Lecture 8

Synthesis stage on the example of a Simple Refal compiler

Introduction to the dialect of the Simple Refal.

A simple Refal is a Refal dialect, oriented to compilation into the source code in C ++. It was designed to study the features of Refal's compilation into imperative languages. Features:

- Support for only a subset of the Base Refal (sentences have the form sample = result), the lack of more advanced features (conditions, kickbacks, actions).
- Support for nested functions.
- A simple scheme of code generation, the absence of any powerful optimizations.
- Is a self-applicable compiler.
- The basis is a classical list implementation.

Data Types of the Simple Refal

The main (and only) type of data Refal - object expression - a sequence of object terms.

Varieties of object terms:

- Atoms:
  - ASCII characters. Examples: 'a', 'c', 'ы'.
  - Integers in the range 0 ... (2^32 - 1). Examples: 42, 121.
  - Closures (a combination of a pointer to a function and the values of some local variables) are created from global functions or nameless nested functions. Examples: Fact, Go, {t.B = (t.A t.B); }.
  - Identifiers. Examples: #True, #Success.
- Composite terms:
  - Structural brackets.
  - Named brackets are not considered here.

Syntax of Simple Refal

Since one of the tasks in designing the language was writing the most simple C ++ code generator, the syntax of the language inherits some features of the target language, in particular - the need for pre-announcements.

Example. The program that replaces 'a' with 'b':

```
// Library function declarations
$EXTERN ReadLine, WriteLine;
// Declaring a local function
$FORWARD Fab;
// Program entry point
$ENTRY Go {
  =
   <WriteLine
     <Fab <ReadLine>>
   >;
}

Fab {
  e.Begin 'a' e.End = e.Begin 'b' <Fab e.End>;

  e.Other = e.Other;
}
```

Example. Implementation of a program that replaces 'a' with 'b'.

```
// At the beginning of the program implementation, the field of view is initialized
// with calling the function <Go>
<Go>
<WriteLine <Fab <ReadLine>>>
// Here there is a suspension of the performance of the Refal Machine,
// waiting for user input
// user enters 'abracadabra!!!'.
<WriteLine <Fab 'abracadabra!!!'>>
<WriteLine 'b' <Fab 'bracadabra!!!'>>
<WriteLine 'bbrb' <Fab 'cadabra!!!'>>
<WriteLine 'bbrbcb' <Fab 'dabra!!!'>>
<WriteLine 'bbrbcbdb' <Fab 'bra!!!'>>
<WriteLine 'bbrbcbdbbrb' <Fab '!!!'>>
<WriteLine 'bbrbcbdbbrb!!!'>
// Here comes the output to the screen 'abrbcbdbbrb!!!',
// The field of view becomes empty, the refal machine stops.
```

Example. Program with nested functions.

```
// The Map function converts each expression term according to a given rule
// Calling <Map s.Trans e.Elems> == e.Transformed
$ENTRY Map {
  s.Trans t.First e.Tail = <s.Trans t.First> <Map s.Trans e.Tail>;

  s.Trans = /* empty */;
}

// The CardProd function calculates the Cartesian product of two sets
// Calling <CartProd ('a' 'b' 'c') (1 2)>
//   == ('a' 1) ('a' 2)  ('b' 1) ('b' 2)  ('c' 1) ('c' 2)
$ENTRY CartProd {
  (e.SetA) (e.SetB) =
    <Map
      {
        t.A =
          <Map
            { t.B = (t.A t.B); }
            e.SetB
          >;
      }
      e.SetA
    >;
}
```

Abstract refal machine

Definition. A refal machine is an abstract device that executes programs on Refal.
Definition. A certain expression is an expression containing parentheses of the instantiation, but not containing variables.
Definition. A certain expression processed by a refal machine is called a field of view. The refal machine works in a step-by-step mode. In one step, the refal machine finds in the field of view the primary active subexpression (the leftmost pair of parentheses that do not contain other parenthesis brackets), causes a closure following the opening parenthesis with the expression between this closure and the closing parenthesis as an argument. Then the leading pair of parentheses is replaced by a certain expression resulting from the execution of the closure, and the refal machine goes on to the next step. The execution of the refal machine continues until the field of view contains the parentheses of the instantiation.

§ 40. Data structures of the Simple Refal

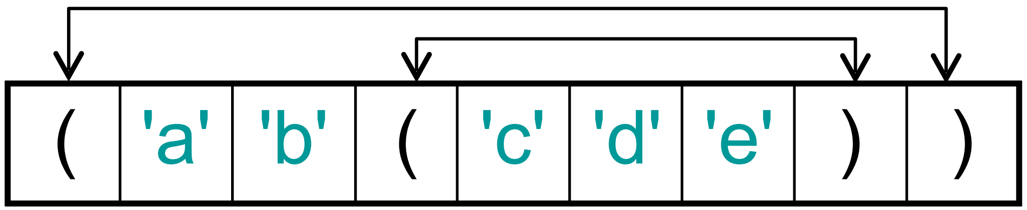
- The field of view is presented in the form of a doubly linked list.
- List nodes contain a tag of type (tag) and an information field (info). A node (depending on the type) can be an atom, one of the structural brackets, or one of the parenthesis brackets.
- Nodes-atoms (numbers, identifiers, symbols, closures without context) in the field info contain the very value of the atom.
- A node representing the structured bracket in the info field contains a reference to the matching parenthesis. This provides an effective (for a constant time) recognition of the brackets in the sample.
- The opening angle brackets contain references to the corresponding closing brackets.
- The closing angle brackets indicate the opening angle brackets, which will become leading after the current pair of parentheses are executed. Thus, the angle brackets form a function call stack.
- To speed up the operations of creating new nodes, as well as to prevent memory leaks, a list of free nodes is used.

Node structure
For clarity, some node types in the DataTag and some fields in the union are skipped.

```
typedef struct Node *NodePtr;
typedef struct Node *Iter;

typedef enum DataTag {
  cDataIllegal = 0,
  cDataChar,
  cDataNumber,
  cDataFunction,
  cDataIdentifier,
  cDataOpenBracket,
  cDataCloseBracket,
  cDataOpenCall,
  cDataCloseCall,
  cDataClosure,
  cDataClosureHead
} DataTag;

typedef
  FnResult (*RefalFunctionPtr) (
    Iter begin, Iter end
  );
```

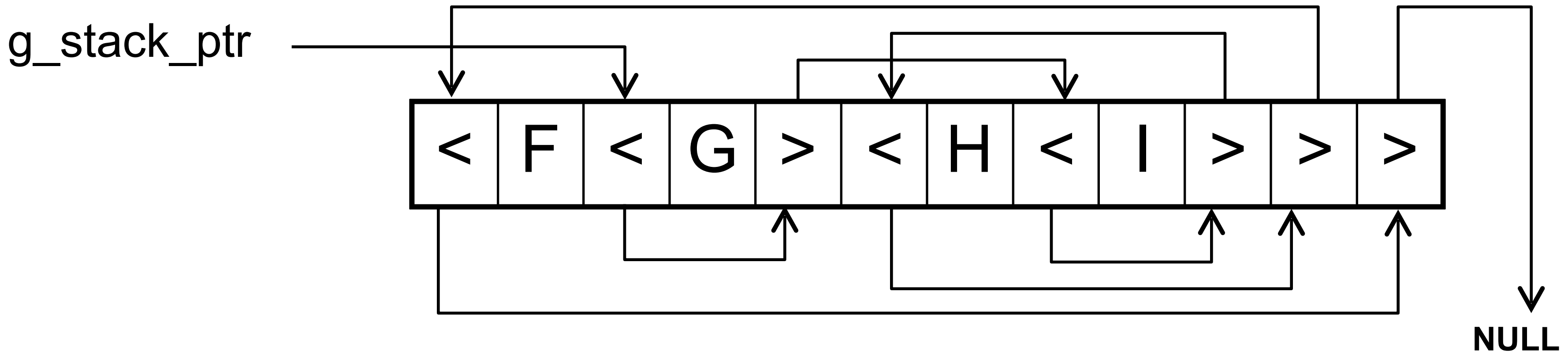
```
typedef struct RefalFunction {
  RefalFunctionPtr ptr;
  const char *name;
} RefalFunction;

typedef unsigned long RefalNumber;

typedef const char
  *(*RefalIdentifier) ();

typedef struct Node {
  NodePtr prev;
  NodePtr next;
  DataTag tag;
  union {
    char char_info;
    RefalNumber number_info;
    RefalFunction function_info;
    RefalIdentifier ident_info;
    NodePtr link_info;
  };
} Node;
```

Representation of structural brackets



Representation of angle brackets

Angular brackets form a simply-connected list, whose head is indicated by the global variable g_stack_ptr.



Representation of context closures

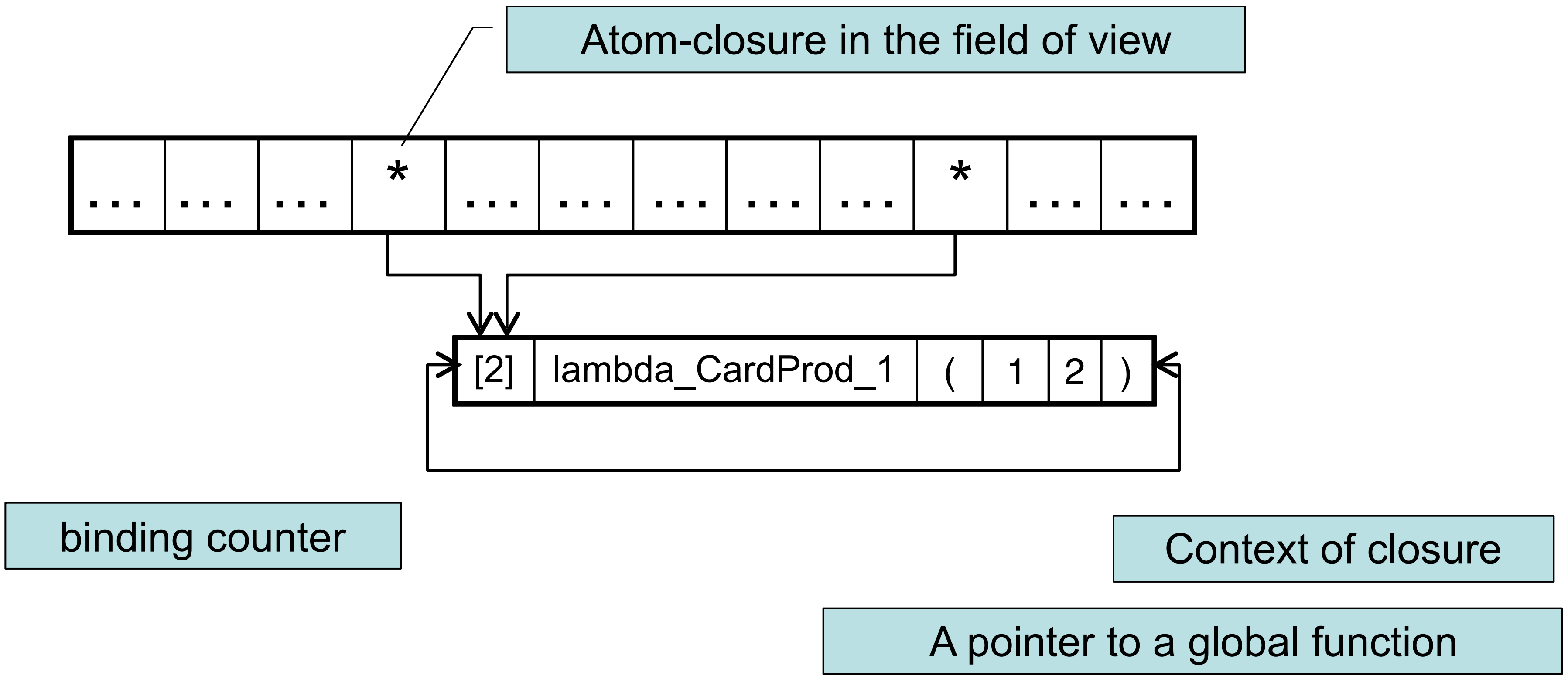
Definition. The context of the closure of an embedded function is the set of variables that are bound outside and used inside a function block.

Example. The context of the external nested function (# 1) is the variable e.SetB.
The context of the internal nested function (# 2) is the variable t.A.

```
 $ENTRY CartProd {
  (e.SetA) (e.SetB) =
    <Map
      { // #1
        t.A =
          <Map
            { t.B = (t.A t.B); } // #2
            e.SetB
          >;
      }
      e.SetA
    >;
}
```

Nested functions are implicitly converted to global functions and context binding operations.

```
lambda_CartProd_0 {
  t.A t.B = (t.A t.B);
}

lambda_CartProd_1 {
  (e.SetB) t.A =
    <Map
      <refalrts::create_closure lambda_CartProd_0 t.A>
      e.SetB
    >;
}

$ENTRY CartProd {
  (e.SetA) (e.SetB) =
    <Map
      <refalrts::create_closure lambda_CardProd_1 (e.SetB)>
      e.SetA
    >;
}
```


Atom-closure in the field of view
…
*
[2]
lambda_CardProd_1
(
1
2
)
binding counter
Context of closure
A pointer to a global function

Closures with the context are implemented as a ring list containing the connection counter, the name of the corresponding global function, and the context. Atoms-closures from the field of view (or from the contexts of other closures) indicate a connection counter. Because when copying the closure atom, the context itself is not copied, then a link counter is used to track the number of pointers to the closure (and at the right time to delete it).

§ 41. The general scheme of code generation in the Simple RefalCompilation is carried out by independent fragments, which are announcements and separate sentences of functions.

Code on Refal

```
// Library function declaration
$EXTERN WriteLine, Dec, Mul;
// Local function declaration
$FORWARD Fact;
// Program entry point
$ENTRY Go {
  = <WriteLine '6! = ' <Fact 6>>;
}

Fact {
  0 = 1;
  s.Number =
    <Mul
      s.Number
      <Fact <Dec s.Number>>
    >;
}
```

Code on C++

```
// Automatically generated file. Don't edit!
#include "refalrts.h"

extern refalrts::FnResult WriteLine(refalrts::Iter arg_begin, refalrts::Iter arg_end);

extern refalrts::FnResult Dec(refalrts::Iter arg_begin, refalrts::Iter arg_end);

extern refalrts::FnResult Mul(refalrts::Iter arg_begin, refalrts::Iter arg_end);

static refalrts::FnResult Fact(refalrts::Iter arg_begin, refalrts::Iter arg_end);

refalrts::FnResult Go(refalrts::Iter arg_begin, refalrts::Iter arg_end) {
  Sentence code
  return refalrts::cRecognitionImpossible;
}

static refalrts::FnResult Fact(refalrts::Iter arg_begin, refalrts::Iter arg_end) {
  First sentence code
  Second sentence code
  return refalrts::cRecognitionImpossible;
}

//End of file
```

Generation of identifiers is based on the fact that duplicate instantiation of templates in C ++ in different translation units is usually eliminated by the linker.

Code on Refal

```
$LABEL Success;
$LABEL Fails;

F {
  #Success =
    #Fails;
}
```

Code on C++

```
// Automatically generated file. Don't edit!
#include "refalrts.h"


//$LABEL Success
template <typename T>
struct Success {
  static const char *name() {
    return "Success";
  }
};

//$LABEL Fails
template <typename T>
struct Fails {
  static const char *name() {
    return "Fails";
  }
};


static refalrts::FnResult F(refalrts::Iter arg_begin, refalrts::Iter arg_end) {
  ...
  ... & Success<int>::name ...
  ...
  ... & Fails<int>::name ...
  ...
}

//End of the file
```

The following function structure is used:

```
refalrts::FnResult
FunctionName(refalrts::Iter arg_begin, refalrts::Iter arg_end) {
  ...
  do {
    Sentence code N
  } while(0);
  ...
  return refalrts::cRecognitionImpossible;
}
```

The logic of execution is:

1. In case of successful execution, the exit from the sentence is performed by the instruction ***return*** *refalrts :: cSuccess.*
2. With a lack of memory, the function is terminated by the ***return*** *refalrts :: cNoMemory* statement.
3. If the pattern is unsuccessfully matched, the break instruction is executed. For the last sentence, you go to the next sentence, in the case of the last one, *refalrts :: cRecognitionImpossible* is returned.

Three stages of the implementation sentence

For convenience of debugging, the function is divided into three stages:

1. *Comparison with the sample.* At this stage, the contents of the activation term (angle brackets, function name and argument itself) do not change, so that in case of failure of the match the next sentence gets an argument in the same form, and if the sentence is the last, then by dump the field of view it was possible to understand in which the function collapsed.
2. *Memory allocation for new nodes.* At this stage, the beginning of the list of free blocks is initialized with new values (copies of variables, new literals nodes: atoms, brackets). The content of the activation term is not changed here either for debugging reasons.
3. *Construction of the result.* Because the construction is carried out only by changing the pointers of the doubly linked list, this stage can not be completed unsuccessfully. Those parts of the activation term, which are not needed as a result, are transferred to the list of free blocks. This step is always terminated by the **return refalrts :: cSuccess;** statement.

Proposal pseudo-code

```
refalrts::FnResult FunctionName(refalrts::Iter arg_begin, refalrts::Iter arg_end) {
  // The first sentence
  do {
    // 1 stage  — pattern matching
    if( comparison unsuccessfully )
      break;
    // 2 stage — memory allocation
    if( the lack of memory )
      return refalrts::cNoMemory;
    // 3 stage — constructing a result
    ...
    return refalrts::cSuccess;
  } while(0);

  // The second sentence
  do {
    ...
  } while(0);

  // Return on recognition failure
  return refalrts::cRecognitionImpossible;
}
```

§ 42. Generating code for matching with a sample in a Simple Refal

The left part of the sentence is transformed into a sequence of elementary recognition commands by a fairly complex algorithm. Elementary commands are the splitting of a given atom, a bracket term, an unrecognized variable, a repeated variable from the right or left end of the object expression.

Features of the comparison with the sample:

- Object expressions are represented by a pair of pointers to the range of nodes in the field of view.
- At the beginning of the match, we only have a couple of pointers to the range of the argument.
- If the hard element is successfully cleaved-an element with the already known length: atom, bracket term, s- and t-variable, repeated e-variable - one of the range pointers is shifted by the length of the recognized splitter element.

Features of the comparison with the sample (continuation):

- Successful splitting off of the bracket term, in addition to changing the range, leads to the initialization of a new pair of range pointers for the subexpression in parentheses.
- Successful splitting of the variable (s-, t- or repeated) in addition to changing the range leads to the initialization of the pointer to this variable (for e-variables - a pair of pointers).
- Comparison with a closed e-variable is always successful. As a result of matching, a pair of pointers to the e-variable is initialized with a pair of range pointers.
- There is a command to associate a range with an empty expression.
- The comparison command with the range containing the open e-variable is translated into the elongation cycle of the e-variable.
- If it is impossible to match outside the elongation cycle of an open e-variable, the ***break*** statement is executed, and the ***continue*** statement inside the loop.

Features of recognizing open e-variables

- Before the extension cycle, a pair of pointers to an open e-variable is initialized as an empty range.
- At each iteration of the cycle, the length of the range is increased by one term.
- The cycle continues until the right end of the open e-variable falls outside the permitted range.
- In case of unsuccessful matching, the **continue** instruction is executed inside the loop of elongation, leading to the next iteration of the loop.

```
do {
 // comparison outside the cycle
 if( comparison failed )
  break;
 // Extension cycle of an open e-variable
 for( initialization; check for the admissibility of length; elongation ) {
  // comparison inside the cycle
  if( comparison failed )
   continue;
  ...
  return refalrts::cSuccess;
 }
} while(0);
```

Example. Generate a sample without open e-variables. For clarity, the prefix refalrts :: removed.

Code on Refal

```
$LABEL A;
$LABEL B;

F {
  (e.X #A) e.Y #B = result;
}
```

Pseudo-code

- B0 ← function argument
- B0 → B0 #B
- B0 → (B1) B0
- B1 → B1 #A
- B1 → e.X
- B0 → e.Y
- Building the result

Code on C++

```
...
do {
   Iter bb_0 = arg_begin;
   Iter be_0 = arg_end;
   move_left( bb_0, be_0 );
   move_left( bb_0, be_0 );
   move_right( bb_0, be_0 );
   static Iter eX_b_1, eX_e_1, eY_b_1, eY_e_1;
   // (~1 e.X # A )~1 e.Y # B
   if( ! ident_right(  & B<int>::name, bb_0, be_0 ) )
     break;
   Iter bb_1 = 0, be_1 = 0;
   if( ! brackets_left( bb_1, be_1, bb_0, be_0 ) )
     break;
   if( ! ident_right(  & A<int>::name, bb_1, be_1 ) )
     break;
   eX_b_1 = bb_1;
   eX_e_1 = be_1;
   eY_b_1 = bb_0;
   eY_e_1 = be_0;

   Building the result

} while ( 0 );
...
```

Example. Generate a sample with open e-variables. At the beginning of the cycle, the state of calculations is saved. Instead of the break statement, use the continue statement. For clarity, the prefix refalrts :: removed.

Code on Refal

```
$LABEL A;


F {
  e.X #A e.Y = result;
}
```

Pseudo-code

```
• B0 ← function argument
• Cycle (e.X B0)
•   B0 → #A B0
•   B0 → e.Y
•   Building the result
• End of cycle
```

Code on C++

```
...
do {
  Iter bb_0 = arg_begin;
  Iter be_0 = arg_end;
  move_left( bb_0, be_0 );
  move_left( bb_0, be_0 );
  move_right( bb_0, be_0 );
  static Iter eX_b_1,  eX_e_1, eY_b_1, eY_e_1;
  // e.X # A e.Y
  Iter bb_0_stk = bb_0, be_0_stk = be_0;
  for(
    Iter
      eX_b_1 = bb_0_stk, eX_oe_1 = bb_0_stk,
      bb_0 = bb_0_stk, be_0 = be_0_stk;
    ! empty_seq( eX_oe_1, be_0 );
    bb_0 = bb_0_stk, be_0 = be_0_stk, next_term( eX_oe_1, be_0 )
  ) {
    bb_0 = eX_oe_1;
    eX_b_1 = bb_0_stk;
    eX_e_1 = eX_oe_1;
    move_right( eX_b_1, eX_e_1 );
    if( ! ident_left(  & A<int>::name, bb_0, be_0 ) )
      continue;
    eY_b_1 = bb_0;
    eY_e_1 = be_0;


    Building the result

  }
} while ( 0 );
...
```

§ 43. Basic principles of code generation for constructing the result of a function in the Simple Refal

In contrast to the first stage, the right-hand side of the sentence is transformed into a sequence of elementary commands by a simpler algorithm. Elemental commands for allocating memory include commands for creating atoms, creating parentheses, and copying variables. Elementary result building commands include commands for assembling the result from fragments, associating pairs of structured parentheses, and placing angle brackets on the call stack.

Features of the stages of memory allocation and construction of the result

- All elementary commands support the invariants of doubly-connected lists of the field of view and free nodes. Those. between the calls of elementary operations, there are no "breaks" of lists, nor the emergence of "hanging" fragments.
- The fragments required for constructing the result of a function call are either in the field of view or in the list of free blocks.

Features of the stages of memory allocation and construction of the result (continuation)

- The result is assembled by the transfer operations of the form *splice (pos, begin, end)*, where *begin* and *end* are pointers to the beginning and end of the transferred fragment, *pos* is a pointer to the node before which the fragment will be placed.
- The fragments from which the result is built are placed in the field of view before the opening parenthesis of the call to the current function. There are transferred both fragments from the list of free nodes, and variables that are present in the sample. Thus, after assembling the result, only unnecessary fragments of the visual field will be between the function call brackets - they are transferred to the list of free nodes.

Example. Generate memory allocation and result assembly.

(generated code1) (generated code2)

Code on Refal

```
Fab {
  e.X #A e.Y =
    e.X #B <Fab e.Y>;

  e.X = e.X;
}
```

Pseudo-code

```
// first sentence
Cycle (e.X1, B0)
  B0 → #A B0
  B0 → e.Y1
  n0 ← allocate(#B)
  n1 ← allocate(<)
  n2 ← allocate(Fab)
  n3 ← allocate(>)
  Push(n3)
  Push(n0)
  Build(e.X1, n0, n1, n2, e.Y1, n3)
  Free(arg_begin, arg_end)
  return cSuccess
End of cycle

// second sentence
B0 → e.X1
Build(e.X1)
Free(arg_begin, arg_end)
return cSuccess
```

Example. Generate memory allocation and result assembly.

Code on Refal

```
Fact {
  0 = 1;
  s.Number =
    <Mul
      s.Number
      <Fact <Dec s.Number>>
    >;
}
```

Pseudo-code (beginning)

```
// first sentence
B0 → 0 B0
B0 → empty
n0 ← allocate(1)
Build(n0)
Free(arg_begin, arg_end)
return cSuccess
```

Pseudo-code (continuation)

```
// second sentence
B0 → s.Number$_1$ B0
B0 → empty
s.Number$_2$ ← copy(s.Number$_1$)
n0 ← allocate(<)
n1 ← allocate(Mul)
n2 ← allocate(<)
n3 ← allocate(Fact)
n4 ← allocate(<)
n5 ← allocate(Dec)
n6 ← allocate(>)
n7 ← allocate(>)
n8 ← allocate(>)
Push(n8)
Push(n0)
Push(n7)
Push(n2)
Push(n6)
Push(n4)
Build(n0, n1, s.Number$_1$, n2, n3, n4, n5, s.Number$_2$, n6)
Free(arg_begin, arg_end)
return cSuccess
```

§44. Two ways of constructing the result of a function: direct code generation and interpretation

In C ++ code, the elementary operations of the last two stages can be explicitly described as calls to corresponding functions (direct code generation mode) and a sequence of interpreted commands in the form of a constant static array, which is then passed to a special interpreter function (interpretation mode).

- The program compiled in direct code generation mode is executed faster than in the interpretation mode (it is noticeable only on old machines or on large volumes of calculations).
- The size of the program compiled in the interpretation mode is approximately one third less than the size of the program compiled in direct code generation mode.
- In the generated text, there is a code generated by both modes. The mode is selected by the conditional compilation directives of the C ++ preprocessor.

Example. Stages 2 and 3 for the example with the function Fab (first sentence). Direct code generation mode.

```
#ifdef INTERPRET
    ...
#else
    refalrts::reset_allocator();
    refalrts::Iter res = arg_begin;
    refalrts::Iter n0 = 0;
    if( ! refalrts::alloc_ident( n0, & B<int>::name ) ) return refalrts::cNoMemory;
    refalrts::Iter n1 = 0;
    if( ! refalrts::alloc_open_call( n1 ) ) return refalrts::cNoMemory;
    refalrts::Iter n2 = 0;
    if( ! refalrts::alloc_name( n2, & Fab, "Fab" ) ) return refalrts::cNoMemory;
    refalrts::Iter n3 = 0;
    if( ! refalrts::alloc_close_call( n3 ) ) return refalrts::cNoMemory;
    refalrts::push_stack( n3 );
    refalrts::push_stack( n1 );
    res = refalrts::splice_elem( res, n3 );
    res = refalrts::splice_evar( res, eY_b_1, eY_e_1 );
    res = refalrts::splice_elem( res, n2 );
    res = refalrts::splice_elem( res, n1 );
    res = refalrts::splice_elem( res, n0 );
    res = refalrts::splice_evar( res, eX_b_1, eX_e_1 );
    refalrts::splice_to_freelist( arg_begin, arg_end );
    return refalrts::cSuccess;
#endif
```

The structure of an elementary interpreted command

```
typedef enum iCmd {
  icChar,
  icInt,
  icFunc,
  icIdent,
  icString,
  icBracket,
  icSpliceSTVar,
  icSpliceEVar,
  icCopySTVar,
  icCopyEVar,
  icEnd
} iCmd;

typedef enum BracketType {
  ibOpenBracket,
  ibOpenCall,
  ibCloseBracket,
  ibCloseCall
} BracketType;
```

```
typedef struct ResultAction {
    iCmd cmd;
    void *ptr_value1;
    void *ptr_value2;
    int value;
} ResultAction;

// Interpreter

extern FnResult interpret_array(
  const ResultAction raa[],
  Iter allocs[],
  Iter begin,
  Iter end
);
```

Example. Stages 2 and 3 for the example with the function Fab (first sentence). Interpretation mode.

```
#ifdef INTERPRET
    const static refalrts::ResultAction raa[] = {
      {refalrts::icSpliceEVar, & eX_b_1, & eX_e_1},
      {refalrts::icIdent, (void*) & B<int>::name},
      {refalrts::icBracket, 0, 0, refalrts::ibOpenCall},
      {refalrts::icFunc, (void*) & Fab, (void*) "Fab"},
      {refalrts::icSpliceEVar, & eY_b_1, & eY_e_1},
      {refalrts::icBracket, 0, 0, refalrts::ibCloseCall},
      {refalrts::icEnd}
    };
    refalrts::Iter allocs[2*sizeof(raa)/sizeof(raa[0])];
    refalrts::FnResult res = refalrts::interpret_array( raa, allocs, arg_begin, arg_end );
    return res;
#else
    ...
#endif
```

§ 45. Run time support

The runtime support library contains the following components:

- Support for the language itself (files refalrts.h and refalrts.cpp) provides a simulation of the abstract refal machine and includes such functions:
  - definition of data structures of visual field units and interpreted instructions;
  - implementation of elementary operations of matching with the sample, allocation of memory and assembly of the result of the function;
  - memory allocation for new nodes, support for a list of free blocks;
  - dumping the field of view for debugging purposes;
  - main program cycle;
  - interpreter (for interpretation mode).
- The library of functions written in C ++ (the Library.cpp file) includes those functions that can not be written on Refal: input-output facilities, arithmetic operations, atomic transformation operations, etc.

The main program cycle

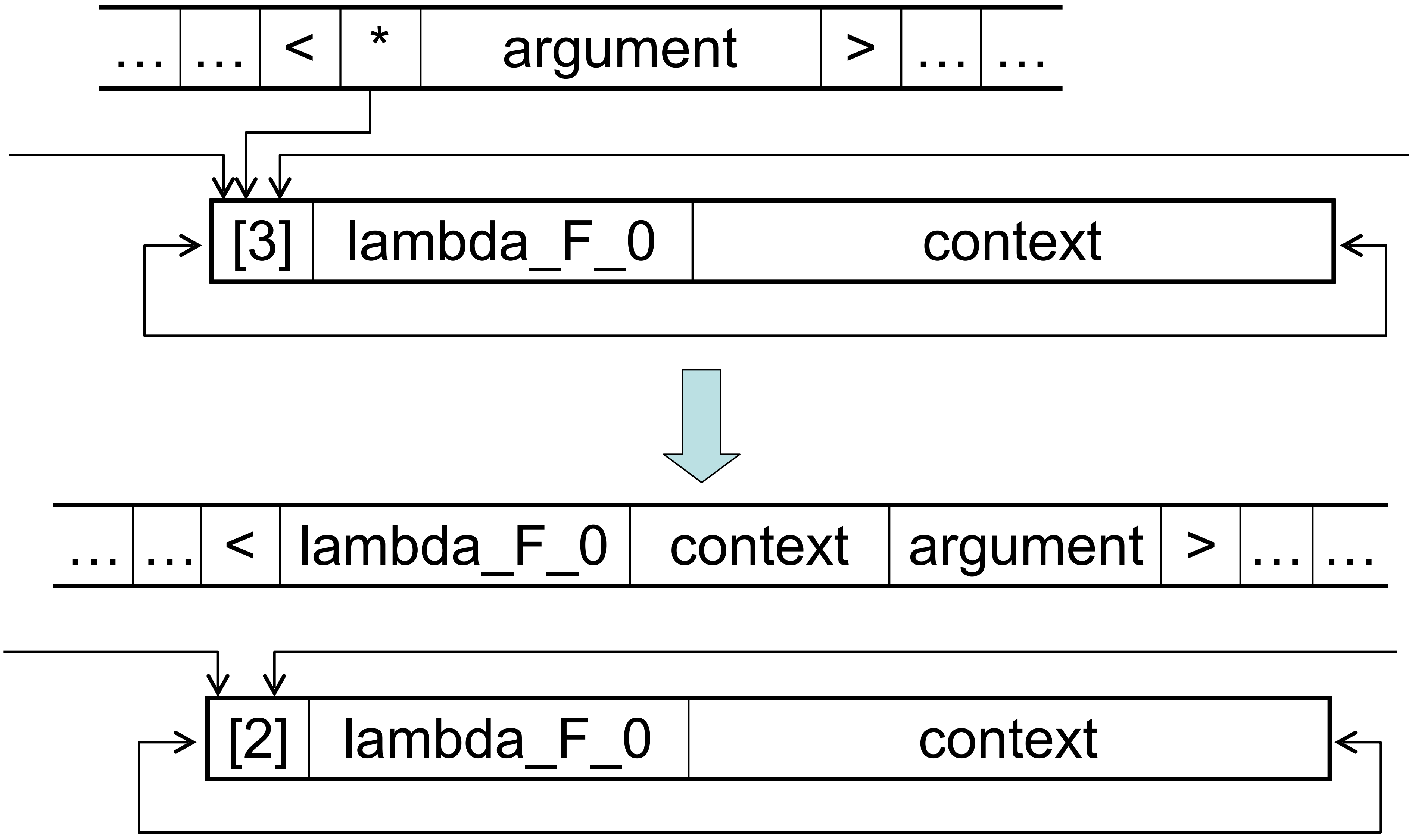
The runtime library simulates the operation of the refal machine in the so-called main program loop. Each iteration of the main loop corresponds to one step of the refal machine and is performed as follows:

- Finds the next primary active subexpression (as a pair of pointers to angle brackets).
- Defines the type of atom following the opening angle bracket. Depending on the type of this atom:
  - if there is an atom corresponding to the global function to the left of "<", then this function is called, and then the result of the function execution is analyzed: if it is different from refalrts :: cSuccess, the refall machine fails abnormally;
  - if there is a closure atom to the left of the <<", depending on the value of the reference counter, the contents of the closure (pointer to the global function + context) are copied (greater than 1) or (equal to 1) in the field of view to the place of the closing atom;
  - in other cases, an emergency stop of the refal machine takes place.

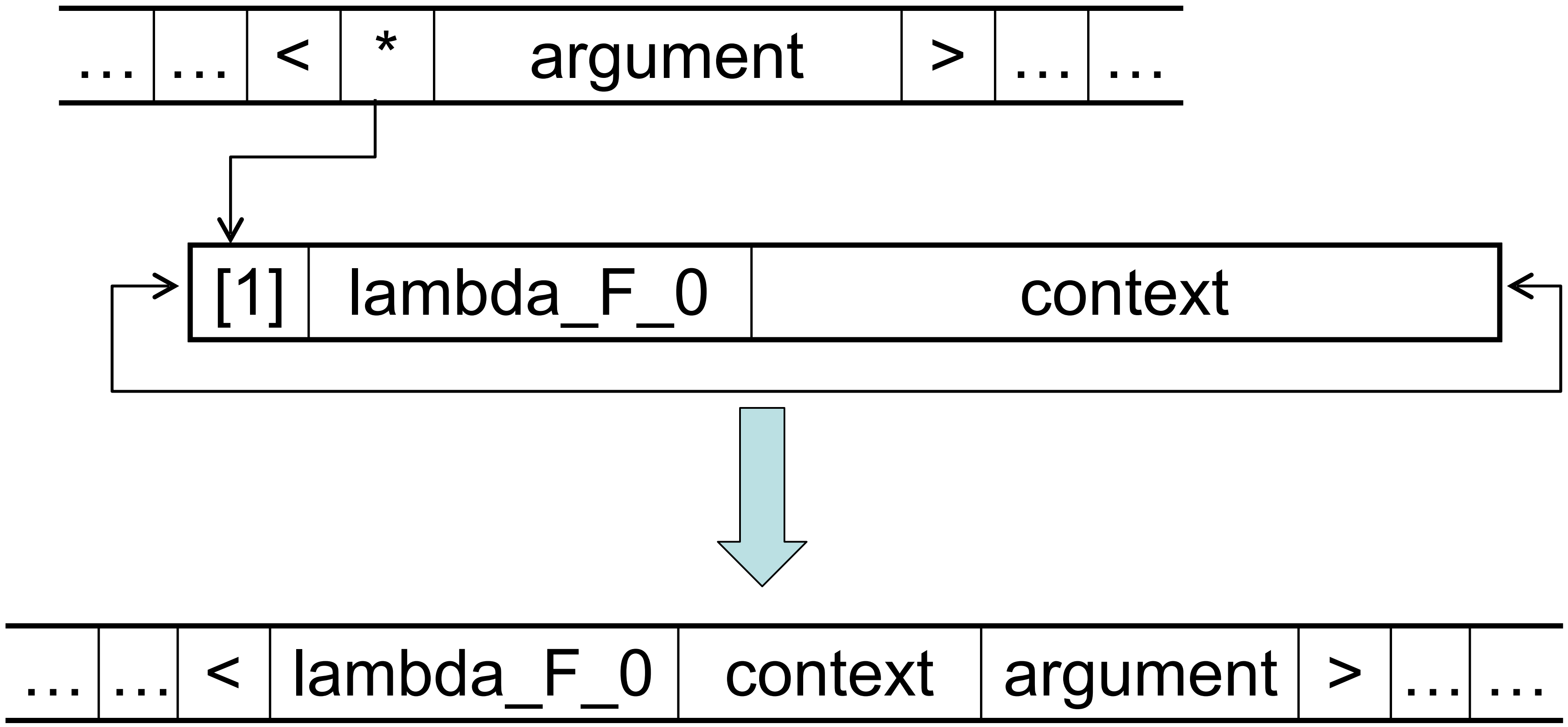
The main program cycle (pseudocode)

```
refalrts::FnResult main_loop() {
  while( g_stack_ptr != NULL )
  {
    arg_begin = pop_stack();
    arg_end = pop_stack();
    if( arg_begin->next->tag == cDataFunction ) {
      result = (*arg_begin->next->function_info.ptr)(arg_begin, arg_end);
      if( result != cSuccess ) {
        return result;
      } else {
        continue;
      }
    } else if( arg_begin->next->tag == cDataClosure ) {
      if( binding counter > 1 ) {
        copy the contents of the closure to the field of view;
        --binding counter;
      } else {
        move the contents of the closure in the field of view;
      }
      push_stack(arg_end);
      push_stack(arg_begin);
      continue;
    } else {
      return cRecognitionImpossible;
    }
  }
  return cSuccess;
}
```

Inserting the contents of the closure in the field of view



Inserting the contents of the closure in the field of view (continuation)


…
…
<
*
argument
>
…
…
[1]
lambda_F_0
context
…
…
<
lambda_F_0
context
argument
>
…
…