-MAIN-END%%

}

A fenti kódban látható, hogy a program legenerálta a szükséges metódusokat és funkciókat. Ezután már nem kell mást tenni, mint egy futtatható osztályt készíteni a példához:

**public** **class** Example\_AnnoRun {

**public** **static** **void** main(String[] args) {

Example\_Annoflex anno = **new** Example\_Annoflex();

anno.setString("Ez 1 teszt string");

System.***out***.println("Scan:" + anno.getString());

String token = anno.getNextToken();

**while** (token != **null**) {

System.***out***.println(token + ":" + anno.getMatchText());

token = anno.getNextToken();

}

}

}

Ebben csak létrehozunk egy példányt az osztályból, hozzáadunk egy szöveget és lefuttatjuk a programot. A képernyőre a következő eredményt fogja a program kiírni:

Scan:Ez 1 teszt string

identifier:Ez

whitespace:

number:1

whitespace:

identifier:teszt

whitespace:

identifier:string

Látható, hogy identifier-nek jelezte a szövegeket, a számot numberként jelenítette meg és megtalálta a fehér karaktereket is, azaz a szóközöket.

\subsection{JFlex}

A JFlex szigorú értelembe véve egy lexikai elemző, azaz lexer generátor, mely Java nyelvhez Java nyelven írt generátor. Itt is a megadott inputot próbálja meg illeszteni a különféle előre definiált nyelvtani elemekre és az annak megfelelő utasításokat hajtja végre.

A JFlex a JLex átírt változata, melynek átírásakor a cél a teljes unicode támogatás és platformfüggetlenség volt, illetve a gyors szkenner generálás, kényelmes szintaktika és az is, hogy kompatibilis legyen a JLex-el. Önállóan is használható, de mivel főképp lexer generátor, így más parser generátorokkal történő együttműködésre tervezték, leginkább a CUP parserrel kompatibilis.

Felépítése alapján a nyelvtani specifikáció három részre osztható, melyeket a ’%%’ jel választ el. Az első a felhasználói kód, a második a beállítások és makrók része, míg a harmadik fogja tartalmazni a lexer szabályokat.

import java\_cup.runtime.\*;

%%

/\* -----------------Options and Declarations Section----------------- \*/

/\*Generálandó osztály neve pl: Lexer.java\*/

%class Lexer

/\*oszlopok és sorok száma, yyline és yycolumn változókkal beállítható \*/

%line

%column

/\* CUP kompatibilitási mód \*/

%cup

/\* Az %{ és %} mindig sor elején kell szerepelnie, és a közöttük lévő leírás egy az egyben átmásolásra kerül a lexer osztályba. \*/

%{

private Symbol symbol(int type) {

return new Symbol(type, yyline, yycolumn);

}

private Symbol symbol(int type, Object value) {

return new Symbol(type, yyline, yycolumn, value);

}

%}

/\* Makrók reguláris kifejezésként, melyek később a Lexer Rules részben felhasználásra kerülnek \*/

LineTerminator = \r|\n|\r\n

WhiteSpace = {LineTerminator} | [ \t\f]

Digit = 0 | [1-9][0-9]\*

String = [A-Za-z\_][A-Za-z\_0-9]\*

%%

/\* ------------------------Lexical Rules Section---------------------- \*/

/\*YYINITIAL állapot esetén kezdődik a szkennelés \*/

<YYINITIAL> {

/\* Return the token SEMI declared in the class sym that was found. \*/

";" { return symbol(sym.SEMI); }

/\* A sym osztályban deklarált tokennel tér vissza \*/

"+" { System.out.print(" + "); return symbol(sym.PLUS); }

"-" { System.out.print(" - "); return symbol(sym.MINUS); }

"\*" { System.out.print(" \* "); return symbol(sym.TIMES); }

"/" { System.out.print(" / "); return symbol(sym.DIVIDE); }

"(" { System.out.print(" ( "); return symbol(sym.LPAREN); }

")" { System.out.print(" ) "); return symbol(sym.RPAREN); }

/\* A NUMBER tokennel tér vissza, ami egy integert reprezentál, az értéke yytext változóban van tárolva \*/

{Digit } { System.out.print(yytext());

return symbol(sym.NUMBER, new Integer(yytext())); }

/\* If an identifier is found print it out, return the token ID

that represents an identifier and the default value one that is

given to all identifiers. \*/

{String { System.out.print(yytext());

return symbol(sym.ID, new Integer(1));}

/\* Whitespace esetén semmi sem történik \*/

{WhiteSpace} { /\* skip \*/ }

}

/\*érvénytelen karakter \*/

[^] { throw new Error("Illegal character <"+yytext()+">"); }

\subsection{AustenX}

Az AustenX, vagy röviden csak Austen szinten egy parser generátor. Az Austen jelenleg csak Java nyelven nyújt generátort, és az előzőekhez hasonlóan ez is reguláris nyelv alapján dolgozza fel a kódot.

Az Austin egy jar fájlként tölthető le és futtatható, aminek futtatásakor paraméterként kell megadni a célfájlt. Az Austen futtatható a jar fájlra kattintva duplán, ilyenkor egy egyszerű felhasználói kezelőfelület nyílik meg. Itt meg kell adni a forrásfájlok helyét, amiben a feldolgozáshoz szükséges adatok vannak, illetve a célfájl helyét, ahova a feldolgozott fájlok kerülnek. A forrásfájlok .austen vagy .austenx kiterjesztéssel kell, hogy rendelkezzenek. Az alábbiakban egy egyszerűbb példa látható a forrásfájlokra.

output austenx.general;

define library GeneralDefs {

//A bit over the top

BASE\_SYMBOLS = {

' '|

{\35-\38}|

{'\40'-'\46'}|

{\128-\65536}|'<'|'>'|'@'|'&'|'.'|'!'|'\_'|'['|']'|'-'|'+'|","|"."|"="|"$"|"+"|"\*"|"\\"|"/"|"\*"|"{"|"}"|"("|")"|":"|";"|"|"|"-"|"?"};

STRING\_ALPHA = {ALPHA|DIGIT|BASE\_SYMBOLS|"'"};

CHAR\_ALPHA = {ALPHA|DIGIT|BASE\_SYMBOLS|'"'};

ALPHA = {{'a'-'z'}|{'A'-'Z'}};

DIGIT = {'0'-'9'};

ALPHA\_HEAD = {ALPHA|'\_'};

ALPHA\_TAIL = {ALPHA\_HEAD|DIGIT};

NEWLINE = {"\r"|"\n"|"\r\n"};

}

library tokens BasicTokens {

String ID;

String STRING;

int INTEGER;

double DOUBLE;

String CHAR;

void LEFT\_CURLY;

void RIGHT\_CURLY;

void LEFT\_ROUND;

void RIGHT\_ROUND;

void SEMI\_COLON\_SYMBOL;

void COLON;

void OR\_SYMBOL;

void MINUS\_SYMBOL;

void AT\_SYMBOL;

void DOLLAR\_SYMBOL;

void COMMA\_SYMBOL;

void FULL\_STOP\_SYMBOL;

void ASSIGNMENT;

void QUESTION\_MARK\_SYMBOL;

void EXCLAMATION\_MARK\_SYMBOL;

void ESCAPE\_SYMBOL, AMPERSAND\_SYMBOL, PLUS\_SYMBOL, STAR\_SYMBOL, SLASH\_SYMBOL;

String OPERATION\_SYMBOL;

}

pattern library Basic tokens BasicTokens uses GeneralDefs {

pattern ( "!" ) { trigger EXCLAMATION\_MARK\_SYMBOL="!"; }

pattern ( "&") { trigger AMPERSAND\_SYMBOL&OPERATION\_SYMBOL="&"; }

pattern ( "@") { trigger AT\_SYMBOL="@"; }

pattern ( "." ) { trigger FULL\_STOP\_SYMBOL="."; }

pattern ( "{") { trigger LEFT\_CURLY="{"; }

pattern ( "}" ) { trigger RIGHT\_CURLY="}"; }

pattern ( "(" ) { trigger LEFT\_ROUND="("; }

pattern ( ")" ) { trigger RIGHT\_ROUND=")"; }

pattern ( ";" ) { trigger SEMI\_COLON\_SYMBOL=";"; }

pattern ( ":" ) { trigger COLON=":"; }

pattern ( "|" ) { trigger OR\_SYMBOL&OPERATION\_SYMBOL="|"; }

pattern ( "-" ) { trigger MINUS\_SYMBOL&OPERATION\_SYMBOL="-"; }

pattern ( "$" ) { trigger DOLLAR\_SYMBOL; }

pattern ( "," ) { trigger COMMA\_SYMBOL; }

pattern ( "=" ) { trigger ASSIGNMENT="="; }

pattern ( "?" ) { trigger QUESTION\_MARK\_SYMBOL="?"; }

pattern ( "+" ) { trigger PLUS\_SYMBOL&OPERATION\_SYMBOL="+"; }

pattern ( "\*" ) { trigger STAR\_SYMBOL&OPERATION\_SYMBOL="\*"; }

pattern ( "/" ) { trigger SLASH\_SYMBOL; }

pattern ( "\\" ) { trigger ESCAPE\_SYMBOL; }

pattern ( '"' $-start STRING\_ALPHA\* $+end '"' ) {

trigger STRING = range(low=start,high=end);

}

pattern ( "'" $-start CHAR\_ALPHA $+end "'" ) {

trigger CHAR = range(low=start,high=end);

}

pattern ( "'" $-start "\\" CHAR\_ALPHA $+end "'" ) {

trigger CHAR = range(low=start,high=end);

}

pattern ( ALPHA\_HEAD ALPHA\_TAIL\*) {

trigger ID;

}

pattern ( DIGIT+ '.' DIGIT+ ) {

trigger DOUBLE;

}

pattern ( DIGIT+ ) {

trigger INTEGER;

}

}

A fenti kód egy általános leírást tartalmaz, library-k szerint szervezve melyet felhasznál az alábbi kód, ami a parser generator pontos leírását tartalmazza.

import general="General.austenx";

/\*

AustenX example defintion file

Author Matthew Goode

\*/

output example.austen;

tokens Language extends general:BasicTokens {

void FUNCTION\_KEYWORD;

void IF\_KEYWORD, ELSE\_KEYWORD;

void WHILE\_KEYWORD;

void LET\_KEYWORD;

boolean BOOLEAN;

}

lex Language tokens Language uses general:GeneralDefs {

initial normal;

mode comment {

pattern ( NEWLINE ) { newline; }

pattern ( "\*/" ) { switch normal; }

pattern ( . ) { }

}

mode simpleComment {

pattern ( NEWLINE ) { newline; switch normal; }

pattern ( . ) { }

}

mode normal {

pattern ( "function" ) { trigger FUNCTION\_KEYWORD&ID="function"; }

pattern ( "let" ) { trigger LET\_KEYWORD&ID="let"; }

pattern ( "if" ) { trigger IF\_KEYWORD&ID="if"; }

pattern ( "while" ) { trigger WHILE\_KEYWORD&ID="while"; }

pattern ( "else" ) { trigger ELSE\_KEYWORD&ID="else"; }

pattern ( "true" ) { trigger BOOLEAN=true; }

pattern ( "false" ) { trigger BOOLEAN=false; }

library general:Basic();

pattern ( "//" ) { switch simpleComment; }

pattern ( NEWLINE ) { newline; }

pattern ( {"\t"|" "} ) { }

pattern ( "/\*" ) { switch comment; }

}

}

packrat Design tokens Language {

root File;

pattern File (

FileElement:elements\*

);

pattern FileElement ( -FunctionDef:function | -Statement:statement );

pattern FunctionDef ( FUNCTION\_KEYWORD @ID:name LEFT\_ROUND { Statement:p / COMMA\_SYMBOL } ? RIGHT\_ROUND LEFT\_CURLY

-Statement:statement\*

RIGHT\_CURLY

);

pattern Statement {

Expression ( -Expression:value SEMI\_COLON\_SYMBOL );

Block ( LEFT\_CURLY -Statement:statement RIGHT\_CURLY );

While ( WHILE\_KEYWORD LEFT\_ROUND -Expression:conditional RIGHT\_ROUND -Statement:body );

If ( IF\_KEYWORD LEFT\_ROUND -Expression:conditional RIGHT\_ROUND -Statement:trueBody { ELSE\_KEYWORD -Statement:falseBody } );

}

pattern Expression {

Assignment ( -Expression:first ASSIGNMENT -Expression:second );

Addition ( -Expression:first PLUS\_SYMBOL -Expression:second );

Subtraction ( -Expression:first PLUS\_SYMBOL -Expression:second );

Multiplication ( -Expression:first STAR\_SYMBOL -Expression:second );

Division ( -Expression:first SLASH\_SYMBOL -Expression:second );

FunctionCall ( @ID:functionName LEFT\_ROUND { -Expression:argument / COMMA\_SYMBOL } ? RIGHT\_ROUND );

Variable ( @ID:variableName );

String ( STRING:value );

Integer (INTEGER:value );

Boolean ( BOOLEAN:value );

Double ( DOUBLE:value );

}

pattern Parameter ( ID:type @ID:parameterName );

}

% TODO: Összeszedni 4-5 használhatót, és csinálni hozzá példákat!