Laboratorium Sztucznej Inteligencji

Sprawozdanie z Projektu

Przetwornik języka Atari Basic na język C

Autorzy:

Mariusz Galler
Stanisław Karlik
Radosław Szalski

Prowadzący:

dr inż. Adam Meissner

Politechnika Poznańska

Wydział Elektryczny
Informatyka, grupa I22
Rok 3, semestr 6

Poznań, 17 czerwca 2013

Spis treści

1 Opis zdania **2**

2 Założenia realizacyjne **2**

3 Podział prac **4**

4 Opis implementacji **5**
4.1 Obsługiwany podzbiór instrukcji języka Atari BASIC 5
4.1.1 Gramatyka w postaci Extended Backus-Naur Form (EBNF) 5
4.2 Wybrane predykaty zaimplementowane w przetworniku 6
4.3 Automatyczne testowanie 8

5 Użytkowanie i testowanie systemu **9**
5.1 Przykład 9

6 Tekst programu **12**

1 Opis zdania

Zadaniem systemu jest przetwarzanie kodu języka Atari BASIC na kod języka C. Program został napisany w języku Prolog przy użyciu konstrukcji gramatyk klauzul definiujących (DCG). Na wejściu program przyjmuje plik tekstowy zawierający poprawny kod napisany w języku Atari BASIC. Na wyjściu, program zwraca przetworzony kod w języku C, albo wiadomość o błędzie, jeżeli przetwarzanie się nie powiodło.

2 Założenia realizacyjne

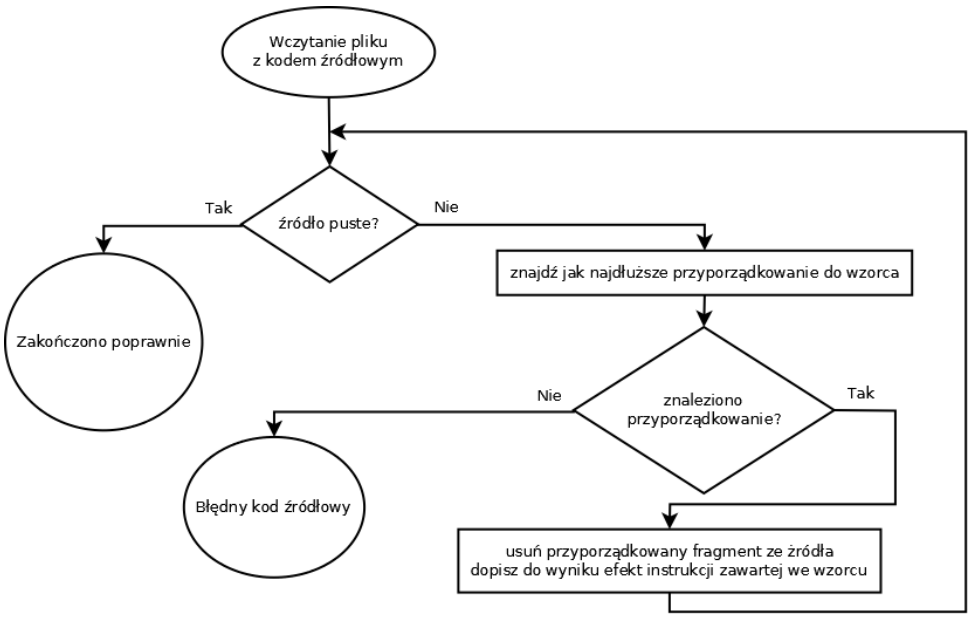
1. Program potrafi przetwarzać pewien podzbiór języka Atari BASIC. Wiele instrukcji byłoby bardzo kłopotliwych w implementacji, ze względu na znaczne różnice w składni i działaniu języków. Instrukcje języka Atari BASIC, do których się ograniczyliśmy, wymienione są w Tablicy 1. Mimo implementacji podzbioru instrukcji, napotkaliśmy problemy, które wynikają z faktu że zmienne liczbowe w BASICu nie muszą być deklarowane przed użyciem oraz z braku rozróżnienia między zmiennymi całkowitymi i zmiennoprzecinkowymi. Powoduje to, że kod wynikowy może zawierać błędy takie jak powtórne deklaracje zmiennych czy błędne typowanie, które trzeba poprawić ręcznie aby skompilować program. Problem ten można by rozwiązać poprzez zastosowanie *preprocesora* ale to rozwiązanie znacznie skomplikowałoby projekt.

2. Ogólny schemat działania translatora

   W programie używamy konstrukcji gramatyk klauzul definiujących do zapisywania schematu przetwarzania tekstu programu napisanego w języku Atari BASIC na tekst programu w języku C. Zapis ten jest swego rodzaju „lukrem syntaktycznym”, który ułatwia operowanie na listach będących sednem ww. konstrukcji. Wymaganymi argumentami każdej klauzuli są dwie listy różnicowe $S1$ i $S2$ (w zapisie listowym: `[S1|S2]` i `S2`). Pierwsza lista to zbiór symboli terminalnych, a druga — nieterminalnych. Stosowanie konstrukcji DCG pozwala ukryć te listy. Mamy możliwość korzystania z dodatkowych argumentów

służących na przykład do ukonkretniania fragmentów kodu programu. Aby korzystać z instrukcji języka Prolog w ciele reguł DCG, należy je umieścić między parą nawiasów klamrowych.

Wejście: Plik z kodem programu w języku BASIC.
Wyjście: Tekst kodu programu w języku C.



Rysunek 1: Ogólny schemat działania translatora

3. Językiem programowania, którego używaliśmy jest Prolog. Implementacja jest pod tym względem jednorodna — nie korzystamy z żadnych modułów/elementów napisanych w innym języku. W czasie pracy posługiwaliśmy się środowiskiem `SWI-Prolog` w wersji 6.2.6, które jest jednym z najpopularniejszych narzędzi do pisania programów w Prologu.

   Na Rysunku 1 widać ogólny schemat przetwarzania tekstu przez nasz program. Poniższy fragment kodu potrafi przetworzyć wyrażenia arytmetyczne. Wyrażenie takie składa się z dwóch operandów (stałej liczbowej, zmiennej liczbowej, albo wbudowanej funkcji operującej na liczbach) i operatora arytmetycznego. Dopuszczamy użycie dowolnej ilości znaków białych w odpowiednich miejscach. Wynik przetwarzania jest konkatenowany i zapisywany w zmiennej C.

Fragment klauzuli parsującej wyrażenia arytmetyczne

```
aexp(C) --> simple_aexp(C1), whitespace, aop(C2), whitespace, simple_aexp(C3),
 {concat_atom([C1, C2, C3], C)}.

simple_aexp(C) --> aconst(C).
simple_aexp(C) --> funcs(C).
simple_aexp(C) --> avar(C).

avar(C) --> alpha_char(C1), {atom_codes(C, [C1])}.
avar(I) --> "-", avar(I1), {concat_atom(['-', I1], I)}.
```

```
avar(I) --> "+", avar(I1), {concat_atom([I1], I)}.

aconst(I) --> number(I).
aconst(I) --> "-", number(I1), {concat_atom(['-', I1], I)}.
aconst(I) --> "+", number(I1), {concat_atom([I1], I)}.
```

3 Podział prac

Wszyscy członkowie zespołu posiadali zadania wspólne oraz indywidualne. W celu efektywnej pracy, podzieliliśmy się po równo instrukcjami języka Atari BASIC, które miały zostać zaimplementowane w programie. Do zadań wspólnych należało:

- analizowanie dokumentacji języka Atari BASIC w postaci podręcznika [1],
- bieżące opracowywanie testów automatycznych dla zaimplementowanych instrukcji,
- praca nad niniejszą dokumentacją.

Autor	**Podzadanie**
Mariusz Galler	Opracowanie gramatyki EBNF wybranego podzbioru języka Atari BASIC
	Przełożenie podstawowych elementów gramatyki EBNF na język Prolog np. `aexp`, `lexp`, `sexp`, `aop`...
	Implementacja instrukcji: `LET`, `VAL`, `DIM`, operatory porównania (`<`, `>`, `=`, `<=`, `>=`, `<>`), operatory logiczne (`AND`, `OR`, `NOT`), operatory arytmetyczne (`+`, `-`, `*`, `/`, `^`)
Stanisław Karlik	Zamiana tekstu wejściowego na małe litery w celu łatwiejszego dopasowania do reguł (DCG)
	Mechanizm etykietowania kodu wynikowego na podstawie numeru linii kodu źródłowego w celu umożliwienia obsługi instrukcji zależnych od numerów linii (np `GOTO`)
	Implementacja instrukcji: `INPUT`, `FOR`, `IF`, `ABS`, `CLOG`, `EXP`, `ASC`, `CHR$`, `LEN`, `GOTO`
Radosław Szalski	Opracowanie mechanizmu wczytywania plików i parsowania argumentów
	Opracowanie skryptów i metodologii uproszczonego automatycznego testowania zaimplementowanych funkcji
	Implementacja instrukcji: `REM`, `STOP`, `PRINT`, `INT`, `LOG`, `RND`, `SGN`, `SQR`, `ATN`, `COS`, `SIN`, `ADR`

Tablica 1: Spis podzadań poszczególnych autorów

4 Opis implementacji

1. W implementacji wykorzystujemy taką strukturę danych jak lista. Jest ona bardzo wygodna w użyciu i umożliwia szybki oraz prosty dostęp do składowych elementów. Na przykład, chcąc uzyskać pierwsze dwa elementy listy możemy zastosować zapis typu: `[H1, H2 | Tail]`, gdzie `H1` i `H2` to odpowiednio pierwszy i drugi element listy.

   Nasz program zapisany jest w notacji (DCG). Innymi słowy jest to „prologowa" reprezentacja formalnej notacji Context Free Grammar (CFG). Jednak zapis listy pozostaje niezmienny. Przykład użycia listy w naszym kodzie:

```
print_exps(C)--> exp(C1), separator(Sep), exps(C2), {concat_atom([C1, Sep, C2], C)}.
```

4.1 Obsługiwany podzbiór instrukcji języka Atari BASIC

4.1.1 Gramatyka w postaci EBNF

Poszczególne, uwzględnione instrukcje języka BASIC wymienione są w Tablicy 1. Dodatkowo, należało uwzględnić ogólną strukturę programu oraz konstrukcję stałych i zmiennych (zarówno liczbowych jak i znakowych). Program składa się z jednej lub więcej linii zakończonych znakami nowej linii. Poszczególne linie zaczynają się od numeru i zawierają jedno lub więcej (oddzielonych znakiem `:`) wyrażeń. Zmiennie znakowe kończą się znakiem `$` i podobnie jak zmienne liczbowe muszą zaczynać się od litery. Dodatkowo zdefiniowane zostały znaki białe (spacje, tabulatory itd.), liczby, cyfry i literały.

```
code = line, [ whitespace, newline, code ] ;
line = lineno, whitespace, statements ;

number = digit | digit excluding zero, { digit } ;
lineno = number ;

digit = "0" | digit excluding zero ;
digit excluding zero = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

whitespace = (" " | "\t"), whitespace | "" ;
newline = "\n" | "\r" | "\r\n" | "\n\r" ;

statements = statement, [ ":", statements ] ;

statement = "PRINT", exp, [ "," | ";", exp ] ;
statement = "INPUT", ( avar | svar ), { ",", ( avar | svar ) } ;
statement = [ "LET", whitespace, ] ( svar, "=", sexp | avar, "=", aexp ) ;
statement = "FOR", avar, "=", aexp, "TO", aexp, [ "STEP", aexp ], "NEXT", avar ;
statement = "IF", whitespace, lexp, whitespace, "THEN", whitespace, ( lineno | statements ) ;

exp = sexp | aexp ;
aexp = ( avar | fun | aconst ) | aexp, whitespace, aop, whitespace, aexp ;
sexp = svar | fun | sconst ;
lexp = "NOT", [ "(" ], lexp, [ ")" ] | [ "(" ], lexp, [ ")" ], ( "AND" | "OR" ), [ "(" ], lexp, [ ")

avar = alphabetic character, [ { alphanumeric } ] ;
```

```
svar = avar, "$" ;
mvar = avar, "(", ( number | avar | aexp ), ")" ;
var = avar | svar | mvar ;

fun = ( ( "ABS" | "CLOG" | "EXP" | "INT" | "LOG" | "RND" | "SGN" | "SQR" |
          "ATN" | "COS" | "SIN" ), "(", aexp, ")" ) | ( "DEG" | "RAD" ) ;

aconst = [ "-" ], { number } ;
sconst = '"' , { all characters - '"' } , '"' ;

aop = "+" | "-" | "*" | "/" | "^" ;
lop = "<" | ">" | "=" | "<=" | ">=" | "<>" ;

alphabetic character = "A" | "B" | "C" | "D" | "E" | "F" | "G"
                     | "H" | "I" | "J" | "K" | "L" | "M" | "N"
                     | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
                     | "V" | "W" | "X" | "Y" | "Z" ;

alphanumeric = alphabetic character | digit ;
all characters = ? all visible characters ? ;
```

4.2 Wybrane predykaty zaimplementowane w przetworniku

`basic2c/0`

Główny predykat programu. Przetwarza argumenty przekazane do programu, wczytuje plik, przetwarza go wg. reguł (DCG) i wyświetla kod wynikowy wraz z podsumowaniem.

```
basic2c :-
    unix(argv([_|B])),
    find_file_arg(B, Arg),
    get_first_elem(Arg, File_name),
    access_file(File_name, 'read'),
    !,
    write(user_error, 'Podano nazwe pliku do wczytania: '), write(user_error, File_name), nl,
    read_file_to_codes(File_name, C, []),
    upper2lower(C, D),
    program(Z, D, []),
    writeln(Z),
    write(user_error, 'Kod programu jest poprawny!'), nl.

basic2c :-
    write('error'),
    !,
    fail.
```

`find_file_arg(+Argumenty, -Nazwa_pliku)`

`+Argumenty` — lista argumentów przekazanych do programu
`-Nazwa_pliku` — atom zawierający nazwę przekazanego pliku

Predykat wyszukujący nazwę pliku z zadanych argumentów. Argumenty użytkownika (podane w czasie wywoływania programu z konsoli) oddzielone są od argumentów środowiska

(zapisane w kodzie) ciągiem znaków `--`.

```
find_file_arg([], _) :-
    writeln('Nie znaleziono argumentu.').
find_file_arg(['--'|File], File).
find_file_arg([_|T], File_arg) :-
    find_file_arg(T, File_arg).
```

`upper2lower(+Kod_programu, -Kod_programu_lowercase)`

`+Kod_programu` — lista kodów ASCII wczytanego tekstu programu
`-Kod_programu_lowercase` — lista kodów ASCII zawierająca wyłącznie małe litery tekstu programu

Predykat przepisujący kod programu, na znak po znaku, na małe litery.

```
upper2lower([], []).
upper2lower([X|T1], [Y|T2]) :-
    checkChar(X,Y),
    upper2lower(T1,T2).
```

`checkChar(+Znak, -Znak_przetworzony)`

`+Znak` — dowolny znak ASCII tekstu programu
`-Znak_przetworzony` — znak ASCII tekstu programu w postaci małej litery

Predykat zamieniający wartość kodu ASCII zadanego znaku, w zależności od tego czy jest wielką literą czy nie.

```
checkChar(X, Y) :-
        upper(X),
    Y is X + 32.
checkChar(X, Y) :-
    not(upper(X)),
    Y is X.
```

`upper(?Znak)`

`?Znak` — znak ASCII tekstu programu

Predykat sprawdzający, czy zadany znak jest wielką literą.

```
upper(X):-
    X > 64,
    X < 91.
```

4.3 Automatyczne testowanie

Z powodu czasochłonnego debugowania programu i zastosowania wielu reguł (DCG), zaimplementowany został mechanizm pozwalający na automatyczne uruchamianie testów. Dzięki niemu mogliśmy niezwłocznie po implementacji przetestować działanie reguł i co najważniejsze, skontrolować, czy czegoś przez przypadek nie naruszyliśmy w innej części programu.

Skrypt został napisany w języku `BASH` i uruchamiany jest z poziomu konsoli. Testowanie składa się z 2 etapów: przetwarzanie kodu i kompilacja do działającego pliku wykonywalnego. Tylko jeżeli te 2 etapy się powiodą, test jest uznawany za zaliczony. W momencie gdy problem występuje w trakcie przetwarzania samego kodu, program zapętla się, dlatego skrypt testujący posiada ustawioną wartość graniczą (*timeout*) po której kończy proces przetwornika i uznaje test za niezaliczony. Skrypt wyszukuje wszystkie pliki z rozszerzeniem `.basic` w folderze w którym się znajduje i podfolderach. Dla każdego znalezionego pliku, wywołuje przetwornik i kompiluje wynikowy kod w języku C. Na końcu wyświetlane są informacje o tym ile testów przeprowadzono w całości i ile się nie powiodło.

Skrypt realizujący automatyczne testowanie przetwarzania kodu

```
#!/usr/bin/env bash

ALL_TESTS=0
FAILED_TESTS=0

# Timeout.
declare -i timeout=2

for basic_file in $(find . -iname "*.basic")
do
    ALL_TESTS=$[$ALL_TESTS + 1]
    last_failed=0
    ( cmdpid=$BASHPID; (sleep $timeout; last_failed=1; kill $cmdpid &>/dev/null) & exec ../basic2c.pl

    gcc -w tmp.c &>/dev/null

    if [[ $? -gt 0 ]] || [[ $last_failed -eq 1 ]]
    then
        echo -e "\e[0;31m [Fail] \e[0m ${basic_file}: Failed."
        FAILED_TESTS=$[$FAILED_TESTS + 1]
        last_failed=0;
    else
        echo -e "\e[0;32m [OK] \e[0m ${basic_file}: Passed."
    fi

    # Cleanup
    rm a.out 2>/dev/null
    rm tmp.c 2>/dev/null
done

echo -e "\nTotal tests run: ${ALL_TESTS}"
echo "Failed tests: ${FAILED_TESTS}"

echo -en "\e[0m"
```

5 Użytkowanie i testowanie systemu

Aby dokonać translacji programu w języku BASIC do języka C należy wywołać program `basic2c`, jako pierwszy parametr podać plik zawierający program źródłowy oraz przekierować standardowe wyjście do pliku w którym chcemy zapisać wynik translacji:

```
sk@debian:~/git/basic2c$ ./basic2c.pl tests/przyklady/p2.basic > tmp.c
```

następnie program w C należy skompilować (np. za pomocą `gcc`):

```
sk@debian:~/git/basic2c$ gcc -o tmp tmp.c -lm
```

Teraz program można uruchomić i zweryfikować poprawność działania.

5.1 Przykład

Program demonstrujący działanie operatorów arytmetycznych, funkcji matematycznych, niektórych operacji na tekście, generatora liczb losowych, pętli FOR, instrukcji warunkowej IF oraz skoku GOTO:

```
Kod BASIC:
1 A=5
2 B=9
3 PRINT A
4 PRINT B
5 PRINT A+B
6 PRINT (A-B)
7 PRINT (A*B)
8 PRINT (A/B)
9 FOR I=0 TO 9 STEP 2
10 A=A+I
11 PRINT A
12 NEXT I
13 F = 7.62
14 U = -5.56
15 PRINT F
16 PRINT U
17 PRINT CLOG(F)
18 PRINT ABS(U)
19 PRINT ABS(F)
20 PRINT EXP(U)
21 PRINT LOG(F)
22 PRINT SGN(F)
23 PRINT SGN(U)
24 PRINT SQR(F)
25 PRINT ATN(U)
26 PRINT COS(F)
27 PRINT SIN(U)
28 R = RND(0)
29 PRINT R
```

```
30 R = INT(100*R)
31 PRINT R
32 IF R>=50 THEN PRINT "rwr"
33 IF R<50 THEN 35
34 GOTO 38
35 PRINT "rm"
36 GOTO 38
37 PRINT "nigdy"
38 PRINT "dalszy"
39 DIM X$(10)
40 X$ = "string1"
41 PRINT X$
42 DIM Y$(10)
43 Y$ = "string2"
44 PRINT Y$
45 C=VAL("35")
46 PRINT C
47 PRINT ASC("e")
```

Wynik translacji i wykonania, porównanie z wynikiem uzyskanym w emulatorze

```
sk@debian:~/git/basic2c$ ./basic2c.pl tests/przyklady/p1.basic > tmp.c
Podano nazwe pliku do wczytania: tests/przyklady/p1.basicKod programu jest poprawny!
sk@debian:~/git/basic2c$ gcc -o tmp tmp.c -lm
sk@debian:~/git/basic2c$ ./tmp   (obok wyniki uzyskane w emulatorze atari)
5.000000                         5
9.000000                         9
14.000000                        14
-4.000000                        -4
45.000000                        45
0.555556                         0.55555
5.000000                         5
7.000000                         7
11.000000                        11
17.000000                        17
25.000000                        25
7.620000                         7.62
-5.560000                        -5.56
0.881955                         0.881954971
5.560000                         5.56
7.620000                         7.62
0.003849                         3.84877652E-03
2.030776                         2.03077636
1.000000                         1
0.000000                         0
2.760435                         2.76043474
-1.392843                        -1.39284276
0.231853                         0.231853505
0.661776                         0.661777155
0.684674                         wynik zalezny od wylosowanej liczby
68.000000
rwr
dalszy                           dalszy
```

```
string1                         string1
string2                         string2
35.000000                       35
101.000000                      101
```

Wynikowy kod programu w języku C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
int main() {
srand(time(NULL));
l1:; float a = 5;
l2:; float b = 9;
l3:; printf("%f\n", (float) a);
l4:; printf("%f\n", (float) b);
l5:; printf("%f\n", (float) a+b);
l6:; printf("%f\n", (float) a-b);
l7:; printf("%f\n", (float) a*b);
l8:; printf("%f\n", (float) a/b);
l9:; int i;
for(i = 0; i <= 9; i += 2){
l10:; a = a+i;
l11:; printf("%f\n", (float) a);
l12:; }
l13:; float f = 7.62;
l14:; float u = -5.56;
l15:; printf("%f\n", (float) f);
l16:; printf("%f\n", (float) u);
l17:; printf("%f\n", (float) log10((double) f ));
l18:; printf("%f\n", (float) fabs(u));
l19:; printf("%f\n", (float) fabs(f));
l20:; printf("%f\n", (float) exp((double) u ));
l21:; printf("%f\n", (float) log(f));
l22:; printf("%f\n", (float) ((f > 0) ? 1 : ((f < 0) ? 0 : 0)));
l23:; printf("%f\n", (float) ((u > 0) ? 1 : ((u < 0) ? 0 : 0)));
l24:; printf("%f\n", (float) sqrt(f));
l25:; printf("%f\n", (float) atan(u));
l26:; printf("%f\n", (float) cos(f));
l27:; printf("%f\n", (float) sin(u));
l28:; float r = ((double) rand() / (RAND_MAX));
l29:; printf("%f\n", (float) r);
l30:; r = floor(100*r);
l31:; printf("%f\n", (float) r);
l32:; if( r>=50 ){
printf("%s\n", "rwr");
}
l33:; if( r<50 )
goto l35;

l34:; goto l38;
l35:; printf("%s\n", "rm");
```

```
l36:; goto l38;
l37:; printf("%s\n", "nigdy");
l38:; printf("%s\n", "dalszy");
l39:; char* x$;
l40:; x$ = "string1";
l41:; printf("%s\n", x$);
l42:; char* y$;
l43:; y$ = "string2";
l44:; printf("%s\n", y$);
l45:; float c = atoi("35");
l46:; printf("%f\n", (float) c);
l47:; printf("%f\n", (float) (int)"e"[0]);

}
```

6 Tekst programu

```
#!/usr/bin/swipl -q -t basic2c -s

% Przetwornik jezyka Atari BASIC na jezyk C
%
% Autorzy:
%   Mariusz Galler
%   Stanislaw Karlik
%   Radoslaw Szalski
%

% ------------------------------------------------------------------------------
% DCG
% ------------------------------------------------------------------------------

% Wszystko otaczane jest int main() {} aby mozna bylo automatycznie kompilowac.
program(Z) --> code(Za),
{concat_atom(['#include <stdio.h>\n#include
 <stdlib.h>\n#include <string.h>\n#include <math.h>\nint main()
  {\nsrand(time(NULL));\n', Za, '\n}'], Z)}.

% Kod - sklada sie z 1 lub wiecej linii zakonczonych znakami nowej linii.
% Przypadek, gdy program nie konczy sie pusta linia
code(Z) --> line(Z1), whitespace, {concat_atom([Z1], Z)}.
code(Z) --> line(Z1), whitespace, newline, code(Z2), {concat_atom([Z1, '\n', Z2], Z)}.
code(Z) --> line(Z1), whitespace, newline, {concat_atom([Z1, '\n'], Z)}.

% Zakomentowane linie, jezeli napotkamy '#', to nic innego nas nie interesuje ->
% ignorujemy do '\n'
line(Z) --> whitespace, comment_sign, anything(_), {concat_atom([''], Z)}.
line(Z) --> whitespace, lineno(_), whitespace, rem, whitespace, anything(A),
 {concat_atom(['// ', A], Z)}.
line(Z) --> whitespace, lineno(N), whitespace, statements(Z1),
 {concat_atom(['l', N, ':; ', Z1], Z)}.
```

```
% Komentarz wew. calkowite ignorowanie linii, rozny od instrukcji REM.
%comment_sign --> "#".
% Wlasciwy 'komentarz' w jezyku Atari BASIC, powoduje ignorowanie
% linii ale zostaje ona zawarta w kodzie wynikowym.
rem --> "rem".

statements(S) --> statement(S).
statements(S) --> statement(S1), whitespace, ":", whitespace,
 statements(S2), {concat_atom([S1, S2], S)}.

% Deklaruje domyslnie zmienne stringowe jako char*
statement(Z) --> "dim", whitespace, dimstmt(D), {concat_atom([D], Z)}.
statement(Z) --> "dim", whitespace, dimstmt(D1), whitespace, ",",
 whitespace, dimstmt(D2), {concat_atom([D1, '\n', D2], Z)}.
% Warunek IF
statement(Z) --> "if", whitespace, lexp(E), whitespace, "then",
 whitespace, statement(ZW), {concat_atom(['if( ', E, ' ){\n', ZW, '\n}'], Z)}.
statement(Z) --> "if", whitespace, lexp(E), whitespace, "then",
 whitespace, lineno(N), {concat_atom(['if( ', E, ' )\n', 'goto l', N, ';\n'], Z)}.

% INPUT
statement(Z) --> "input", whitespace, aexp(A),
 {concat_atom(['int ', A, ';\n', 'scanf("%d", &', A, ');'], Z)}.
statement(Z) --> "input", whitespace, sexp(A),
 {concat_atom(['scanf("%s", ', A, ');'], Z)}.

% GOTO
statement(Z) --> "goto", whitespace, lineno(N), {concat_atom(['goto l', N, ';'], Z)}.

% Petla FOR
% C nie dopuszcza deklaracji w for(), dlatego zmienna sterujaca jest zawsze wczesniej
% deklarowana. Jestesmy narazeni na blad redeklaracji zmiennej
% w przypadku ponownego uzycia tej samej w forze..
statement(Z) --> "for", whitespace, avar(A), whitespace, "=", aexp(B), whitespace,
 "to", whitespace, avar(A), whitespace, aoper(O), whitespace, aexp(C),
  {concat_atom(['int ', A, ';\n',
  'for(', A, ' = ', B, '; ', A, ' ', O, ' ', C, '; ', A, ' += 1', '){'], Z)}.
statement(Z) --> "for", whitespace, avar(A), whitespace, "=", aexp(B), whitespace, "to",
 whitespace, avar(A), whitespace, aoper(O), whitespace, aexp(C), whitespace, "step",
  whitespace, aexp(D), {concat_atom(['int ', A, ';\n',
   'for(', A, ' = ', B, '; ', A, ' ', O, ' ', C, '; ', A, ' += ', D, '){'], Z)}.
statement(Z) --> "next", whitespace, avar(_), {concat_atom(['}'], Z)}.
% wersja tylko z liczba w warunku (FOR I=0 TO 2 ...)
statement(Z) --> "for", whitespace, avar(A), whitespace, "=", aexp(B), whitespace, "to",
 whitespace, aexp(C), {concat_atom(['int ', A, ';\n',
  'for(', A, ' = ', B, '; ', A, ' ', '<=', ' ', C, '; ', A, ' += 1', '){'], Z)}.
statement(Z) --> "for", whitespace, avar(A), whitespace, "=", aexp(B), whitespace, "to",
 whitespace, aexp(C), whitespace, "step", whitespace, aexp(D),
 {concat_atom(['int ', A, ';\n',
 'for(', A, ' = ', B, '; ', A, ' ', '<=', ' ', C, '; ', A, ' += ', D, '){'], Z)}.
statement(Z) --> "next", whitespace, avar(_), {concat_atom(['}'], Z)}.

statement(Z) --> whitespace, "stop", anything(_),
 {concat_atom(['\nprintf("Stopped at line");'], Z)}.
% Sam print nie moze narzucac cudzyslowow, zaleza one od instrukcji wew.
```

```
statement(Z) --> "print", whitespace, print_exps(C), {concat_atom([’printf(’, C, ’);’], Z)}.
statement(Z) --> let_or_not, whitespace, avar(A), whitespace, "=", whitespace,
 aexp(B), {concat_atom([’float ’, A, ’ = ’, B, ’;’], Z)}.
% Funkcja INT - moze wystapic tylko w przypisaniu
statement(Z) --> let_or_not, whitespace, avar(A), whitespace, "=",
 whitespace, int_fnc(B), {concat_atom([’int ’, A, ’ = ’, B, ’;’], Z)}.
statement(Z) --> let_or_not, whitespace, svar(A), whitespace, "=",
 whitespace, sexp(B), {concat_atom([A, ’ = ’, B, ’;’], Z)}.

statement(Z) --> strcat(A), {concat_atom([A, ’;’], Z)}.

dimstmt(Z) --> svar(A), "(", aexp(_), ")", {concat_atom([’char* ’, A, ’;’], Z)}.
dimstmt(Z) --> avar(A), "(", aexp(L), ")", {concat_atom([’int ’, A, ’[(int)’, L, ’];’], Z)}.
dimstmt(Z) --> avar(A), "(", aexp(L1), ",", aexp(L2), ")", {concat_atom([’int ’, A, ’[(int)’, L1, ’]

print_exps(C) --> sexp(C1), {concat_atom([’"%s\\n", ’, C1], C)}.
print_exps(C) --> aexp(C1), {concat_atom([’"%f\\n", ’, ’(float) ’, C1], C)}.
print_exps(C) --> "(", whitespace, aexp(C1), whitespace, ")", {concat_atom([’"%f\\n",
 ’, ’(float) ’, C1], C)}.
print_exps(C) --> exp(C1), separator(Sep), exps(C2), {concat_atom([C1, Sep, C2], C)}.
% ogolny przypadek exp na koncu!
print_exps(C) --> exp(C1), {concat_atom([’"’, C1, ’"’], C)}.

% Funkcja INT
int_fnc(B) --> "int", whitespace, "(", whitespace, aexp(B1), whitespace, ")",
 {concat_atom([’floor(’, B1, ’)’], B)}.

exp(C) --> aexp(C).
exp(C) --> sexp(C).

anyvar(C) --> avar(C).
anyvar(C) --> svar(C).

aexp(C) --> aconst(C).
aexp(C) --> funcs(C).
aexp(C) --> avar(C).
aexp(C) --> simple_aexp(C1), whitespace, aop(C2), whitespace, simple_aexp(C3),
 {concat_atom([C1, C2, C3], C)}.
aexp(C) --> simple_aexp(C1), whitespace, "^", whitespace, simple_aexp(C2),
 {concat_atom([’pow((double)’,C1,’, (double)’, C2, ’)’], C)}.

simple_aexp(C) --> aconst(C).
simple_aexp(C) --> funcs(C).
simple_aexp(C) --> avar(C).

aop(R) --> "+", {concat_atom([’+’], R)}.
aop(R) --> "-", {concat_atom([’-’], R)}.
aop(R) --> "*", {concat_atom([’*’], R)}.
aop(R) --> "/", {concat_atom([’/’], R)}.

sexp(C) --> sfunc(C).
sexp(C) --> svar(C).
sexp(C) --> sconst(C).
sexp(C) --> strcat(C).
```

```
lexp(R) --> aexp(E), {concat_atom(['fabs( ', E, ' )'], R)}.
lexp(R) --> aexp(E1), whitespace, lop(O), whitespace, aexp(E2),
{concat_atom([E1, O, E2], R)}.
lexp(R) --> "(", whitespace, lexp(E), whitespace, ")",
 {concat_atom(['(', E, ')'], R)}.
lexp(R) --> "not(", whitespace, lexp(E), whitespace, ")",
 {concat_atom(['!(', E, ')'], R)}.
lexp(R) --> single_lexp(E1), whitespace, "and", whitespace, single_lexp(E2),
 {concat_atom([E1, ' && ', E2], R)}.
lexp(R) --> single_lexp(E1), whitespace, "or", whitespace, single_lexp(E2),
 {concat_atom([E1, ' || ', E2], R)}.

single_lexp(R) --> aexp(E), {concat_atom(['fabs( ', E, ' )'], R)}.
single_lexp(R) --> aexp(E1), whitespace, lop(O), whitespace,
 aexp(E2), {concat_atom([E1, O, E2], R)}.
single_lexp(R) --> "(", whitespace, lexp(E), whitespace, ")",
 {concat_atom(['(', E, ')'], R)}.
single_lexp(R) --> "not(", whitespace, lexp(E), whitespace, ")",
 {concat_atom(['!(', E, ')'], R)}.

lop(R) --> "=", {concat_atom(['=='], R)}.
lop(R) --> "<>", {concat_atom(['!='], R)}.
lop(R) --> ">", {concat_atom(['>'], R)}.
lop(R) --> "<", {concat_atom(['<'], R)}.
lop(R) --> "<=", {concat_atom(['<='], R)}.
lop(R) --> ">=", {concat_atom(['>='], R)}.

funcs(C) --> fun(C1), whitespace, "(", whitespace, aexp(C2), whitespace, ")",
 {concat_atom([C1, '(', C2, ')'], C)}.
funcs(C) --> "val", whitespace, "(", whitespace, sexp(C2), whitespace, ")",
 {concat_atom(['atoi(', C2, ')'], C)}.
% Funkcja SGN, traktowana osobno, poniewaz zastepujemy ja odpowiadajacym wyrazeniem w C.
funcs(C) --> "sgn", whitespace, "(", whitespace, aexp(C2), whitespace, ")",
 {concat_atom(['((', C2, ' > 0) ? 1 : ((', C2, ' < 0) ? 0 : 0))'], C)}.
% RND Wymaga inicjalizacji generatora, liczba w nawiasach to 'dummy variable'.
funcs(C) --> "rnd", whitespace, "(", whitespace, aexp(_), whitespace, ")",
 {concat_atom(['((double) rand() / (RAND_MAX))'], C)}.
% Wartosc adresu musi byc rzutowana najpierw na int.
funcs(C) --> "adr", whitespace, "(", whitespace, anyvar(C2), whitespace, ")",
 {concat_atom(['(int) &', C2], C)}.
% Funkcje log10 i exp w C nie przyjmuja parametrow typu int
funcs(C) --> "clog", whitespace, "(", whitespace, aexp(C2), whitespace, ")",
 {concat_atom(['log10((double) ', C2, ' )'], C)}.
funcs(C) --> "exp", whitespace, "(", whitespace, aexp(C2), whitespace, ")",
 {concat_atom(['exp((double) ', C2, ' )'], C)}.
funcs(C) --> "asc", whitespace, "(", whitespace, sexp(C2), whitespace, ")",
 {concat_atom(['(int)', C2, '[0]'], C)}.
%funcs(C) --> "len", whitespace, "(", whitespace, sexp(C2), whitespace, ")",
 {concat_atom(['strlen( ', C2, ' )'], C)}.
fun(C) --> "log", {concat_atom(['log'], C)}.
fun(C) --> "sqr", {concat_atom(['sqrt'], C)}.
fun(C) --> "atn", {concat_atom(['atan'], C)}.
fun(C) --> "cos", {concat_atom(['cos'], C)}.
fun(C) --> "sin", {concat_atom(['sin'], C)}.
fun(C) --> "abs", {concat_atom(['fabs'], C)}.
```

```
sfunc(C) --> "chr$", whitespace, "(", aexp(E), whitespace, ")", {concat_atom(['(char)', E], C)}.

avar(C) --> alpha_char(Ca), string(C2), {atom_codes(C1, [Ca]), concat_atom([C1, C2], C)}.
avar(C) --> alpha_char(C1), {atom_codes(C, [C1])}.
avar(I) --> "-", avar(I1), {concat_atom(['-', I1], I)}.
avar(I) --> "+", avar(I1), {concat_atom([I1], I)}.

aconst(I) --> number(I).
aconst(I) --> "-", number(I1), {concat_atom(['-', I1], I)}.
aconst(I) --> "+", number(I1), {concat_atom([I1], I)}.

svar(C) --> avar(C1), "$", {concat_atom([C1, '$'], C)}.
sconst(C) --> "\"", string_or_empty(C1), "\"", {concat_atom(['"', C1 ,'"'], C)}.

strcat(C) --> svar(C1), "(", whitespace, "len(", whitespace, svar(_),
 whitespace, ")", whitespace, "+", whitespace, "1", whitespace, ")",
  whitespace, "=", whitespace, svar(C3),
 {concat_atom(['strcat(', C1, ',', C3, ')'], C)}.

% Operatory arytmetyczne
aoper(O) --> "<", {concat_atom(['<'], O)}.
aoper(O) --> ">", {concat_atom(['>'], O)}.
aoper(O) --> "<=", {concat_atom(['<='], O)}.
aoper(O) --> "<=", {concat_atom(['<='], O)}.

% Separator ";" nie generuje zadnego znaku.
separator(Sep) --> whitespace, ";", whitespace, {Sep=''}.
% Separator "," generuje znak tab.
separator(Sep) --> whitespace, ",", whitespace, {Sep='\t'}.

lineno(N) --> integer_number(N).

% Dowolne napisy zlozone ze znakow alfanumerycznych.
string_or_empty(C) --> "", {concat_atom([], C)}.
string_or_empty(C) --> string(C).
string(C) --> chars(C).
chars(C) --> char(C1), chars(Rest), {concat_atom([C1, Rest], C)}.
chars(C) --> char(C).
char(C) --> [C1], {code_type(C1, alnum), atom_codes(C, [C1])}.
alpha_char(C) --> [C], {code_type(C, alpha)}.

% Liczby
number(N) --> integer_number(N).
number(N) --> float_number(N).

% Liczby calkowite, zlozone z jednej lub wiecej cyfr.
integer_number(I) --> digit(I1), integer_number(Rest), {concat_atom([I1, Rest], I)}.
integer_number(I) --> digit(I).
digit(I) --> [I1], {code_type(I1, digit), atom_codes(I, [I1])}.

% Liczby float
float_number(I) --> digit(C), ".", integer_number(R), {concat_atom([C, '.', R], I)}.
float_number(I) --> digit(Ca), integer_number(Cb), ".", integer_number(R),
 {concat_atom([Ca, Cb, '.', R], I)}.
```

```
% Biale znaki.
whitespace --> " ", whitespace.
whitespace --> "\t", whitespace.
whitespace --> "".

% Znaki nowej linii.
newline --> "\n".
newline --> "\r".
newline --> "\r\n".
newline --> "\n\r".

% Unifikuje sie z dowolnym ciagiem, ale nie moze pochlaniac znakow nowej linii.
anything(A) --> [A1], anything(A3), {atom_codes(A2, [A1]), concat_atom([A2, A3], A)}.
anything(A) --> [A1], {atom_codes(A2, [A1]), concat_atom([A2], A)}.

let_or_not --> "let".
let_or_not --> "".

% ------------------------------------------------------------------------------
% Utilities
% ------------------------------------------------------------------------------

basic2c :-
    unix(argv([_|B])),
    find_file_arg(B, Arg),
    Arg = [File_name|_],
    access_file(File_name, 'read'),
    !,
    write(user_error, 'Podano nazwe pliku do wczytania: '),
    write(user_error, File_name), nl,
    read_file_to_codes(File_name, C, []),
    upper2lower(C, D),
    program(Z, D, []),
    writeln(Z),
    write(user_error, 'Kod programu jest poprawny!'), nl.

% Druga klauzula wywolywana w momencie, gdy podano bledny plik.
%Powoduje wyswietlenie bledu bez zapetlenia.
basic2c :-
    write('error'),
    !,
    fail.

% Konwersja wielkich liter do malych
checkChar(X, Y) :-
    upper(X),
    Y is X + 32.
checkChar(X, Y) :-
    not(upper(X)),
    Y is X.

upper2lower([], []).
upper2lower([X|T1], [Y|T2]) :-
    checkChar(X,Y),
```

```
    upper2lower(T1,T2).

upper(X):-
    X > 64,
    X < 91.

% Znak '--' oddziela argumenty Prologa od argumentow uzytkownika.
% Jezeli glowa jest '--' to ogonem jest lista argumentow.
find_file_arg([], _) :-
    writeln('Nie znaleziono argumentu.').
find_file_arg(['--'|File], File).
find_file_arg([_|T], File_arg) :-
    find_file_arg(T, File_arg).

% vim:filetype=prolog
```

Literatura

[1] ATARI Software Support Group, *ATARI 400/800 BASIC Reference Manual*. ATARI, Sunnyvale, CA, USA, 2nd Edition, 1980.