Учебный тренажер Mlispgen.

Руководство по применению.

Потребности компилятора языка Микролисп полностью покрываются кортежем из пяти атрибутов.

```
struct tSA{
  std::string line;
  string name;
  int count;
  int types;
  string obj;
}
```

В дальнейшем весь кортеж мы будем называть одним словом «атрибут», имея я виду составную природу этого объекта. Таким унифицированным атрибутом мы будем снабжать каждый узел дерева разбора.
Значение атрибута будем записывать так
[line| name | count | types | obj]
Для листьев дерева, помеченных токенами, значение атрибута имеет вид [line| lexeme | 0 | 0 |]
Таким образом, через поле name этого атрибута компилятор получает исходные данные об именах и литералах.
В поле line записан номер строки, содержащей лексему.

Продукции атрибутов это функции вида
int pi() , i уникальный номер продукции грамматики.
Функция возвращает значение 0, если она не обнаруживает ошибок, в противном случае 1.

Продукции обмениваются значениями атрибутов через стек. Управляет стеком атрибутов синтаксический анализатор.
Когда на вершине стека символов формируется основа сентенциальной формы
#...x a1 a2 ... an ,

на вершине стека атрибутов находятся атрибуты соответствующих узлов дерева разбора.

…Sx S1 S2 … Sn

Действует очень важное соглашение о передаче параметров. Продукция вызывается перед тем, как основа сворачивается к нетеминалу A. Она анализирует значения атрибутов S1,S2, … Sn и синтезирует новое значение атрибута для родительского узла, помеченного A. Это значение записывается по адресу S1.

pi(S1,S2, … Sn) -> S1

После того как продукция отработала, анализатор заменяет основу нетерминалом A и удаляет из стека атрибутов n-1 элемент.

 # …x A

…Sx S1

Это соглашение имеет очень важный побочный эффект, который можно назвать «транзитом» атрибутов. Даже еслсли функция pi имеет вид

int pi(){return 0;} ,

то есть не выполняет никаких операций с атрибутами, атрибут первого символа основы без изменений передается родительскому узлу. Атрибут остается на вершине стека и просто меняет «владельца».

Из методических соображений я разделил реализацию языка МИКРОЛИСП на две независимые части – генератор кода и семантический анализатор. По частям разрабатывать и тестировать компоненты компилятора проще и эффективнее.

В дальнейшем я буду называть генератор кода транслятором, поскольку именно этот компонент переводит текст со входного языка на целевой.

В тренажере реализована объектно-ориентированная модель транслятора с МИКРОЛИСПа на C++. В основе модели лежит базовый класс tBC, в который инкапсулирован функционал лексического и синтаксического анализа. Специфика транслятора отражена в производном классе tCG. Определение класса содержится в файле code-gen.h, а реализация в code-gen.cpp.

В классе определена переменная и функция, которые доступны всем продукциям атрибутов:

```
  std::string declarations;
  std::string decor(const std::string& id);
```

Переменная declarations собирает объявления всех функций и глобальных переменных. По завершении трансляции они вставляются в начало целевой программы.
Функция decor преобразует имена МИКРОЛИСПа в имена C++, применяя п.3 Правил перевода (см. TranslatioRules22.rtf). Реализация функции записана в конце файла code-gen.h .

Командный интерфейс тренажера Mlispgen полностью повторяет интерфейс тренажера Pars.

Рассмотрим работу тренажера на примере языка грамматики M21.

```
# $m21
  $id   $int     (      )
  define   set!
#
    S -> PROG #1
  PROG -> CALCS #2 |
          DEFS #3 |
          DEFS CALCS #4
 CALCS -> CALC #5 |
          CALCS CALC #6
  CALC -> E #7
    E -> $id #8 |
          $int #9 |
          CPROC #10
 CPROC -> HCPROC ) #11
HCPROC -> ( $id #12 |
          HCPROC E #13
   SET -> HSET E ) #14
  HSET -> ( set! $id #15
   DEF -> PROC #16
  DEFS -> DEF #17 |
          DEFS DEF #18
  PROC -> HPROC E ) #19
 HPROC -> PCPAR ) #20 |
```

HPROC SET #21
PCPAR -> (define ($id #22 |
PCPAR $id #23

Эта небольшая грамматика вполне адекватно описывает такие грамматические формы, как «Определение числовой процедуры», «Вызов числовой процедуры» и «Присваивание».

Тестовую цепочку возьмем из файла t0 и применим тренажер в режиме трассировки действий.

```
Input gramma name>m21
Gramma:m21.txt
Source>'t0
Source:t0.ss
  1|(define(
  2|       f x)x)
  3|
```

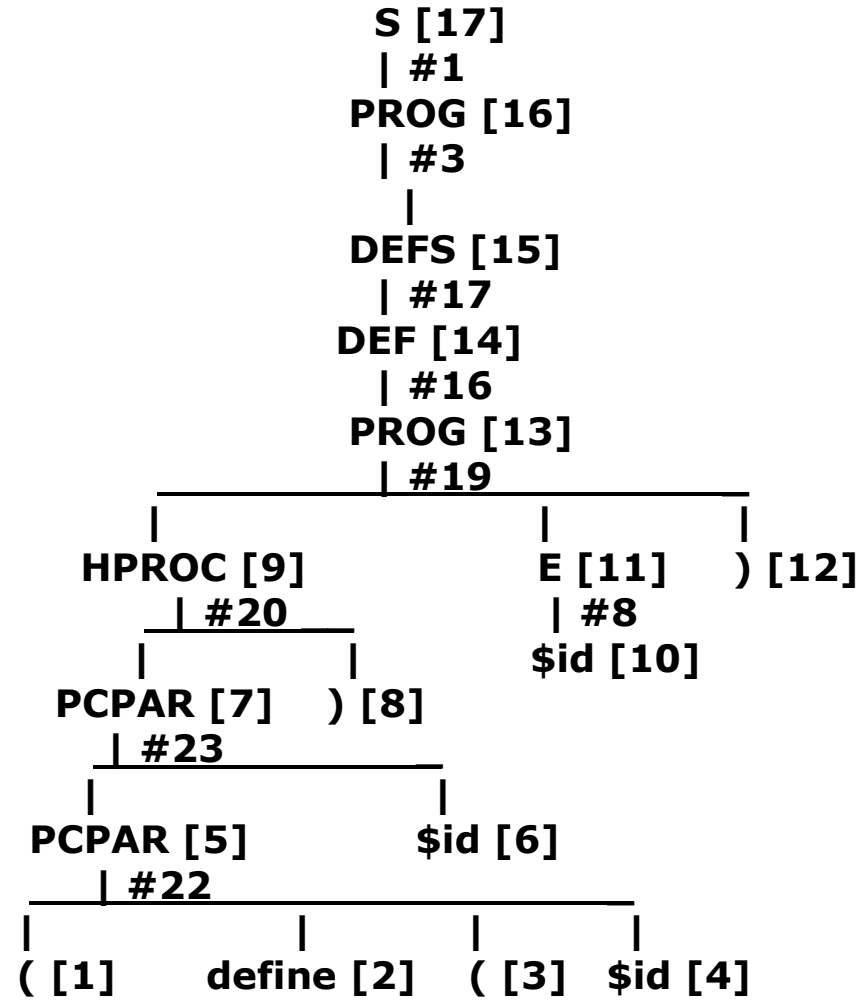
```
 <-  (
 1   [ 1| (|0| 0|  ]
 <-  define
 2   [ 1| define|0| 0|  ]
 <-  (
 3   [ 1| (|0| 0|  ]
 <-  $id
 4   [ 2| f|0| 0|  ]
  PCPAR -> ( define ( $id #22
 5   [ 1| f|0| 0| double f/*2*/ ( ]
 <-  $id
 6   [ 2| x|0| 0|  ]
  PCPAR -> PCPAR $id #23
 7   [ 1| f|1| 0| double f/*2*/ (double x ]
 <-  )
 8   [ 2| )|0| 0|  ]
  HPROC -> PCPAR ) #20
 9   [ 1| f|1| 0| double f/*2*/ (double x){
 ]
 <-  $id
10   [ 2| x|0| 0|  ]
     E -> $id #8
11   [ 2| x|0| 0| x ]
 <-  )
```

```
12  [ 2| )|0| 0|  ]
  PROC -> HPROC E ) #19
13  [ 1| f|1| 0| double f/*2*/ (double x){
return
x;
     }
]
   DEF -> PROC #16
14  [ 1| f|1| 0| double f/*2*/ (double x){
return
x;
     }
]
  DEFS -> DEF #17
15  [ 1| f|1| 0| double f/*2*/ (double x){
return
x;
     }
]
  PROG -> DEFS #3
16  [ 1| f|1| 0| double f/*2*/ (double x){
return
x;
     }
int main(){
 display("No calculations!");
       newline();
        std::cin.get();
       return 0;
       }
]
     S -> PROG #1
Code:
/*  KPG   */
#include "mlisp.h"
double f/*2*/ (double x);
//____________________
double f/*2*/ (double x){
 return
 x;
      }
int main(){
```

```
 display("No calculations!");
       newline();
        std::cin.get();
       return 0;
       }  <-  (
Code is saved to file t0.cpp !
```

Трасса трансляции помимо перечня действий синтаксического анализатора содержит атрибуты, синтезированные после каждого действия.
Все атрибуты пронумерованы. Ссылку на атрибут обозначим [i].

Ниже построено дерево разбора тестовой цепочки, аннотированное атрибутами узлов.



Проследим последовательность синтеза атрибутов по шагам. Номер шага соответствует номеру атрибута.

Шаги 1-4. Перенос токенов в стек символов.
Одновременно в стек атрибутов переносятся [1][2][3][4]. В поле name записана лексема токена. В поле line – номер строки, содержащей лексему.
Шаг 5. Свертка с применением продукции #22.

```
int tCG::p22(){ // PCPAR -> ( define ( $id
S1->obj =  "double " + decor(S4->name) +
 "/*" + S4->line + "*/ ("; S1->count = 0;
S1->name = S4->name;
  return 0;}
```

Доступ к верхним элементам стека атрибутов продукция получает через указатели S1, S2, S3, S4. Количество активных указателей равно длине правой части продукции. Указатели переустанавливаются при каждой свертке. В данном случае расположение указателей выглядит так

```
стек символов:  ...  (    define     (    $id
стек атрибутов: ... [1]    [2]      [3]  [4]
                      ^      ^       ^    ^
                     S1     S2      S3   S4
```

В поле S1->obj продукция синтезирует первый фрагмент целевой программы – заголовок объявителя функции. Оператор + выполняет сцепление строк. После имени функции вставляется комментарий с номером строки исходного текста, в которой записано имя соответствующей процедуры. Обратите внимание на то, что S4->line имеет тип std::string. Кроме того, продукция копирует имя процедуры из поля S4->name в поле S1->name, тем самым обеспечивая транзит имени к вышестоящим узлам дерева разбора.
Шаг 6. Перенос токена $id.
Шаг 7. Свертка с применением продукции #23.

```
int tCG::p23(){ // PCPAR -> PCPAR $id
if(S1->count)S1->obj += S1->count%2 ? ", " : "\n\t , ";
 S1->obj += "double " + decor(S2->name);
 ++(S1->count);  return 0;}
```

```
стек символов:  ...  PCPAR    $id
стек атрибутов: ...   [5]      [6]
                       ^        ^
                      S1       S2
```

Продукция добавляет к объявителю функции объявитель параметра x.
Кроме того, продукция в поле S1->count пересчитывает параметры процедуры.

Шаг 8. Перенос токена).

Шаг 9. Свертка с применением продукции #20.

```
int tCG::p20(){ // HPROC -> PCPAR )
 S1->obj += ")";
 declarations += S1->obj + ";\n";
 S1->obj += "{\n ";
   return 0;}
    стек символов:   ...  PCPAR       )
    стек атрибутов:  ...    [7]        [8]
                             ^          ^
                            S1         S2
```

Продукция завершает синтез объявителя функции и добавляет его к переменной declarations.
В поле S1->obj продолжается синтез определения функции.

Шаг 10. Перенос токена $id.

Шаг 11. Свертка с применением продукции #8.

```
int tCG::p08(){ //     E -> $id
 S1->obj = decor(S1->name);
   return 0;}
    стек символов:   ...    $id
    стек атрибутов:  ...   [10]
                            ^
                           S1
```

В поле S1->obj продукция синтезирует имя переменной в C++ из имени в МИКРОЛИСПе.

Шаг 12. Переос токена).

Шаг 13. Свертка с применением продукции #19.

```
int tCG::p19(){ //  PROC -> HPROC E )
S1->obj += "return\n " + S2->obj+";\n\t }\n";
   return 0;}
    стек символов:   ...  HPROC     E         )
    стек атрибутов:  ...    [9]       [11]    [12]
                             ^         ^       ^
                            S1        S2      S3
```

Продукция завершает синтез определения функции.

Шаг 14. Свертка с применением продукции #16.

```
int tCG::p16(){ //   DEF -> PROC
```

return 0;}

Транзит [13] в [14].

Шаг 15. Свертка с применением продукции #17.

int tCG::p17(){ // DEFS -> DEF

return 0;}

Транзит [14] в [15].

Шаг 16. Свертка с применением продукции #3.

int tCG::p03(){ // PROG -> DEFS

S1->obj += "int main(){\n "

"display(\"No calculations!\");\n\t newline();\n\t "

" std::cin.get();\n\t return 0;\n\t }\n"; return 0;}

стек символов: ... DEFS

стек атрибутов: ... [15]

^

S1

Продукция добавляет к перечню определений функцию main.

Шаг 17. Свертка с применением продукции #1.

int tCG::p01(){ // S -> PROG

string header ="/* " + lex.Authentication()+" */\n";

header += "#include \"mlisp.h\"\n";

header += declarations;

header += "//_____ \n";

S1->obj = header + S1->obj;

return 0;}

стек символов: ... PROG

стек атрибутов: ... [16]

^

S1

Продукция завершает синтез целевой программы.

Тренажер отобразит программу на экране и запишет ее в файл t0.cpp .