Translacja czystego Lispu na język C

Raport „Opis projektu”

27 maj 1997

Autorzy: Łukasz Gwoździ, Tomasz Jańczuk, Łukasz Józefowski, Marcin Przysucha

Recenzenci: dr inż. Jolanta Cybulka i dr inż. Beata Jankowska

1. Analiza wymagań

1.1 Cel projektu

Celem projektu jest stworzenie programu sprawdzającego poprawność składniową funkcji czystego LISPu i tłumaczącego je na odpowiednie funkcje w języku C.

1.2 Wejście

Translator powinien akceptować na wejściu plik tekstowy zawierający definicje funkcji czystego LISPu. Plik wejściowy nie może zawierać samodzielnych wywołań funkcji, gdyż translator nie tworzy kodu służącego do uruchomienia przetłumaczonych funkcji.

1.3 Wyjście

W pierwszym kroku translator powinien zbadać poprawność składniową funkcji LISPowskich z pliku wejściowego.

Po stwierdzeniu poprawności składni translator powinien utworzyć bibliotekę funkcji w języku C będących odpowiednikami funkcji LISPowskich z pliku wejściowego. Do biblioteki dołączony powinien też być zbiór podstawowych funkcji LISPu (CAR, CDR, CONS etc.) niezależnie, czy są one wykorzystywane w przetłumaczonych funkcjach. Dodatkowo, wyjściowa biblioteka powinna zawierać dwie funkcje służące do konwersji pomiędzy ciągiem znaków a przyjętą reprezentacją list w C.

1.4 Realizacja

W szczególności realizacja projektu wymagać będzie:

- zaprojektowania w C struktury danych odzwierciedlającej struktury LISPu: atom i parę kropkową,

- określenie sposobu reakcji na błędne wywołania funkcji np. niezgodności parametrów.

2. Projekt wstępny

2.1 Wstęp

Każdą z funkcji elementarnych lispu tłumaczyć będziemy na jej odpowiednik w C. W związku z tym, konieczne będzie stworzenie biblioteki funkcji – odpowiedników funkcji czystego lispu, oraz pewnej funkcji umożliwiającej prezentację wyników. Zarówno format danych, jak i implementacja odpowiedników funkcji elementarnych lispu są niezależne od elementów przedstawionych w tym raporcie i zostaną przedstawione w raporcie „Opis realizacji".

2.2 Analizator leksykalny

Analizator został stworzony z wykorzystaniem MKS Lex'a. Specyfikacja przedstawia się następująco:

```
label                   return _label;
lambda                  return _lambda;
c[ad]+r                 { yylval.str = (char *)calloc((strlen(yytext)-1),sizeof(char));
                          strncpy(yylval.str,yytext+1,strlen(yytext)-2);
                          return _car; }
quote                   return _quote;
cons                    return _cons;
cond                    return _cond;
atom                    return _atom;
eq                      return _eq;
[a-zA-Z][a-zA-Z0-9]*    { yylval.str = (char *)malloc((strlen(yytext)+1)*sizeof(char));
                          strcpy(yylval.str,yytext);
                          return _id; }
"("                     return '(';
")"                     return ')';
" "|\t                  ;
\n                      LineNr++;
.                       { printf("%d: Lexical error\n",LineNr); exit(1); }
```

Jak widać, do przekazywane do analizatora składniowego jednostki leksykalne w większości odpowiadają funkcjom czystego lispu. Wyjątek stanowią funkcje złożone, równoważne sekwencji wywołań funkcji car i cdr. Translację na odpowiedni ciąg wywołań funkcji elementarnych zaplanowaliśmy do realizacji na etapie analizy składniowej. Będzie ona wykonana przez wykonanie stosownej akcji semantycznej.

2.3 Analizator składniowy

Nasza specyfikacja akceptora czystego lispu, zapisana w formacie MKS Yacc ma postać:

```
R:      R F     // program wejściowy to zbiór wywołań i deklaracji funkcji
|       F
;
F:      '(' _label _id '(' _lambda '(' PARAM ')' '(' CIALO ')' ')' ')'  // funkcja zapisana w postaci lispowej
;
PARAM: _id PARAM        // ciąg (w szczególności pusty) parametrów formalnych funkcji, z których każdy
|                       // posiada swój identyfikator
;
CIALO:  _car PAR1       // ciało funkcji jest wywołaniem jednej z funkcji elementarnych
|       _quote QPARAM
|       _cons PAR1 PAR1
|       _cond CONDPARAM
|       _atom PAR1
|       _eq PAR1 PAR1
|       _id CPARAM       // lub funkcji zdefiniowanej przez użytkownika
;
CPARAM: PAR1 CPARAM // ciąg parametrów funkcji zdefiniowanych przez użytkownika
|
;
CONDPARAM:    '(' PAR1 PAR1 ')' CONDPARAM       // specyficzna struktura parametrów funkcji COND,
|                                               // postaci ciągu wyrażeń ( warunek wartość)
;
PAR1:   '(' CIALO ')'   // pojedynczym parametrem może być wartość pewnej funkcji
|       _id             // lub wartość skojarzonej z identyfikatorem zmiennej
;
QPARAM:         LISTA   // parametrem funkcji QUOTE może być pewna lista
|       _id             // lub atom
;
LISTA:  '(' ')'         // lista może być listą pustą
|       '(' ILISTA ')'  // lub ciągiem elementów listy
;
ILISTA: ILISTA ELLISTY  // ciąg elementów listy może składać się z wielu
|       ELLISTY         // lub jednego elementów
;
ELLISTY:        _id     // element listy może być atomem
|               LISTA   // lub listą
;
```

Ze względu na różnorodność postaci parametrów funkcji, zdefiniowaliśmy kilka typów parametrów. Mamy nadzieję, że poprawia to czytelność specyfikacji akceptora i wprowadzanie ewentualnych zmian.

3. Opis realizacji projektu

3.1 Translator

Translator powstał przez dodanie akcji semantycznych do opisanego w poprzednim raporcie akceptora czystego lispu. Przyjęliśmy zasadę, aby wypisanie przetłumaczonej na C funkcji opóźnić do momentu całkowitego sprawdzenia jej poprawności. W związku z tym, wszystkie akcje związane z symbolami nieterminalnymi jako wartość przekazują łańcuch, będący fragmentem kodu w C. Tenże łańcuch zostaje wypisany do pliku wyjściowego w wyniku akcji semantycznej związanej z zaakceptowaniem zapisu danej funkcji. Specyfikację translatora w MKS Yacc-u dołączono do raportu.

3.2 Opis kodu w C

3.2.1 Uwagi ogólne

Aby zapewnić prawidłowe działanie przetłumaczonych programów postanowiliśmy ujednolicić typ danych, na jakich działają odpowiedniki funkcji czystego lispu. Deklaracje funkcji – odpowiedników funkcji elementarnych lispu, stałe oraz deklaracja typu danych, na jakich operują wszystkie funkcje zawarte są w pliku nagłówkowym lisp_std.h, automatycznie włączanym do każdego pliku wynikowego dyrektywą #include. Łączenie z programem użytkownika odbywa się według standardowych reguł dla programów w C. Nie dostarczamy programu wywołującego przetłumaczone funkcje, wychodząc z założenia, że użytkownik sam zdecyduje, jak i gdzie zastosować wygenerowany kod.

3.2.2 Typ danych

Zarówno parametry, jak i wartości wszystkich przetłumaczonych i wbudowanych funkcji są typu *Element. Poniżej podajemy deklarację tego typu:

```
enum typ {ATOM , PARA} // dla określenia typu danego elementu: atom czy para kropkowa
typedef struct dane {
  enum typ rodzaj; // pole określające rodzaj danego elementu : atom czy para kropkowa
  union {
    char *atom;        // wartość atomu
    struct {
      struct dane *lewy, *prawy; // wskaźniki na lewy i prawy element pary kropkowej
    } para;
  } d;
} Element;
```

3.2.3 Definicje stałych

Zdefiniowaliśmy następujące stałe:

#define NIL "()"
#define FALSE "()"
#define TRUE "#T"

Pierwsza definicja wyraża równoważność istniejącą w czystym lispie, pozostałe mają związek z zasadami ewaluacji wyrażeń logicznych w lispie (patrz opis funkcji true i false).

3.2.4 Opis funkcji biblioteki „lisp_std”

3.2.4.1 funkcje – odpowiedniki funkcji czystego lispu

- Element *car(const Element *lst);
  funkcja zwraca lewy element pary kropkowej; w przypadku wywołania dla zmiennej atomowej wartość funkcji jest przypadkowa, podobnie jak w czystym lispie
- Element *cdr(const Element *lst);
  funkcja zwraca prawy element pary kropkowej; w przypadku wywołania dla zmiennej atomowej — partrz opis funkcji car
- Element *cons(Element *l1, Element *l2);
  funkcja tworzy (lub zwraca wskaźnik na istniejącą - patrz podrozdział „zarządzanie pamięcią”) parę kropkową postaci (l1 . l2)
- Element *atom(const Element *lst);
  funkcja ma wartość logiczną (patrz opis funkcji true i false) true, gdy lst->rodzaj ma wartość ATOM
- Element *eq(const Element *l1, const Element *l2);
  funkcja ma wartość „prawda”, gdy porównywane atomy są identyczne; jeśli choć jeden z parametrów jest listą, wartość funkcji jest przypadkowa

3.2.4.2 funkcje obliczające wartość logiczną wyrażeń

Przyjęliśmy (na podstawie zasad reprezentacji prawdy i fałszu w lispie) zasadę, że fałsz jest reprezentowany jako lista pusta (patrz definicja stałej FALSE), (zatem atom NIL ma wartość fałsz), a prawda jako lista niepusta lub dowolny atom różny od nil. Zwyczajowe reprezentowanie prawdy jako QUOTE #T znalazło odbicie w definicji stałej TRUE. Zgodnie z tymi zasadami działają funkcje:

int true(const Element *lst);
int false(const Element *lst);

3.2.4.3 funkcje pomocnicze

- Element *newatom(const char *atom);
  Tworzy nowy atom i przypisuje mu wartość określoną przez parametr atom (lub zwraca wskaźnik na istniejący atom o identycznej wartości)
- char *listtoa(const Element *lst, char *buf);
  Tworzy reprezentację elementu lub listy w postaci łańcucha znakowego
- void lispFlush(void);

Zwalnia pamięć zajmowaną przez ewentualne zmienne wewnętrzne utworzone w wyniku uruchomienia jednej lub wielu funkcji z biblioteki lisp_std

3.2.5 Zarządzanie pamięcią

Dla ograniczenia zajmowanej przez zmienne pamięci wykorzystujemy tablicę zmiennych utworzonych przez funkcje *newatom* lub *cons*. Podczas każdego wywołania jednej z tych funkcji następuje sprawdzenie, czy w tablicy zmiennych nie istnieje zmienna o identycznej wartości. Taki mechanizm gwarantuje, że pamięć jest alokowana tylko raz dla zmiennej o określonej wartości. Jedyny sposób na zwolnienie pamięci zajmowanej przez zmienne tworzone w ten sposób to wywołanie funkcji lispFlush. Przepełnienie tablicy zmiennych wewnętrznych powoduje wygenerowanie błędu wykonania.

4. Opis testowania

Testowanie naszego translatora składało się z dwóch etapów: pierwszy obejmował testowanie samego translatora, drugi zaś miał za zadanie zweryfikować poprawność i efektywność biblioteki funkcji standardowych lispu.

Etap testowania samego translatora okazał się być dość szybki. Nie wykryto w zasadzie żadnych błędów w działaniu zarówno akceptora, jak i finalnej wersji translatora.

Testowanie działania samych funkcji standardowych pokazało, że funkcjonują bez zarzutu. Niestety, podczas uruchamiania programu okazało się, że wymagania pamięciowe pierwszej wersji są zbyt duże i nie było możliwe wykorzystanie nieco bardziej skomplikowanych funkcji lub bardziej rozbudowanych danych. W związku z tym, zdecydowaliśmy się na wprowadzenie poprawki, obejmującej modyfikacje funkcji cons i newatom tak, aby wyeliminować zbędne alokacje pamięci na, na przykład identyczne atomy. Po tej poprawce przetestowaliśmy cały system na nieco bardziej skomplikowanych funkcjach i ich złożeniach. Przykłady tego typu funkcji (oraz ich translacji) zamieściliśmy na dołączonej dyskietce.

5. Dokumentacja użytkowa

Jak to opisano w poprzednich raportach, na wejściu naszego translatora ma pojawić się plik zawierający zbiór funkcji czystego lispu przeznaczonych do translacji. Uruchomienie translatora ma postać:

l2c.exe < plik_wejścowy > plik_wyjściowy

Na początku pliku wyjściowego, zawierającego przetłumaczone na C funkcje z pliku wejściowego, automatycznie dołączona zostaje dyrektywa kompilatora c

#include „lisp_std.h”

W pliku lisp_std.h znajdują się prototypy funkcji podstawowych czystego lispu. Z kolei plik lisp_std.c, zawierający definicje tych funkcji, musi zostać włączony do realizowanego przez użytkownika projektu. Innym sposobem jest włączenie skompilowanego już pliku lisp_std.c, na przykład lisp_std.obj na etapie łączenia wykonywalnego programu użytkownika.

Same przetłumaczone funkcje można wykorzystać na wiele sposobów. Najprostszym, choć może mało eleganckim sposobem, jest włączenie dyrektywą #include także pliku zawierającego przetłumaczone już funkcje. Można także umieścić ich prototypy w osobnym pliku nagłówkowym i włączyć ich definicje w sposób opisany powyżej dla biblioteki standardowych funkcji lispu.

W programie należy przygotować dane do użycia przez funkcje lispu przy pomocy funkcji newatom oraz cons (opis funkcji w raporcie „Opis realizacji projektu”). Po zakończeniu działania na danej porcji danych, zaleca się użycie funkcji LispFlush, dla zwolnienia pamięci, zajmowanej przez zmienne powstałe w wyniku działania programu.

Dla przykładowego pliku extract.lsp, zawierającego funkcję extract, linearyzującą funkcję listę zagnieżdżoną, oraz pomocniczą funkcję append(łączącą dwie listy), postaci:

```
(label append (lambda (x y)
    (cond
        ((eq x (quote ())) y)
        ((quote T) (cons (car x) (append (cdr x) y)))
    )
))
(label extract (lambda (x)
    (cond
        ((eq x (quote ())) x)
        ((atom x) (cons x (quote ())))
        ((quote T) (append (extract (car x)) (extract (cdr x))))
    )
))
```

plik wynikowy ma postać:

```
#include "lisp_std.h"
```

```
Element *append(Element *x, Element *y) {
    return (true(eq(x, newatom(NIL))) ? y
    : (true(newatom("T")) ? cons(car(x), append(cdr(x), y))
    : newatom(NIL)));
}

Element *extract(Element *x) {
    return (true(eq(x, newatom(NIL))) ? x
    : (true(atom(x)) ? cons(x, newatom(NIL))
    : (true(newatom("T")) ? append(extract(car(x)), extract(cdr(x)))
    : newatom(NIL))));
}
```

Przykładowy program używający tej funkcji miałby postać:

```
#include <stdio.h>
#include "lisp_std.h"
#include „extract.c”

extern int elnum; // zmienna pozwala nam określić, ile jest zmiennych na „stosie wewnętrznym lispu”

void main() {
    Element *x, *y, *a;
    char xs[80],ys[80],as[80];
/* przygotowanie danych */
    x = cons(newatom("a"),newatom(NIL));
    y = cons(newatom("c"),cons(newatom("d"),newatom(NIL)));

/* wypisanie liczby zmiennych na „stosie lispu” */
    printf("Przed append i extract: %d\n",elnum);
    a = extract(cons(cons(y,x),newatom(NIL)));
    printf("Po extract: %d\n",elnum);
    printf("x: %s\ny: %s\nxr: %s\n",listtoa(x,xs),listtoa(y,ys),
        listtoa(a,as));
/* zwolnienie pamięci zajmowanej przez zmienne lispu */
    lispFlush();
    printf("Po lispFlush: %d\n",elnum);
}
```

6. Rozliczenie wykonania

Na propozycję rozdziału ewentualnych punktów otrzymanych za projekt wpłynął fakt, że trzech członków zespołu ma już zapewnioną ocenę „bardzo dobry” z przedmiotu „Języki formalne i kompilatory”:

- Tomasz Jańczuk — ze względu na ocenę ze sprawdzianu (5.0),
- Marcin Przysucha — ze względu na zajęcie I miejsca w I Wiosennym Turnieju Programistycznym (ocena ze sprawdzianu : 4.5),
- Łukasz Gwóźdź — z obu powyższych powodów.

W związku z powyższym zespół proponuje **całość punktów otrzymanych za projekt przepisać na konto Łukasza Józefowskiego.**