Arkadiusz Danilecki inf40703
Konrad Jurkiewicz inf40726

Poznan,27.05.97

PROJEKT Z JĘZYKOW FORMALNYCH

Translator z LISP'u na C

recenzenci :

Dr.inz Jolanta Cybulka
Dr.inz Beata Jankowska

Dokumentacja.

I. Analiza wymagań

1. Cel i przeznaczenie programu.

Niniejszy projekt miał na celu konstrukcje oprogramowania wykonującego translacje z czystego LISP'u na C. LISP posiada dobrze zdefiniowana i niezbyt rozbudowana gramatykę, dzięki czemu można go w miarę łatwo tłumaczyć na inne języki, takie jak właśnie m.in. C, mimo rozbieżności pomiędzy programowaniem deklaratywnym, a proceduralnym. Program powstał w ramach zajęć z języków formalnych w celach doświadczalnych, czyli nie jest produktem komercyjnym. Używanie go jest nieodpłatne, wyłączając modyfikację kodu źródłowego, która powinna być uzgodniona z autorami.

2. Założenia dotyczące użytkowników

Przyjmujemy założenie, ze program nie musi być specjalnie "przyjazny" wobec użytkowników Zakładamy ze użytkownicy znają całkiem nieźle LISP'a i C, stad będą oni podawać w miarę poprawne zadania dla naszego translatora. Poza tym (w związku z tym, iż LISP jest językiem używanym przede wszystkim na uczelniach) przyszli użytkownicy rekrutują się z grona informatyków, którzy większa uwagę zwracają na funkcjonalność programu.

3. Zadanie wykonywane przez program.

Program, który przygotowaliśmy, wykonuje translacje plików z zapisami funkcji w LISPie na równoważne im funkcje w C oraz dołącza kod prostego interpretera tychże funkcji.
Przykład zadania :

Przetłumaczyć program w LISPie na C (plik wejściowy):

```
label[lewy; lambda [[x];car[x]]]
```

Przykładowy wynik (plik wyjściowy) :

```
lewy(lista x) {return car(x) } // gdzie 'lista' jest predefiniowanym typem danych
                               // opisującym strukturę danych taka jaka maja dane
                               // LISPowe, a 'car' powinno być dołączane jako
                               // funkcja standardowa.
```

4. Wymagania sprzętowe, objętość systemu itd.

Przyjęliśmy założenie ze tak prosty program powinien chodzić na każdym komputerze klasy PC wyposażonym w twardy dysk i/lub stacje dysków. W fazie projektowania przewidywaliśmy ze system nie będzie zbyt obszerny. Zdecydowaliśmy, ze powinien składać się z translatora oraz biblioteki standardowych funkcji LISP-u napisanych w C.

Program napisany został w języku C++ , z wykorzystaniem YACC-a oraz LEX-a. Skompilowany został z wykorzystaniem rozkazów procesora 80486, dlatego tez do jego uruchomienia potrzebny jest procesor 486 DX lub lepszy, aczkolwiek ponowne przekompilowanie źródła na rozkazy "słabszego" procesora powinno rozwiązać problemy z niekompatybilnością.

5.Komunikacja z użytkownikiem

Początkowo przewidywaliśmy system komunikujący się z użytkownikiem przy pomocy okienek i systemu prostych menu, ale w trakcie prac doszliśmy do wniosku ze byłoby to tylko niepotrzebnym zwiększeniem rozmiaru kodu, zrezygnowaliśmy wiec z tego pomysłu. Jednakże musieliśmy wprowadzić pewna interakcje z użytkownikiem aby mógł on wywoływać przetłumaczone funkcje z podanymi przez siebie parametrami (w LISPie nie ma odpowiednika funkcji main!).

6. Sytuacje wyjątkowe i błędy

Do błędów z którymi nasz program musi sobie radzić, należą :

a) błędy leksykalne pliku źródłowego w LISPie
b) braki pamięci w trakcie translacji
c) błędy podczas zapisu i odczytu danych
d) błędy "kontekstowe", tj. np. nieodpowiednia liczba argumentów itd.

Nie na wszystkie błędy nasz program musi reagować Ponieważ chcieliśmy by nasz program mial maksymalnie prosta strukturę, zrezygnowaliśmy z pełnej analizy kontekstowej, wychodząc z założenia ze program będzie tłumaczył poprawne pliki operujące na danych jednorodnego typu.
Podczas pozostałych błędów program powinien zakończyć prace lub zawiesić ja w oczekiwaniu na decyzje użytkownika. Zdecydowaliśmy się na to pierwsze rozwiązanie. Program informuje w tym wypadku użytkownika o zaistniałym błędzie.

7. Harmonogram i inne.

Zdecydowaliśmy się najpierw gruntownie przemyśleć cały problem i rozwiązać wszystkie wątpliwości zanim przystąpimy do pisania programu. Przewidywaliśmy ze około 90% czasu realizacji poświecimy na projektowanie, w rzeczywistości podczas wykonywania i zapisywania algorytmów pojawiły się nieoczekiwane problemy, które spowodowały zmianę tych proporcji.

II. Dokumentacja użytkowa

1.Sposob translacji.

Nie zalecamy powtórnej kompilacji programu lisp2c, aczkolwiek dołączamy wszystkie potrzebne do tego biblioteki i pliki źródłowe W celu kompilacji należy stworzyć wykorzystać najlepiej lisp2c.prj - projekt. Do stworzenia lisp2c.exe wymagane są :

- biblioteka chlexcpp.lib (lub inna biblioteka YACC-a zgodna z wykorzystywanym modelem pamięci),

- pliki lisp.y (ylisp.cpp),lisp.l (llisp.cpp), lisp2c.cpp wraz z niezbędnymi nagłówkami.
- plik pars.bat uruchamiający generację parsera.
- plik stdlisp.c.

Po otrzymaniu programu wykonywalnego należy go uruchomić z nazwa pliku do translacji jako parametrem, np.:

```
lisp2c ample
```

Jako plik wynikowy otrzymamy plik lispout.c oraz nagłówek lispout.h. Należy je skompilować, przy czym w katalogu w którym są te pliki musi się znajdować plik stdlisp.c (definicje standardowych funkcji typu car).

2. Uruchomienie programu.

Po skompilowaniu otrzymamy plik lispout.exe. Należy go uruchomić . Pojawi się np. następujący ekran:

```
| 0 - silnia |
  | 1 - car |
  | 2 - cdr |
     ...
  Wpisz numer funkcji
```

Należy wpisać odpowiedni numer, np. dla funkcji silnia 0, i nacisnąć enter. Pojawia się zapytania o kolejne argumenty tej funkcji, za każdym razem należy je wpisać i nacisnąć enter. Program , jeżeli argumenty były poprawnie wpisane, wyświetli wyniki i zapyta czy chcemy kontynuować Jeżeli nie chcemy, wciskamy q, w przeciwnym razie naciskamy dowolny inny klawisz.

Należy pamiętać ze program nie jest "idioto odporny" i zakłada pewne minimum dobrej woli z strony użytkownika, tzn. należy wpisywać poprawne argumenty gdyż błędy interpretera NIE BEDA automatycznie poprawione.

III. Projekt wstępny

W tym rozdziale omówimy rożne pomysły na rozwiązanie nasuwających się nam problemów;
dokładny zaś opis zastosowanych rozwiązań zostanie podany w rozdziale następnym

1. LISP.

LISP jest praktycznym zastosowaniem rachunku lambda w informatyce. Jest to język nieimperatywny, operujący na listach, reguły obliczania funkcji i wartościowania form są podobne jak w rachunku lambda.

Wyrażenia tego rachunku wyglądają np. tak:

```
lambda x lambda y (x z)
```

Obliczenia są dokonywane dzięki redukowaniu tych wyrażeń przy pomocy pewnych reguł, takich jak : reguła przemianowywania zmiennych:

```
lambda x    ==  lambda y
```

reguła nazywana podstawowa reguła redukcji:

```
lambda e[x]M -> e[M/x]
```

gdzie M jest wyrażeniem wolnym z względu na zastąpienie nim zmiennej x w wyrażeniu e[x],

reguła pomocnicza:

```
lambda x (F x) -> F
```

Przykładowy proces obliczeń :

```
(lambda x ((+1)x)5) -> ((+1)5) -> 6
```

Dane LISP-u są to dane atomowe oraz nieatomowe. ATOMY, zapisywane jako ciągi dużych liter i cyfr zaczynających się od dużej litery, są niepodzielnymi i podstawowymi danymi LISPu. Z nich składają się LISTY i pary kropkowe. Para kropkowa jest to struktura która składa się z dwóch elementów, lista zaś jest to struktura której prawy element jest lista, lista jest tez NIL.

LISP posiada ściśle określona gramatykę Poza tym LISP był pierwszym językiem który był interpretowany przez program w LISPie. Jego główna cześć przytaczam niżej , jest to próbka której używaliśmy do testowania poprawności programu :

```
label[evalquote;
  lambda[[fu;x];
    apply[fu;x;NIL]]]
label[apply;
  lambda[[fu;x;a];
    [atom[fu] -> [eq[fu;CAR] -> car[car[x]];
                  eq[fu;CDR] -> cdr[car[x]];
                  eq[fu;CONS] -> cons[car[x];car[cdr[x]]];
                  eq[fu;ATOM] -> atom[car[x]];
                  eq[fu;EQ] -> eq[car[x];car[cdr[x]]];
                  T -> apply[eval[fu;a];x;a]];
     eq[car[fu];LAMBDA] -> eval[car[car[cdr[fu]]];pairlis[car[cdr[fu]];x;a]];
     eq[car[fu];LABEL] -> apply[caddr[fu];x;cons[cons[car[cdr[fu]];
                                                 car[cdr[cdr[fu]]]];
                                                a]]]]]
label[eval;
  lambda[[fo;a];
    [atom[fo] -> cdr[assoc[fo;a]];
     atom[car[fo]] -> [eq[car[fo];QUOTE] -> car[cdr[fo]];
                       eq[car[fo];COND] -> evcon[cdr[fo];a];
                       T -> apply[car[fo];evlis[cdr[fo];a];a]];
     T -> apply[car[fo];evlis[cdr[fo];a];a]]]]
label[evcon;
  lambda[[c;a];
    [null[l] -> NIL;
     T -> cons[eval[car[l];a];evlis[cdr[l];a]]]]]
```

W LISPie są możliwe poza tym następujące, dziwne na pierwszy rzut oka przypadki :

```
label[nazwa;nazwa_1]
   - przemianowywanie funkcji
lambda[[x];forma]
   - funkcja bez nazwy !
lambda[[x];forma][x]
   - definicja i od razu wywołanie funkcji.
```

Poza tym w LISPie jest dozwolone by w wyniku wartościowania form lub zmiennych powstawały wartości nieokreślone Rozważmy następujący przykład :

```
lambda[[x];atom[x]->car[x];car[x]->F]
```

Co się stanie jeżeli x nie jest atomem ? Wartość funkcji wtedy znajduje się przez wartościowanie reszty wyrażenia warunkowego. Jeżeli car[x] będzie atomem "T" wtedy cale wyrażenie będzie miało wartość F (fale) jeżeli F (fale) nie będzie miało żadnej określonej wartości, a jeżeli car[x] zwróci inne atomy lub nawet dane nieatomowe, wtedy cale wyrażenie będzie miało wartość nieokreślona

Jak widać wynika z tego wiele problemów W C nie może istnieć funkcja bez nazwy, przemianowywanie funkcji można zaimplementować jak zamianę adresów funkcji lub ich wywoływanie, nie można zagnieżdżać definicji funkcji, nie można wywoływać funkcje od razu przy jej definicji, wreszcie nie może istnieć zmienna która raz ma wartość określona a inny raz nieokreślona Dodatkowym problemem jest to , ze nie wiemy od której funkcji w LISPie zacząć wykonywanie programu, a przecież C wymaga funkcji main().

2. Pomysły.

Pierwsze co nam się nasuwa to wykorzystanie oprócz C jakichś gotowych narzędzi wspomagających analizę leksykalna. Musimy napisać bowiem parser oraz analizator leksykalny. Na analizator leksykalny zrzucimy prace związana z decydowaniem czy dane słowo w tekście jest zmienna, atomem czy tez czymś innym. Odrzuca on tez już od razu najbardziej oczywiste błędy. Parser sprawdzi poprawność gramatyczna. Nasunął nam się pomyśl by plik wejściowy badać w czterech przebiegach parsera:

pierwszy przebieg : polegałby na sprawdzeniu poprawności gramatycznej,

drugi przebieg : zbierałby informacje na temat funkcji i ich argumentów, decydowałby o tym czy 'nazwa' ma być traktowana jako funkcja czy jako zmienna,

trzeci przebieg : badałby zależności miedzy funkcjami, przy okazji dokonywałby analizy kontekstowej,

czwarty przebieg: definiowałby funkcje i dokonywałby właściwej translacji na C przy wykorzystaniu wcześniej zebranych informacji.

Oczywiście widać na pierwszy rzut oka ze liczbę tych przebiegów da się zmniejszyć Drugim problemem stały się dla nas zmienne lispowskie. Należałoby wszystkie je zbierać na jakąś strukturę a następnie dekldeklarowaćo zmienne globalne. Po analizie jednak wielu programów doszliśmy do wniosku ze jest to niepotrzebne. W wszystkich analizowanych przez nas programach w LISPie jedyne zmienne jakie występowały, były to argumenty przekazywane do i z funkcji.

Trzeci problem polega na tym, ze każda funkcja LISPowska może być wywoływana w dowolnym miejscu , nawet przed jej zdefiniowaniem Tak wiec funkcje należy traktować jako cały czas dostępne niezależnie od tego w jakiej kolejności były deklarowane. Kolejny problem to w jaki sposób reprezentować dane LISP-u oraz funkcje standardowe.

Pierwsze rozwiązanie które się narzuca jest nastepujace :

zdefiniować strukturę dana_lispowska :

```
struct dana_lispowska
  {
 int status;
  union dana;
  }
```

Gdzie : status- informacja na temat struktury (do którego pola unii można się odwoływać)

dana zaś jest unia postaci :

```
union dana
       {
        atom _atom;
        struct para_kropkowa;
        }
```

Atom jest typem char * tyle ze ograniczonym do dużych liter i cyfr, natomiast para_kropkowa ma postać :

```
struct para_kropkowa
        {
        struct dana_lispowska lewy;
        struct dana_lispowska prawy;
        }
```

Wtedy np. funkcja car(x) wyglądałaby tak :

```
struct dana_lispowska car(struct dana_lispowska x)
```

```
        {
        return x.lewy;
        }
```

Jednakże z taka reprezentacja i dokładnym odwzorowywaniem LISP-u są związane
pewne problemy, począwszy od pamieciozernosci i pracochłonności zarządzania tymi
strukturami, a skończywszy na konieczności deklarowania odpowiednio wcześniej wszystkich stałych lispowskich użytych w programie. Tak wiec zdecydowaliśmy ze zastosujemy inne rozwiązanie (patrz dalej).

IV Opis realizacji

.

1.Zastosowane środowisko

Do stworzenia programu użyliśmy lexa, yacca i kompilatora C++ firmy Borland. Program składa się z plików :

- -lisp2c.cpp w którym znajduje się funkcja main i od którego zaczyna się wykonywanie całości
- -ylisp.y zawierający parser i główna cześć programu,
- -llisp.l zawierający analizator leksykalny,
- -stdlisp.c zawierający standardowe funkcje LISP-u,
- -ylisp.h plik nagłówkowy parsera,
- -llisp.h plik nagłówkowy scannera,
- -lisp2c.h plik nagłówkowy programu głównego,
- -lista.h plik nagłówkowy zawierający definicje listy używanej przy parsingu

Parser jest tylko jednoprzebiegowy!
Rezygnujemy z wiernego odwzorowania danych lispowskich na C i przyjmujemy ze wszystkie funkcje będą operować na stringach (char*).

2.STDLISP.C

Plik ten zawiera standardowe funkcje LISP-u, przy czym są one jedyne przykładami
jak można je zrealizować
Co więcej, najlepiej by było gdyby napisał je jakiś doświadczony programista w LISP-ie. Błędy w działaniu większych programów, jeżeli się pojawia, wynikają zazwyczaj z budowy właśnie tych funkcji standardowych a nie z zlej translacji (patrz "Opis testowania"). W końcu zadaniem naszym było napisanie translatora a nie całego środowiska

```
        <->
#include <string.h>
#include <alloc.h>
#include <ctype.h>
#include <_null.h>
        <->
```

Deklaracje funkcji używanych w pliku. ctype.h miedzy innymi zwiera definicje isupper(), string zawiera funkcję operujące na łańcuchach, alloc zawiera funkcje zajmujące się przydziałem pamięci, _null.h miedzy innymi zawiera definicje NULLa.

```
          <->
int evalint(char* a)
{
 int eff;
 if (!strcmp(a,"T")) eff=1; // jeżeli argumentem jest atomem "prawda"
 else
  if (!strcmp(a,"F")) eff=0;// jeśli argument jest atomem "fałsz"
  else
  {
```

```
    printf("Undetermined variable!");// w innym przypadku, wartość nieokreślona
    exit(0);
  }
 return eff;
}
        <->
 Funkcja ta zamienia atomy "T" oraz "F" na zmienne typu int, wartości
odpowiednio 1 i 0. Funkcja ta jest wykorzystywana potem bardzo często .
        <->

char* null(char* a)
{
  if ((!strcmp(a,"NIL")) || (!strcmp(a,"()"))) return "T"; //jeśli jest NULLEM
  else return "F";
}
<->
Funkcja null sprawdza czy argument jest atomem o wartości NIL.
<->

char* atom(char* a)
{
 if (!strcmp(a,"()")) return "T"; //jeśli jest NIL
 else
  if (isupper(a[0]))  //jeśli pierwsza litera jest duża
  {
   for(int i=1;a[i]!='\0';i++)
   {
    if (isupper(a[i]) || isdigit(a[i]));  //jeśli są same małe litery i cyfry
    else return "F";//jesli zdarzy się choć jedna mała litera
   }
   return "T";
  }
  else return "F";
}
<->
 Funkcja atom sprawdza czy argument jest atomem.
 <->
 char* cons(char* a,char* b)
{
 char* newstr;

 newstr=(char*) malloc(strlen(a)+strlen(b)+4);//utwórz nowy łańcuch
 sprintf(newstr,"(%s\.%s)",a,b); //skopiuj do niego wartość (a.b)
 return newstr;
}
<->
Funkcja cons tworzy parę kropkowa z argumentów
<->
char *car(char *_a)
{
char *tmp='\0';
int a=0,k=0;

if (!strcmp(atom(_a),"F"))
{
tmp=(char *)malloc(sizeof(char));
tmp[1]='\0';
if (_a[1]=='(') k=1;
for (int i=1;_a[i]!='\0';i++)
    {
    if ((_a[i]=='.')&&(k==0))
```

```
        {
   return tmp;
        }
    else if ((_a[i]=='(')&&(k>0)&&(i>1))
        {
        k++;
        tmp[a++]=_a[i];
        tmp[a]='\0';
        }
    else if ((_a[i]==')')&&(k>1))
        {
        k--;
        tmp[a++]=_a[i];
        tmp[a]='\0';
        }
    else if ((_a[i]==')')&&(k==1))
        {
        tmp[a++]=_a[i];
        tmp[a]='\0';
        return tmp;
        }
    else if ((_a[i]==' ')&&(k==0))
        {
        return tmp;
        }
    else if ((_a[i]==')')&&(k==0))
        {
        return tmp;
        }
    else
        tmp[a++]=_a[i];
        tmp[a]='\0';
    }
return tmp;
}
else
 {
  printf("Non atomic variable required for function (car)!");
  exit(0);
 }

}

<->
Funkcja car zwraca pierwszy element pary.
<->
char *cdr(char *_a)
{
char *tmp='\0';
int a=0,k=0,i=0;

if (!strcmp(atom(_a),"F"))
{
tmp=(char *)malloc(sizeof(char));
tmp[1]='\0';
if (_a[1]=='(') k=1;
for (i=1;_a[i]!='\0';i++)
    {
    if ((_a[i]=='.')&&(k==0))
        {
        break;
```

```
        }
    else if ((_a[i]=='(')&&(k>0)&&(i>1))
        {
        k++;
        }
    else if ((_a[i]==')')&&(k>1))
        {
        k--;
        }
    else if ((_a[i]==')')&&(k==1))
        {
        break;
        }
    else if ((_a[i]==' ')&&(k==0))
        {
        break;
        }
    else if ((_a[i]==')')&&(k==0))
        {
        break;
        }
    else
      ;
    }
i++;
if (_a[i]==')')
        {
        tmp[0]='N';
        tmp[1]='I';
        tmp[2]='L';
        tmp[3]='\0';
        }

for (a=0;_a[i+1]!='\0';i++)
{
tmp[a++]=_a[i];
tmp[a]='\0';
}
return tmp;
}
else
 {
  printf("Non atomic variable required for function (cdr)!");
  exit(0);
 }

}


<->
Funkcja cdr zwraca drugi element pary, działa analogicznie jak car.
<->

char* eq(char* a,char* b)
{
 if ((!strcmp(atom(a),"T")) && (!strcmp(atom(b),"T"))) //jeśli oba argumenty są
atomami
 {
  if (!strcmp(a,b)) return "T"; //jeśli strcmp==0
  else return "F";
 }
```

```
 else
 {
  printf("Atomic variables required for function (eq)!");
  exit(0);
 }
}
```

<->

Eq sprawdza czy argumenty, atomy, są sobie równe Strcmp zwraca zero gdy oba jej argumenty są równe, wartość większa od zera gdy pierwszy jest większy i mniejsza od zera gdy drugi jest większy

3. Ogólna idea.

Zdecydowaliśmy się wykorzystać dwa rzadko kiedy wykorzystywane mechanizmy C (rzadko przynajmniej wśród naszych przyjaciół). Pierwszy z nich, to operator warunkowy ?:
wyrażenie wart?a:m ma wartość a jeżeli wart jest rożne od zera, a m gdy jest wprost przeciwnie. Jak widać wyrażenie to idealnie nadaje się do zaimplementowania warunków
LISP-u. Warunek car[x]->car[y] odzwierciedlamy jako :

```
(evalint(car(x))?car(y):("NIL");
```

Drugi mechanizm to podawanie funkcji jako argumentu innej funkcji, czyli np.:
car(car(car(car(x)))). Jak widać ten zapis również przypomina lispowy wiec idealnie nadaje się
do zastosowania w naszym translatorze.

Oto zaś schemat działania naszego translatora:

Na początku każdego pliku wstawiamy linijki:

```
#include <stdio.h>
#include <stdlib>
#include "lispout.h"
#include "stdlisp.c"
```

Plik nagłówkowy stdlisp.c został opisany wcześniej i zawiera definicje standardowych funkcji
LISPu. Zakładamy, ze użytkownik może sam napisać sobie te funkcje i podmienić je
pod nazwa stdlisp.c. Plik nagłówkowy lispout.h będzie zawierać deklaracje wszystkich
funkcji użytych w programie.
Wszystkie funkcje są typu char *. gdyż wszystkie operują na danych lispowych.
Argumenty mogą być typu char * albo tez może ich nie być wcale (void). Ogólnie
funkcja składa się z nazwy, wołania argumentów oraz ciała w którym znajduje się od razu
słowo return i zwracana wartość, czyli ogoleni funkcja ma postać :

```
nazwa(arg) {return wartość;}
```

W YACC-u do przekazywania wartości służy unia, jako której składnik zdefiniowaliśmy pole, z którym to związane są dwie wartości : text, oraz ivalue. Dla funkcji pole text zawiera ich nazwę, natomiast dla form zawiera treść formy.

Pole ivalue zawiera dla funkcji liczbę jej argumentów, dla innych tokenow informacje obecnie nieistotne.

Wyszukujemy wszystkie funkcje postaci lambda[[arg];forma] gdzie arg i forma mogą być ciągami pustymi. Funkcjom tym nadawane są nazwy USRDEFN%nr gdzie %nr jest to numer funkcji. Deklaracje funkcji zgrywane są do pliku "lispout.h" natomiast do pliku "lispout.c" zgrywane są ich ciała. Ciało funkcji otrzymujemy poprzez uzgadnianie pola FORMA - patrz niżej. Wyżej (znaczy w YACC-u do formy bardziej skomplikowanej) jako wartość pola jest przekazywana liczba argumentów świeżo zdefiniowanej funkcji USRDEFN oraz jej nazwa.
Czyli, dla lambda[[a;b];[...nieważne.]]otrzymamy :

w pliku lispout.h :
```
char *USRDEFN0(char *,char *);
```
w pliku lispout.c :
```
char *USRDEFN(char *a,char *b) {return <forma> ;}
```

Wyżej przekazujemy : $$.text = "USRDEFN0", $$.ivalue = liczba argumentów, czyli $3.ivalue- tutaj dwa.

Wyżej może się np. uzgodnić label[nazwa;lambda[[a;b];[..nieważne.]]

otrzymamy w pliku lispout.h :
```
char *nazwa(arg);
```
w pliku lispout.c : gdzie arg="char *"*liczba argumentów np. arg="char *,char *"
```
char *nazwa(arg) {return USRDEFN0(arg);)
```
gdzie arg="char *a%nr"*liczba argumentów np. "char *a0,char *a1" .

Wyżej przekazujemy $$.text=nazwa;
$$.ivalue=liczba argumentów dla USRDEFN0.

Dodatkowo na specjalna listę wpisujemy nazwę oraz liczbę argumentów tej funkcji. Postępując w ten sposób przebadamy wszystkie niemal funkcje.

Teraz natomiast przedstawię jak są tłumaczone formy:

dla zmiennej $$.text = zmienna , $$.ivalue++ i wartości te przekazujemy wyżej,

dla atomu $$.text =ATOM np. $$.text="T" i $$.ivalue nieistotne.

dla wołania funkcji $$.text="nazwa_funkcji(arg)".

Wszystkie funkcje zagnieżdżone są wyrzucane na zewnątrz, w środku mamy tylko ich wywoływanie. Dla warunku : ?: użyjemy operatora warunkowego w sposób opisany przedtem. Jeżeli formy się zagnieżdżają, to łańcuchy reprezentujące ich treść są łączone. Na samym końcu, gdy już wypisaliśmy wszystkie formy, funkcje itd. pozostanie jeszcze jeden przypadek :
label[nazwa;nazwa]; pole to ma wartość $$.text=nazwa, a $$.ivalue=-1

Tak wiec na końcu gdy widzimy ze którąś funkcja ma liczbę argumentów równa minus jeden, usiłujemy ja powiązać z jakąś inna, na podstawie informacji funparent i funname typu lista. Dla każdej funkcji na liście są zrzucone wartości jej nazwy i nazwy funkcji, z która jest ona powiązana Najniżej w hierarchii funkcje odwołują się same do siebie. Tak wiec należy przejrzeć cala listę w celu odpowiedniego potraktowania każdej funkcji oraz wykrycia błędów niezdefiniowania funkcji oraz zagnieżdżeń cyklicznych (np. funkcja a odwołuje się do b, b do c, a c do a).

Na samym końcu do pliku lispout.c dopisujemy funkcję main, w której generuje się samoistnie obsługa zdefiniowanych w pliku funkcji.

V Opis testowania

W początkowych testach zlokalizowaliśmy kilka błędów, które następnie poprawiliśmy Nie były one związane z sama translacja. Pierwszy błąd wynikał z tego, iż w funkcji main pobieraliśmy argumenty dla wywoływanych funkcji przy pomocy funkcji sscanf, co powodowało ze np. (A B) było traktowane jako dwa argumenty zamiast jako jeden. Tak wiec pomimo poprawnej translacji otrzymywaliśmy błędne wyniki. Po zamianie sscanf na gets, wywoływanej dla każdego argumentu,

błąd zniknął Przed funkcja gets musieliśmy jeszcze wstawić czyszczenie bufora, gdyż zdarzało się ze po pierwszym wpisaniu jako argumentu (A B) każde następne wywołanie następowało z tym właśnie argumentem.

Kolejny błąd który udało nam się zlokalizować był związany z niepoprawnością funkcji standardowych. Założyliśmy, ze null powinien otrzymywać jako argumenty dane atomowe, w przeciwnym razie powinien się wieszać Tymczasem przecież null powinien działać również dla danych nieatomowych (i zwracać dla nich atom "F" czyli "fałsz"). Napisaliśmy wiec jeszcze raz tą funkcje.

Zauważyć należy, ze żaden błąd nie był związany z sama translacja, kłopoty mieliśmy jedynie z funkcjami bibliotecznymi. Podejrzewamy, ze inne błędy, które być może będą powodowały powstawanie nieprawidłowych wyników również będą związane z funkcjami bibliotecznymi.

Nasz translator testowaliśmy dla interpretatora LISP-u w postaci takiej jaka wypisaliśmy nieco wcześniej , otrzymaliśmy wynik :

plik lispout.h

```
#ifndef __LISPOUT.H
#define __LISPOUT.H

char* evalquote(char*,char*);
char* apply(char*,char*,char*);
char* eval(char*,char*);
char* evcon(char*,char*);
char* evlis(char*,char*);
char* car(char*);
char* cdr(char*);
char* cons(char*,char*);
char* eq(char*,char*);
char* null(char*);
char* atom(char*);

int evalint(char*);

#endif
```

---

plik lispout.c

```
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include "lispout.h"
#include "stdlisp.c"

/*********************************************************/
/*                                                       */
/*           LISP 2 C - copyright i inne takie by        */
/*          A.D.Danilecki "szopen" i K.Jurkiewicz        */
/*                                                       */
/*********************************************************/

char* USRDEFN0(char* fu,char* x)
{
 return apply(fu,x,"NIL");
}

char* evalquote(char* a0 ,char* a1)
{
 return USRDEFN0(a0 ,a1);
}
```

```
char* USRDEFN1(char* fu,char* x,char* a)
{
 return
(
 evalint(atom(fu)) ?
(
 evalint(eq(fu,"CAR")) ? car(car(x)) :
     (
      evalint(eq(fu,"CDR")) ? cdr(car(x)) :
    (
     evalint(eq(fu,"CONS")) ? cons(car(x),car(cdr(x))) :
   (
    evalint(eq(fu,"ATOM")) ? atom(car(x)) :
  (
   evalint(eq(fu,"EQ")) ? eq(car(x),car(cdr(x))) :
 (
  evalint("T") ? apply(eval(fu,a),x,a) : "NIL"
 )
  )
   )
    )
     )
) :
  (
   evalint(eq(car(fu),"LAMBDA")) ?
eval(car(car(cdr(fu))),pairlis(car(cdr(fu)),x,a)) :
 (
  evalint(eq(car(fu),"LABEL")) ?
apply(car(cdr(cdr(fu))),x,cons(cons(car(cdr(fu)),car(cdr(cdr(fu)))),a)) : "NIL"
 )
  )
);
}

char* apply(char* a0 ,char* a1 ,char* a2)
{
 return USRDEFN1(a0 ,a1 ,a2);
}

char* USRDEFN2(char* fo,char* a)
{
 return
(
 evalint(atom(fo)) ? cdr(assoc(fo,a)) :
  (
   evalint(atom(car(fo))) ?
(
 evalint(eq(car(fo),"QUOTE")) ? car(cdr(fo)) :
  (
   evalint(eq(car(fo),"COND")) ? evcon(cdr(fo),a) :
 (
  evalint("T") ? apply(car(fo),evlis(cdr(fo),a),a) : "NIL"
 )
  )
) :
 (
  evalint("T") ? apply(car(fo),evlis(cdr(fo),a),a) : "NIL"
 )
  )
);
}
```

```
char* eval(char* a0 ,char* a1)
{
 return USRDEFN2(a0 ,a1);
}

char* USRDEFN3(char* c,char* a)
{
 return
(
 evalint(eval(car(car(c)),a)) ? eval(car(cdr(car(c))),a) :
 (
  evalint("T") ? evcon(cdr(c),a) : "NIL"
 )
);
}

char* evcon(char* a0 ,char* a1)
{
 return USRDEFN3(a0 ,a1);
}

char* USRDEFN4(char* l,char* a)
{
 return
(
 evalint(null(l)) ? "NIL" :
 (
  evalint("T") ? cons(eval(car(l),a),evlis(cdr(l),a)) : "NIL"
 )
);
}

char* evlis(char* a0 ,char* a1)
{
 return USRDEFN4(a0 ,a1);
}

char* USRDEFN5(char* x,char* a,char* c)
{
 return "T";
}

char* pairlis(char* a0 ,char* a1 ,char* a2)
{
 return USRDEFN5(a0 ,a1 ,a2);
}

char* USRDEFN6(char* a,char* f)
{
 return "T";
}

char* assoc(char* a0 ,char* a1)
{
 return USRDEFN6(a0 ,a1);
}


void amin(void)
{
```

```
char *s;
int i,z;
char *wyn;
int koniec=0;
char c;
char *arg[10];

for (z=0;z<10;z++) arg[z]=(char*) malloc(256);
while(!koniec)
       {

       printf("| 0 - evalquote |\n");
       printf("| 1 - apply |\n");
       printf("| 2 - eval |\n");
       printf("| 3 - evcon |\n");
       printf("| 4 - evlis |\n");
       printf("| 5 - pairlis |\n");
       printf("| 6 - assoc |\n");
       printf("| 7 - car |\n");
       printf("| 8 - cdr |\n");
       printf("| 9 - cons |\n");
       printf("| 10 - eq |\n");
       printf("| 11 - null |\n");
       printf("| 12 - atom |\n");
       printf("Wpisz numer funkcji\n");
       scanf("%d",&i);
       switch (i)
              {

              case(0) :
              for (z=0;z<2;z++)
                     {

                     printf("Podaj parametr nr %d funkcji evalquote: ",z+1);
                     fflush(stdin);
                     gets(arg[z]);
                     }
              wyn=evalquote(arg[0],arg[1]);
              break;

              case(1) :
              for (z=0;z<3;z++)
                     {

                     printf("Podaj parametr nr %d funkcji apply: ",z+1);
                     fflush(stdin);
                     gets(arg[z]);
                     }
              wyn=apply(arg[0],arg[1],arg[2]);
              break;

              case(2) :
              for (z=0;z<2;z++)
                     {

                     printf("Podaj parametr nr %d funkcji eval: ",z+1);
                     fflush(stdin);
                     gets(arg[z]);
                     }
              wyn=eval(arg[0],arg[1]);
              break;
```

```
case(3) :
for (z=0;z<2;z++)
        {

        printf("Podaj parametr nr %d funkcji evcon: ",z+1);
        fflush(stdin);
        gets(arg[z]);
        }
wyn=evcon(arg[0],arg[1]);
break;

case(4) :
for (z=0;z<2;z++)
        {

        printf("Podaj parametr nr %d funkcji evlis: ",z+1);
        fflush(stdin);
        gets(arg[z]);
        }
wyn=evlis(arg[0],arg[1]);
break;

case(5) :
for (z=0;z<3;z++)
        {

        printf("Podaj parametr nr %d funkcji pairlis: ",z+1);
        fflush(stdin);
        gets(arg[z]);
        }
wyn=pairlis(arg[0],arg[1],arg[2]);
break;

case(6) :
for (z=0;z<2;z++)
        {

        printf("Podaj parametr nr %d funkcji assoc: ",z+1);
        fflush(stdin);
        gets(arg[z]);
        }
wyn=assoc(arg[0],arg[1]);
break;

case(7) :
for (z=0;z<1;z++)
        {

        printf("Podaj parametr nr %d funkcji car: ",z+1);
        fflush(stdin);
        gets(arg[z]);
        }
wyn=car(arg[0]);
break;

case(8) :
for (z=0;z<1;z++)
        {

        printf("Podaj parametr nr %d funkcji cdr: ",z+1);
```

```
                        fflush(stdin);
                        gets(arg[z]);
                        }
                wyn=cdr(arg[0]);
                break;

                case(9) :
                for (z=0;z<2;z++)
                        {

                        printf("Podaj parametr nr %d funkcji cons: ",z+1);
                        fflush(stdin);
                        gets(arg[z]);
                        }
                wyn=cons(arg[0],arg[1]);
                break;

                case(10) :
                for (z=0;z<2;z++)
                        {

                        printf("Podaj parametr nr %d funkcji eq: ",z+1);
                        fflush(stdin);
                        gets(arg[z]);
                        }
                wyn=eq(arg[0],arg[1]);
                break;

                case(11) :
                for (z=0;z<1;z++)
                        {

                        printf("Podaj parametr nr %d funkcji null: ",z+1);
                        fflush(stdin);
                        gets(arg[z]);
                        }
                wyn=null(arg[0]);
                break;

                case(12) :
                for (z=0;z<1;z++)
                        {

                        printf("Podaj parametr nr %d funkcji atom: ",z+1);
                        fflush(stdin);
                        gets(arg[z]);
                        }
                wyn=atom(arg[0]);
                break;

                }
        printf("Wynik funkcji jest następujący: %s\n",wyn);
        printf("Czy kontynuowac?jesli chcesz przerwac nacisnij małe q\n");
        c=getch();
        if (c=='q')
                koniec=1;
        while(kbhit())
                getch();
        }
for (z=0;z<10;z++)
        {
```

```
        free(arg[z]);
        }

}

-----------------------
Jak widać, z 30 linii w LISPie otrzymaliśmy 330 linii w C, lub inaczej mówiąc
zamiast 1206 kb tekstu w LISPie mamy ponad 5798 kb w C.
 Dla funkcji:
 label[silnia;lambda[[a;x];
     [x->silnia[a;F];
     T->ABBA]
     ]
]
lambda[[];silnia[(A.B);()]]
otrzymaliśmy natomiast:
-----------------------
lispout.h
----------------------------
#ifndef __LISPOUT.H
#define __LISPOUT.H

char* silnia(char*,char*);
char* car(char*);
char* cdr(char*);
char* cons(char*,char*);
char* eq(char*,char*);
char* null(char*);
char* atom(char*);

int evalint(char*);

#endif
------------------------
 lispout.c
--------------------------
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include "lispout.h"
#include "stdlisp.c"

/*********************************************************/
/*                                                       */
/*           LISP 2 C - copyright i inne takie by        */
/*          A.D.Danilecki "szopen" i K.Jurkiewicz        */
/*                                                       */
/*********************************************************/

char* USRDEFN0(char* a,char* x)
{
 return
(
 evalint(x) ? silnia(a,"F") :
 (
  evalint("T") ? "ABBA" : "NIL"
 )
);
}

char* silnia(char* a0 ,char* a1)
```

```
{
 return USRDEFN0(a0 ,a1);
}

char* USRDEFN1(void)
{
 return silnia("(A.B)","()");
}


void amin(void)
{
 char *s;
 int i,z;
 char *wyn;
 int koniec=0;
 char c;
 char *arg[10];

 for (z=0;z<10;z++) arg[z]=(char*) malloc(256);
 while(!koniec)
        {

        printf("| 0 - silnia |\n");
        printf("| 1 - car |\n");
        printf("| 2 - cdr |\n");
        printf("| 3 - cons |\n");
        printf("| 4 - eq |\n");
        printf("| 5 - null |\n");
        printf("| 6 - atom |\n");
        printf("Wpisz numer funkcji\n");
        scanf("%d",&i);
        switch (i)
                {

                case(0) :
                for (z=0;z<2;z++)
                        {

                        printf("Podaj parametr nr %d funkcji silnia: ",z+1);
                        fflush(stdin);
                        gets(arg[z]);
                        }
                wyn=silnia(arg[0],arg[1]);
                break;

                case(1) :
                for (z=0;z<1;z++)
                        {

                        printf("Podaj parametr nr %d funkcji car: ",z+1);
                        fflush(stdin);
                        gets(arg[z]);
                        }
                wyn=car(arg[0]);
                break;

                case(2) :
                for (z=0;z<1;z++)
                        {
```

```
                        printf("Podaj parametr nr %d funkcji cdr: ",z+1);
                        fflush(stdin);
                        gets(arg[z]);
                        }
                wyn=cdr(arg[0]);
                break;

                case(3) :
                for (z=0;z<2;z++)
                        {

                        printf("Podaj parametr nr %d funkcji cons: ",z+1);
                        fflush(stdin);
                        gets(arg[z]);
                        }
                wyn=cons(arg[0],arg[1]);
                break;

                case(4) :
                for (z=0;z<2;z++)
                        {

                        printf("Podaj parametr nr %d funkcji eq: ",z+1);
                        fflush(stdin);
                        gets(arg[z]);
                        }
                wyn=eq(arg[0],arg[1]);
                break;

                case(5) :
                for (z=0;z<1;z++)
                        {

                        printf("Podaj parametr nr %d funkcji null: ",z+1);
                        fflush(stdin);
                        gets(arg[z]);
                        }
                wyn=null(arg[0]);
                break;

                case(6) :
                for (z=0;z<1;z++)
                        {

                        printf("Podaj parametr nr %d funkcji atom: ",z+1);
                        fflush(stdin);
                        gets(arg[z]);
                        }
                wyn=atom(arg[0]);
                break;

                }
        printf("Wynik funkcji jest następujący: %s\n",wyn);
        printf("Czy kontynuowac?jesli chcesz przerwac nacisnij małe q\n");
        c=getch();
        if (c=='q')
                koniec=1;
        while(kbhit())
                getch();
        }
for (z=0;z<10;z++)
```

```
        {
        free(arg[z]);
        }

}
```

Jak widać translacja przebiega prawidłowo zarówno dla skomplikowanych jak i małych przypadków

Po drobnych poprawkach w pliku stdlisp.c oraz dodaniu funkcji main sprawdziliśmy dla niektórych funkcji poprawność obliczeń Ku naszemu zadowoleniu testy przebiegły pomyślnie

VI Rozliczenie wykonania.

W pracy nad projektem udział wzięli :
Arkadiusz Daniel Danilecki, inf40703, grupa i1:
- kierowanie projektowaniem,
- strona teoretyczna projektu (LISP),
- coding (funkcje biblioteczne stdlisp.c, część interpretera),
- dokumentacja.

Konrad Marek Jurkiewicz, inf40726, grupa i2 :
- strona techniczna (LEX&YACC),
- coding (wszystko pozostałe),
- dokumentacja.

Ustaliliśmy następujący podział przyznanych punktów:
- Arkadiusz Daniel Danilecki - 60% punktów,
- Konrad Marek Jurkiewicz - 40% punktów.

Nieoficjalne podziękowania:
- dr inż. Jolanta Cybulka - pomoc przy projektowaniu strony teoretycznej, cierpliwość,
- Agnieszka Ławrynowicz - udostępnienie literatury,
- Łukasz "Joseph" Józefowski :), Tomasz Jańczuk, Łukasz Gwóźdź, Marcin Przysucha - idea exchange,
- (jeszcze nie mgr) Grażyna Przywarska - od "Szopena" za wszystko, od Konrada za mobilizowanie "Szopena",
- Julian Konior - placki, miła atmosfera, wsparcie moralne,
- extra od "Szopena" - mgr. inż. Rafał Chmielewski - ;).

Literatura:
- wykłady Języki formalne,
- dr inż. J. Martinek "LISP".