
Escola de Ciências e Tecnologia

Mestrado em Engenharia Informática

Tópicos Avançados de Compilação

# *A Specification of an Intermediate Representation for TACL*

*Author:*
Marlene Oliveira, Nº 11327

December 3, 2014

# Contents

# 1 Introduction

This assignment consisted in the creation of a specification of an intermediate representation (IR) for TACL. The starting point is the Abstract Syntax Tree (AST) of a TACL program. In order to create this specification, a Prolog program was created. Prolog was chosen because it simplifies the creation of the specification and the generation of the intermediate representation. The AST to which the intermediate representation is generated by the program is in the form of a Prolog term, which is read from the standard input. The environment used to create the specification was SWI-Prolog [1].

# 2 The Specification

As was mentioned in 1, the specification is in the form of a program. This specification aims to be able to generate the intermediate representation of a TACL program, given its AST. In order to make the code of the specification easier to read, it has been separated in five files:

- **tacl_ir.pl** - This file contains the Prolog predicates that check the node of the AST to which we want to generate the IR. These predicates pass along the relevant information to the predicates that actually generate the IR;
- **tacl_ir_exp.pl** - Which contains the Prolog predicates that generate the IR for the Expressions (Arithmetic, Boolean and Comparisons);
- **tacl_ir_LSV.pl** - Contains the Prolog terms that generate the IR for the Load and Store operations as well as for the Literals and Boolean expressions;
- **tacl_ir_stat.pl** - This file contains the Prolog predicates that generate the IR of the Statements (IFs, WHILEs, Assigns, Function Calls, etc);
- **tacl_ir_aux.pl** - Contains the auxiliary predicates of the specification (predicates to get the type of expressions, predicates to generate new labels or new temporary registers, etc).

The easiest way to test the specification is using *fun* predicates, since they output the respective intermediate representation. Alternatively, it is possible to test the specification using either the individual predicates (given the necessary arguments to do so) or the *rep* predicates. A brief explanation of how these predicates work can be seen in 2.1.

## 2.1 Identifying the Node of the AST

The generation of the intermediate representation is done in two steps. In the first step the node of the AST is identified and the relevant information is acquired and passed on to another predicate. In the second step, this last predicate uses the information acquired

to generate the IR for the AST node.

In order to identify the Node of the AST to which we are generating the IR, the *rep* predicates are used, the only exception being the predicate that identifies functions. This last predicate is named *fun*. The *rep* predicates have two arguments, one is the node of the AST and the other is a variable in which will be stored the last temporary used by the instruction. Generally, these predicates all work the same way. The argument corresponding to the AST node is checked in the *rep* predicates and the relevant information is acquired (type of instruction, arguments, etc). Save for an additional step for label generation executed when needed (for example, in an IF statement), the predicate corresponding to the node is called and the intermediate representation is generated.

To illustrate the way *rep* works, let us look at what happens when we pass the node *id* (identifier). An id node of the AST is characterized by three things: an identifier (the identifier itself), a kind (var, local or arg) and a type (int, real or bool). In order to verify that this node is, in fact, an id, we use the predicate *rep* described above. An example of the *rep* predicate mentioned can be seen in the next code snippet.

```
1 % Predicate that checks if the AST node is an identifier.
2 rep(EXP,T):-
3    EXP = id(X,KIND,TYPE):TYPE,
4    id(X,KIND,TYPE,T).
5
6 % An example of a predicate that generates the IR of an identifier.
7 id(ID,local,int,TC):-
8    tTemp(TC),
9    format('t~w <- i_lload @~w',[TC,ID]).
```

In our example, the expression of the identifier is passed as the first argument of the *rep* predicate ($EXP$). As can be seen in line 2, the node is identified as being an id node of the AST. After this, the relevant information[1] of the node is passed to the predicate that generates the representation. In this case, the predicate that generates the IR for the node id is the *id* predicate, which can be seen in line 7 of the code snippet provided.

To check if a node of the AST is a function it is only needed to call the *fun* predicate (shown in the next code snippet).

```
1 % Predicate that checks if the AST node is an identifier.
2 fun(FI,_,FB):-
3    format('function @~w\n',FI),
4    rep(FB,_),
5    resetAll(_).
```

[1]As mentioned before, "relevant information" usually consists of arguments necessary to the generation of the intermediate representation.

When this predicate is called, the first thing it does is write the header of the function in the intermediate representation. This header is comprised by two elements: the word "function" followed by the name of the function with the prefix "@". For example, if the name of the function is "fibonacci", it's header will be "function @fibonacci". The arguments of this predicate are the identifier of the function (its name), the formal arguments (which, are here represented by an underscore since, in the specification, nothing is generated for them) and the function body. The function body is passed as the argument of a *rep* predicate, that will identify the nodes of the AST and generate the corresponding IR. Finally, the *resetAll* predicate is used, which allows for a reset of all the counters after the representation is generated. This last step was created in order to make the debugging process easier. The *resetAll* predicate was ultimately kept as a means to automatically reset the counters without the need to consult the Prolog files of the specification each time a reset is needed.

## 2.2 The Intermediate Representation of the TACL Expressions

The vast majority of the predicates that generate the IR of the TACL expressions are included in the file named **"tacl_ir_exp"**. The only exception to this are the predicates that generate the intermediate representation of the boolean expressions *true* and *false*. These last predicates are in the same file as the specification for the identifiers, because its structure is similar to the one used to generate the intermediate representation for the literals.

### 2.2.1 Generating the Intermediate Representation of the Identifiers and Literals

As was previously mentioned, the predicates that generate the IR of the identifiers and of the literals are grouped in the file named **"tacl_ir_LSV"**.

After the node of the AST is identified as being an identifier, the *rep* predicate calls the predicate that generates the representation. These predicates have four arguments: the arguments that characterize an id (identifier, kind and type) and a variable that will store the index of the last temporary used by the instruction *load*. The predicates that generate the IR of the identifiers are named *id*. The *id* predicates generate the intermediate representation corresponding to the *load* instructions and there is one version of it for each kind of identifier (global variable, local variable and argument) and type (int or real). An example of a predicate of this sort and the *rep* predicate that calls it can be seen in the following code snippet.

```
1 % The predicate that identifies the node as being an id node.
2 rep(EXP,T):-
3    EXP = id(X,KIND,TYPE):TYPE,
4    id(X,KIND,TYPE,T).
5
```

```
% Example of a predicate that generates the IR for the load instruction (local
    variable).
id(ID,local,int,TC):-
   tTemp(TC),
   format('t~w <- i_lload @~w',[TC,ID]).
```

There are also predicates that generate the intermediate representation of the store instructions. These predicates are named *id_st* and have the same arguments as the *id* predicates (identifier, kind, type and a variable to store the temporary register generated after the intermediate representation of the store). When an assign is done, these predicates are used. As happens with the *id* predicates, the *id_st* predicates too have a version for each kind of identifier and each type. After the IR of the store instruction is written, a new temporary register is created in order to prevent reuse of the previous register[2]. The next code snippet includes an example of one of these predicates and the *rep* predicate that calls it.

```
% The predicate that identifies the node as being a node corresponding to a
    store (usually used in assign statements).
rep(EXP,T):-
   EXP = id(X,KIND,TYPE),
   id_st(X,KIND,TYPE,T).

% Example of a predicate that generates the IR for the store instruction (local
    variable).
id_st(ID,local,int,X):-
   tTemp(TC),
   format('@~w <- i_lstore t~w',[ID,TC]),
   newTemp(X).
```

The literals are also represented in the intermediate representation. In order to generate the IR for the literals, two predicates were created *int_literal* and *real_literal*. These predicates have two arguments: the value of the literal and a variable that will hold the index (value of the counter after the representation of the literal is completed) of the last temporary register used by the instruction. Both the *rep* predicate and an example of a predicate that generates the intermediate representation of the int literals can be seen in the next code snippet.

```
% Predicate that identifies the node as being a literal.
rep(EXP,T):-
   EXP = int_literal(X):int,
   int_literal(X,T).
```

[2]This is only done so the counter of the temporary registers can go forward in this step and the previous register isn't improperly reused after the store instruction.

```
5
6  % Predicate that generates the IR of the literal.
7  int_literal(ID,TC):-
8     tTemp(TC),
9     format('t~w <- i_value ~w',[TC,ID]).
```

### 2.2.2 Generating the Intermediate Representation for *true* and *false*

The predicates that generate the IR for *true* and *false* receive as arguments a letter that identifies the expression (either a $t$ or an $f$) and a variable that will hold the last temporary register used. The specification considers that a *true* corresponds to loading the number 1 into a temporary register and a *false* corresponds to loading the number 0 into a temporary register. The predicate that generates the intermediate representation for the *true* expression can be seen in the following code snippet (the predicate that generates the IR for the *false* expression is similar).

```
1  % Predicate that generates the IR of true.
2  bool(t,T):-
3     tTemp(T),
4     format('t~w <- i_value ~w',[T,'1']).
```

### 2.2.3 Generating the Intermediate Representation for the Arithmetic and Comparison Expressions

To make this part of the specification easier (and shorter), a simplification was used here. Instead of having a few predicates that are similar with the exception of the operation, the *bin_op* predicate was created. This predicate receives as input two expressions (the arguments of the operation), the type of the operation, the operation[3] and a variable that will hold the last temporary used. There are two *bin_op* predicates: one for integer operations and another for real operations. This predicate is both used to most of the arithmetic expressions (the ones with two operands) and to the comparison expressions. An example of these predicates can be seen in the next code snippet.

```
1  % Predicate that generates the IR for the binary operations
2  bin_op(A,B,int,OP,TN):-
3     rep(A,X),
4     write('\n'),
5     newTemp(_),
6     rep(B,XX),
```

[3]The operation is in the form $x_op$, in which $x$ can be $i$ (if the type of the operation is int) or $r$ (if the type of the operation is real) and $op$ is an operation (sum, multiplication, equal comparison, less or equal comparison, etc).

```
7     newTemp(_),
8     tTemp(TN),
9     format('\nt~w <- ~w t~w, t~w',[TN,OP,X,XX]).
```

As can be seen, after each of the intermediate representations of each of the expressions are generated, there's an increment in the counter of the respective temporary register. This happens in order to prevent improper use of temporary registers (using a temporary register whose value is still needed). The last step is writing the IR of the given operation. Not all expressions have their IR generated by these predicates. The *inv* and *toreal* are examples of expressions that have separate predicates to generate its IR.

The predicate that generates the intermediate representation of the *inv* instruction has the following arguments: the expression to which the inverse instruction should be applied, the type of the operation (int or real), the operation to write in the IR (with the correct type already associated) and a variable in which will be stored the value of the last temporary register used by the instruction. In order to generate the intermediate representation for this expression, the first step is to generate the intermediate representation of the expression that will be inversed (calling the *rep* predicate to do this). Then it is created a new floating-point temporary that will hold the value returned by the *inv* instruction. Finally, the IR of this instruction is generated.

It is used a similar method to the one used before to generate the IR of the *toreal* expression. The predicate that generates the intermediate representation for this expression has the following arguments: the expression to convert and a variable that will hold the value of the last temporary used by the instruction. In this case, that temporary will be a floating-point temporary. Similarly to what happens with the *inv* instruction, the generation of the intermediate representation of the *toreal* instruction also starts with the generation of the IR of the expression to convert. After this step is completed, a new floating-point temporary register is created in order to save the result of the coercion. A new temporary register of type int is also created in order to prevent incorrect reuse of a previous register that might be needed later. The last step is the generation of the IR for the instruction that performs the conversion (*itor*). These predicates are also included in the **"tacl_ir_exp"** file.

### 2.2.4 Generating the Intermediate Representation for the Logical Expressions

The logical expressions include the OR, AND and NOT expressions. The labels used in these predicates are generated in the corresponding *rep* predicates.

The predicate that generates the intermediate representation for the OR expression has five arguments: the two expressions that are arguments of the OR, the two labels (for the cases in which the OR is true and for the cases in which it is false) and the variable that'll hold the last temporary register used. In the OR expression, there's a copy instruction that's used to copy the value that needs to be preserved in order to be passed as argument for the next instruction. After the IR is generated, a reset is performed in order to prevent improper reuse of temporaries. The predicate that generates the IR of the AND expression

is similar, the only difference being in the labels (the AND expression evaluates the second expression given as argument if the first argument is true). The NOT expression negates the given expression.

*or_cond* and *and_cond* are predicates that generate the intermediate representation of AND or OR instructions used in the condition of IF or WHILE statements. The IR generated by these predicates differs from the one generated by the OR and AND predicates in the sense that it doesn't include the copy instruction and the last label. In an OR inside a condition, its label true is the same as the label true of the IF or WHILE statement of which the condition is part. Similarly, in the case of an AND expression, the label false is the same of the one from the statement of which this AND is condition. These predicates are called in the *cond* predicates explained in 2.3.3.

```
%------ Predicate that generates the IR for the OR expression
or(A,B,LT,LF,X):-
  rep(A,X),
  format('\ncjump t~w, ~w, ~w\n~w:\n',[X,LT,LF,LF]),
  newTemp(_),
  newFTemp(_),
  rep(B,TN),
  format('\nt~w <- i_copy t~w\n~w:',[X,TN,LT]),
  resetT(X).

%------ Predicate that generates the IR for the OR expression used in
    conditions (from IF, WHILE, etc)
or_cond(A,B,LT,LF,TN):-
  rep(A,X),
  format('\ncjump t~w, ~w, ~w\n~w:\n',[X,LT,LF,LF]),
  newTemp(_),
  newFTemp(_),
  rep(B,TN).

%------ Predicate that generates the IR for the AND expression
and(A,B,LT,LF,X):-
  rep(A,X),
  format('\ncjump t~w,~w,~w\n~w:\n',[X,LT,LF,LT]),
  newTemp(_),
  newFTemp(_),
  rep(B,TN),
  format('\nt~w <- i_copy t~w\n~w:\n',[X,TN,LF]),
  resetT(X).

%------ Predicate that generates the IR for the AND expression used in
    conditions (from IF, WHILE, etc)
and_cond(A,B,LT,LF,TN):-
  rep(A,X),
```

```
32    format('\ncjump t~w,~w,~w\n~w:\n',[X,LT,LF,LT]),
33    newTemp(_),
34    newFTemp(_),
35    rep(B,TN).
36
37  %------ Predicate that generates the IR for the NOT expression
38  not(A,T):-
39    rep(A,X),
40    newTemp(TN),
41    format('\n~w <- not t~w',[TN,X]),
42    tTemp(T).
```

## 2.3 Intermediate Representation of the TACL Statements

The predicates that generate the intermediate representation of the statements of TACL are grouped in the **"tacl_ir_stat"** file. Additionally, this file also includes the predicates needed to process declarations of variables, compound statements, function and procedure bodies, function return statements, print statements and assert statements.

### 2.3.1 Generating The Intermediate Representation for the IF Statement

There are two IF statements: one in which there is an ELSE after the IF and another one in which there is no ELSE after the IF[4]. In both statements the condition is the first element of which the intermediate representation is generated. It is followed by the *cjump* instruction of the IF, that uses the value returned by the condition and two labels (the label to which the program should branch if the condition is true and the label to which the program should branch if the condition is false). The predicates that generate the intermediate representation of the conditions are explained in 2.2.4. The next step in the IR generation consist in writing the label to the case in which the condition of the IF is true and the intermediate representation of the expressions that should be executed. After this part, and if we're generating the IR to an IF with an ELSE, there will be added to the IR a *jump* instruction, that allows the program to jump to the end of the IF statement. If we're dealing with an IF without an ELSE, this is not written in the IR because the label used when the condition returns false coincides with the end of the IF. Each time new temporaries are created in these predicates it is done to avoid improper reuse of temporary registers that still are needed. The labels are generated in the *rep* predicate of the IF statement. The Prolog terms that generate the IR of both IFs and the corresponding *rep* predicates can be seen in the next code snippet.

```
1  % Predicate that identifies EXP as an IF Without ELSE
2  rep(EXP,T):-
3     EXP = if(A,B,nil),
```

[4]In this case, the statements in the ELSE are represented by *nil*.

```
4     new_label(LT),
5     new_label(LF),
6     if(A,B,nil,LT,LF,T).
7
8  % Predicate that generates the intermediate representation of an IF Without
       ELSE statement
9  if(C,A,nil,LT,LF,T):-
10    cond(C,LT,LF,X),
11    format('\ncjump t~w, ~w, ~w',[X,LT,LF]),
12    format('\n~w:\n',LT),
13    newTemp(_),
14    newFTemp(_),
15    rep(A,T),
16    format('\n~w:\n',LF).
17
18 % Predicate that identifies EXP as an IF With an ELSE
19 rep(EXP,T):-
20    EXP = if(A,B,C),
21    new_label(LT),
22    new_label(LF),
23    if(A,B,C,LT,LF,T).
24
25 % Predicate that generates the intermediate representation of an IF With an ELSE
26 if(C,A,B,LT,LF,T):-
27    cond(C,LT,LF,X),
28    format('\ncjump t~w, ~w, ~w',[X,LT,LF]),
29    format('\n~w:\n',LT),
30    newTemp(_),
31    newFTemp(_),
32    rep(A,_),
33    new_label(LFF),
34    format('\njump ~w',LFF),
35    format('\n~w:\n',LF),
36    rep(B,T),
37    format('\n~w:\n',LFF).
```

### 2.3.2 The Intermediate Representation for the WHILE Statement

The predicate that generates the intermediate representation for the WHILE statement is rather simple. The first step is to write the label at the beginning of the WHILE and then generating the IR for the condition of this statement. When this is done, the value returned by the condition is used in a *cjump* instruction of the intermediate representation in order to determine where to jump next (either to the body of the WHILE or to the end of the WHILE). The next step is writing the corresponding label of the WHILE and the

intermediate representation of the statements that should be executed if the condition is true. After this, it is added to the IR a *jump* instruction that allows for the branch to the beginning of the WHILE statement. Finally, the label to which the program should jump if the condition is false is written at the end of the WHILE statement. New temporaries are created before the generation of the statements that should be executed if the condition of the WHILE is true. This is done to prevent incorrect use of temporary registers that are still needed. The labels used in the generation of the intermediate representation of the WHILE statement are generated in the corresponding *rep* predicate. The predicate that generates the intermediate representation of the WHILE statement can be seen next.

```
1  % The predicate that identifies the node as being a WHILE statement
2  rep(EXP,T):-
3     EXP = while(A,B),
4     label(L),
5     atom_concat('l',L,LS),
6     new_label(LT),
7     new_label(LF),
8     while(A,B,LS,LT,LF,T).
9
10 % Predicate that generates the IR of the WHILE statement.
11 while(A,B,LS,LT,LF,T):-
12    format('~w:\n',LS),
13    cond(A,LT,LF,X),
14    format('\ncjump t~w, ~w, ~w',[X,LT,LF]),
15    newTemp(_),
16    newFTemp(_),
17    format('\n~w:\n',LT),
18    rep(B,T),
19    format('jump ~w\n~w:\n',[LS,LF]).
```

### 2.3.3 Generating the Intermediate Representation for the Conditions of the IF and WHILE Statements

In order to make understanding the IR generation easier, an auxiliary predicate is used to distinguish the kind of condition (from an IF or WHILE statement) to which the IR is going to be generated. Therefore, there are a few predicates that check the kind of condition used, which is done in a way similar to the one used in the *rep* predicates. These predicates are named *cond*. One of these predicates check if the condition is an OR and generate the necessary internal labels for this kind of condition. In an analogous way, there is one *cond* predicate that does the same for the AND condition. The third predicate of this kind simply passes the condition to a *rep* predicate, because this will allow the generation of the IR of every other possible conditions (comparison expressions or boolean expressions). The $or_cond$ and $and_cond$ predicates are explained in the section 2.2.4.

```
% Checks if the condition is an OR
cond(C,LT,_,T):-
  C = or(X,Y):bool,
  new_label(LF),
  or_cond(X,Y,LT,LF,T).

% Checks if the condition is an AND
cond(C,_,LF,T):-
  C = and(X,Y):bool,
  new_label(LT),
  and_cond(X,Y,LT,LF,T).

% Checks if the condition is anything else
cond(C,_,_,T):-
  rep(C,T).
```

### 2.3.4 Generating the Intermediate Representation for the Function and Procedure Calls

The *pcall* and $fcall$ predicates generate the intermediate representation for procedure and function calls, respectively. The first step is to reset the list of arguments to prevent reuse of arguments from previous functions or procedure calls[5]. The next step is generating the intermediate representation for the arguments and building the list of arguments used by the call. After this, the intermediate representation of the *call* instruction is generated and the corresponding temporary register is incremented. Finally, the list of arguments of the call is reset, because it is no longer needed. The *rep* predicate used to select which is the function call is similar to the other ones. The *rep* predicate for the procedure calls adds the gathering of the type of the procedure (using the $arg[2]$ predicate from SWI Prolog and the auxiliary predicate $get_type$). An example of these predicates is available in the next code snippet.

```
% Predicate that identifies the node of the AST as being a Function Call
rep(EXP,T):-
  EXP = call(X,Y):TYPE,
  fcall(X,Y,TYPE,T).

% Predicate that identifies the node of the AST as being a Procedure Call
rep(EXP,T):-
  EXP = call(X,Y),
  arg(1,Y,TY),
  get_type(TY,TYPE),
  pcall(X,Y,TYPE,T).
```

[5]These lists are initialized when the program is compiled and changed when needed (the old lists are replaced by new ones or even reset).

```
12
13 % Predicate that generates the IR of the Procedure Call
14 pcall(A,B,int,T):-
15   resetALT(_),
16   callArg(B,_),
17   arg_listT(K),
18   format('call @~w, ~w',[A,K]),
19   newTemp(_),
20   tTemp(T),
21   resetALT(_).
22
23 % Predicate that generates the IR of the Function Call
24
25 fcall(A,B,int,T):-
26   resetALT(_),
27   callArg(B,_),
28   arg_listT(K),
29   sort(K,KK),
30   newTemp(XN),
31   format('~w <- call @~w, ~w',[XN,A,KK]),
32   resetALT(_),
33   tTemp(T).
```

### 2.3.5 Generating the Intermediate Representation of the Assign Statements

The predicate that generates the intermediate representation of an assign statement has three arguments: the identifier of the variable to which a value will be assigned, the expression whose value will be assigned to the identifier and a variable that will hold the last temporary used. When this predicate is called, it first generates the intermediate representation of the expression of which value will be assigned. The next step is generating the IR of the identifier of the variable to which the value of the expression will be assigned. In this particular case, when the program generates the intermediate representation of the identifier, it reads the current temporary register from the predicate $tTemp$ or $fTemp$ (depending on the type of the instructions)[6].

```
1 % Predicate that generates the IR of the Assign Statement
2 assign(A,B,X):-
3   rep(B,_),
4   write('\n'),
5   rep(A,X).
```

[6]To review this last part, please refer to 2.2.1

### 2.3.6 Generating the Intermediate Representation of the Print Statements

To generate the intermediate representation of the *print* statement, the predicate *print* is used. This predicate has three arguments: the expression to print, the type (int, real or bool) and a variable that will hold the value of the current temporary register. When generating the intermediate representation of this statement, the first step is to generate the IR of the expression that will be printed, which is done by calling the *rep* predicate. The following step consists in generating the IR of the *print*, using the temporary register provided by the previous predicate. The last step consists in an increment of the corresponding temporary register, which occurs so there is no improper reuse of temporaries later. There are three of these predicates, one for each data type (int, real, bool).

```
% Example of a predicate that generates the IR of the print statements
print(A,int,T):-
  rep(A,TN),
  format('\ni_print t~w',TN),
  newTemp(T).
```

### 2.3.7 Generating the Intermediate Representation for the Compound Statements

When generating the intermediate representation for a compound statement (list of statements) there can be used two predicates: one in which the list is empty and another one in which the list is not empty. In the first case, no intermediate representation is generated (nothing is done). On the second case, it is checked if the list is not empty. If this is true, the auxiliary predicates created to manipulate lists are used to isolate the head of the list in order to generate the IR for this element. After this is done, the auxiliary predicates are used to remove the head of the list (which was already processed) and the IR of the other elements of the list is then generated.

```
% The predicates that generate the IR of the compound statements
comp(A,T):-
  A\=[],
  gethead(A,C),
  rep(C,_),
  write('\n'),
  removehead(A,B),
  rep(B,T).

comp([],_).
```

### 2.3.8 Generating the IR of the Declarations

There are two predicates that generate the intermediate representation of the declarations: one to generate the IR of the declaration when the expression is *nil* and another one to generate the IR when the expression isn't *nil*. On the second case, the predicate checks if the expression of the declaration is *nil* and, if not, the intermediate representation is generated. When the second argument of the *var* predicate is *nil*, nothing is generated.

```
% The predicates that generate the IR of the declarations
var(ID,A,T):-
  A\=nil,
  rep(A,_),
  write('\n'),
  rep(ID,T).

var(_,nil,_).
```

### 2.3.9 Generating the IR of the Function Bodies, Procedure Bodies and Return Statements

The predicates that generate the bodies of the functions and procedures are similar. The first just generates the IR of each part of the function (declarations, body and return expression) using the *rep* predicates. This predicate uses the auxiliary predicate *get_type* to obtain the type of the return statement and then invokes the correct predicate that generates the intermediate representation of the return expression.

The predicate that generates the IR of the procedure bodies is very similar to the one described above, but the argument that corresponds to the return expression is *nil*. The generation of the intermediate representation of the declarations and statements of the procedure is done using the *rep* predicate.

There are two predicates that generate the intermediate representation of the return statement: one for the returns of type int and another for the returns of the functions of type real. The only thing this predicate does is generate the return statement of the correct type.

```
% Predicate that generates the IR of a body of a function
body(D,S,E,T):-
  E\=nil,
  rep(D,_),
  rep(S,_),
  rep(E,T),
  get_type(E,TYPE),
  return(TYPE).

% Predicate that generates the IR of a body of a procedure
```

```
body(D,S,nil,T):-
  rep(D,_),
  write('\n'),
  rep(S,T).

% Example of a predicate that generates the IR of the Return Statement of a
    function
return(int):-
  tTemp(TN),
  format('\ni_return t~w',TN).
```

## 2.4 The Auxiliary Predicates of the Specification

In order to be able to generate an IR of a TACL program, there are some extra predicates that are necessary, such as the predicates that generate new labels and new temporaries, among others. These are especially important because they produce information used by everything else throughout the IR generation. When a new temporary register, either integer or floating-point, is needed, we call a predicate that generates a new temporary register. The last register created, i.e. the register in which the last operation stopped, is saved in a default predicate and placed in a variable in the predicate that generates the IR. There are two predicates that generate temporary registers: one for integer temporaries (named $tTemp$) and one for floating-point temporaries (named $fTemp$). These predicates can be called when we need to know the register in which the last operation stopped (that is, if we don't have information from a previous instruction). There is also one default predicate to store the last label generated (named *label*). When the Prolog files of the specification are consulted, there are default values generated for the predicates that store the information related to the temporary registers and the labels[7].In order to allow changing the values of the temporaries, predicates to reset the counters were created. These predicates can reset the counters of the temporary registers to the default values or to a given value (start in a given value X). There are no predicates to reset solely the counter of the labels[8]. Examples of predicates that generate new temporaries and labels and the corresponding predicates that reset the counters can be seen in the next code snippet.

```
% Counter for Integer Temporaries
newTemp(K):-
   tTemp(X),
   TN is X + 1,
   retract(tTemp(X)),
   asserta(tTemp(TN)),
   atom_concat('t',TN,K).
```

[7]The default value is zero (all counters are reset).

[8]This counter can only be reset using the *resetAll* predicate.

```
8
9  % Reset Integer Temporary Counter completely (start in zero)
10 resetallT(_):-
11    retractall(tTemp(_)),
12    assert(tTemp(0)).
13
14 % Reset the Integer Temporary Counter to start in a given temporary
15 resetT(X):-
16    retractall(tTemp(_)),
17    assert(tTemp(X)).
18
19 % Predicate that generates a new Label
20 new_label(N):-
21    label(P),
22    X is P + 1,
23    retract(label(P)),
24    asserta(label(X)),
25    atom_concat('l',X,N).
26
27 % Predicate that resets all counters, labels and lists to default values
28 resetAll(_):-
29 retractall(tTemp(_)),
30 retractall(fTemp(_)),
31 retractall(label(_)),
32 retractall(arg_listT(_)),
33 retractall(arg_listF(_)),
34 assert(tTemp(0)),
35 assert(fTemp(0)),
36 asserta(label(0)),
37 asserta(arg_listT([])),
38 asserta(arg_listF([])).
```

The predicate *newTemp* creates a new integer temporary by reading the old value, increments it by one and replaces it by the new value. The argument of this predicate, $K$ serves as a return value, which helps when other instructions need to have this information in order to generate the IR correctly. In some cases, this predicate is used solely to advance the counter, thus preventing the reuse of temporaries.

*resetallT* was created to reset all the counters for the integer temporaries[9]. This predicate removes from the database the predicate which has the old value and replaces it with a predicate with the default value (zero). *resetT* is a predicate very similar to this one. The only difference between these two predicates is the fact that *resetT* adds a new predicate with a given value, so that the temporaries can start from a value that isn't zero. For example, if we call $resetT(2)$, the new predicate will be $tTemp(2)$. In this example, the

[9]There is also a version of this predicate to the floating-point temporaries.

counter would start in $t2$ instead of starting in $t0$.

The predicate used to generate a new label is very similar to those used to generate a new temporary register. The only difference between these predicates is what is "returned" in its only argument. This predicate "returns" an atom with "l-" in which the symbol "-" will be a number, for example "l0".

Aside from these, there are other auxiliary predicates that are also very helpful. These predicates are: *get_type*, *argsList* and *callArg*. The predicate *get_type* retrieves the type of an expression. *argsList* is used when generating the lists of arguments used in function or procedure calls. To do this, the predicate retrieves the old list, adds the new argument to that list and then replaces the old list by the new one. The predicate *callArg* generates the intermediate representation of the arguments of a function or procedure call. These predicates can be consulted in the next code snippet.

```
% Auxiliary predicate to get the type of an expression
get_type(X,C):-
   arg(2,X,C).

% Auxiliary predicate to manage function call arguments
argsList(_):-
   tTemp(X),
   fTemp(Y),
   atom_concat('t',X,TA),
   atom_concat('fp',Y,FA),
   arg_listT(LTe),
   arg_listF(FTe),
   retract(arg_listT(LTe)),
   retract(arg_listF(FTe)),
   append(LTe,[TA],KA),
   append(FTe,[FA],KB),
   asserta(arg_listT(KA)),
   asserta(arg_listF(KB)).

callArg(A,T):-
   A\=[],
   gethead(A,C),
   rep(C,_),
   argsList(_),
   newTemp(_),
   newFTemp(_),
   write('\n'),
   removehead(A,B),
   rep(B,T).

callArg([],_).
```

# 3 Conclusions and Future Work

After testing the specification created[10] we can conclude that the generation of the intermediate representation seems to be working as intended. There are some minor bugs, such as some newlines on the output and some double increments of the counter of the temporary registers. However, since this doesn't affect the generation of the intermediate representation, these are not considered problematic. Future work would include fixing these minor bugs and extending the specification to eventual cases that were not considered in the current version.

# Appendices

# A Test Results

The test results presented in these appendixes consist in the IR generated by the specification created. The ASTs used in these tests are the ones provided on the course's Moodle page.

## A.1 Function *twice*

```
function @twice
t0 <- i_value 2
t1 <- i_aload @n
t2 <- i_mul t0, t1
i_return t2
```

## A.2 Function *example*

```
function @example

t0 <- i_value 1
@r <- i_lstore t0
l0:
t1 <- i_value 0
t2 <- i_aload @n
t3 <- i_lt t1, t2
cjump t3, l1, l2
l1:
```

[10] The results of these tests are available in the appendices of this report.

```
t4 <- i_lload @r
t5 <- i_aload @n
t6 <- i_mul t4, t5
@r <- i_lstore t6
t7 <- i_aload @n
t8 <- i_value 1
t9 <- i_sub t7, t8
@n <- i_astore t9
jump l0
l2:

t10 <- i_lload @r
i_return t10
```

## A.3 Function *triangular*

```
function @triangular


t0 <- i_aload @n
@a <- i_lstore t0
t1 <- i_value 0
t2 <- i_lload @a
t3 <- i_lt t1, t2
cjump t3, l1, l2
l1:
t4 <- i_aload @n
t5 <- i_value 1
t6 <- i_sub t4, t5
t8 <- call @triangular, [t6]
@b <- i_lstore t8
t9 <- i_lload @a
t10 <- i_lload @b
t11 <- i_add t9, t10
@a <- i_lstore t11

l2:

t12 <- i_lload @a
i_return t12
```

## A.4 Function *g*

```
function @g
t0 <- i_value 0
fp1 <- itor t0
@s <- r_lstore fp1
l0:
t1 <- i_value 1
t2 <- i_aload @n
t3 <- i_le t1, t2
cjump t3, l1, l2
l1:
fp3 <- r_lload @s
t4 <- i_aload @n
fp5 <- itor t4
fp6 <- r_add fp3, fp5
@s <- r_lstore fp6
t5 <- i_aload @n
t6 <- i_value 1
t7 <- i_sub t5, t6
@n <- i_astore t7
jump l0
l2:
fp7 <- r_lload @s
r_return fp7
```

## A.5 Function *factorial*

```
function @factorial
t0 <- i_value 1
@r <- i_lstore t0
t1 <- i_value 0
t2 <- i_aload @n
t3 <- i_lt t1, t2
cjump t3, l1, l2
l1:
t4 <- i_aload @n
t5 <- i_aload @n
t6 <- i_value 1
t7 <- i_sub t5, t6
t9 <- call @factorial, [t7]
t10 <- i_mul t4, t9
@r <- i_lstore t10
```

```
l2:
t11 <- i_lload @r
i_return t11
```

## A.6 Function *Code from Fig. 9 of the CCC - Parte 2*

```
function @sub
t0 <- i_aload @i1
t1 <- i_aload @i2
t2 <- i_sub t0, t1
i_return t2

function @fibonacci

t0 <- i_aload @n
t1 <- i_value 0
t2 <- i_eq t0, t1
cjump t2, l1, l3
l3:
t3 <- i_aload @n
t4 <- i_value 1
t5 <- i_eq t3, t4
cjump t5, l1, l2
l1:
t6 <- i_aload @n
@r <- i_lstore t6
jump l4
l2:
t7 <- i_aload @n
t8 <- i_value 1
t9 <- i_sub t7, t8
t11 <- call @fibonacci, [t9]
t12 <- i_aload @n
t13 <- i_value 2
t14 <- call @sub, [t12]
t16 <- call @fibonacci, [t14]
t17 <- i_add t11, t16
@r <- i_lstore t17
l4:
t18 <- i_lload @r
i_return t18

function @main
t0 <- load @n
```

```
t2 <- call @fibonacci, [t0]
@fib <- i_lstore t2
t3 <- i_lload @fib
i_print t3
t4 <- i_value 14
t5 <- i_value 3
t6 <- i_value 4
t7 <- i_mul t5, t6
t8 <- i_add t4, t7
t10 <- call @fibonacci, [t8]
i_print t10
t11 <- i_value 121393
i_print t11
t12 <- i_value 0
i_return t12
```

## A.7 Function *fibonacci (v2)*

```
function @fibonacci


t0 <- i_aload @n
t1 <- i_value 0
t2 <- i_eq t0, t1
cjump t2, l0, l1
l1:
t3 <- i_aload @n
t4 <- i_value 1
t5 <- i_eq t3, t4
t2 <- i_copy t5
l0:
@c <- i_lstore t2
t3 <- i_lload @c
cjump t3, l2, l3
l2:
t4 <- i_aload @n
@r <- i_lstore t4
jump l4
l3:
t5 <- i_aload @n
t6 <- i_value 1
t7 <- i_sub t5, t6
t9 <- call @fibonacci, [t7]
t10 <- i_aload @n
```

```
t11 <- i_value 2
t12 <- i_sub t10, t11
t14 <- call @fibonacci, [t12]
t15 <- i_add t9, t14
@r <- i_lstore t15
l4:

t16 <- i_lload @r
i_return t16
```

## A.8 Procedure

```
function @p

fp0 <- load @x
fp1 <- r_aload @y
fp2 <- r_mul fp0, fp1
@x <- r_store fp2
true .

function @main

fp0 <- r_value 1.5
@x <- r_store fp0
t0 <- i_value 3
fp2 <- itor t0
call @p, [fp2]
fp4 <- load @x
r_print fp4
```

# References


[1] S. Prolog, "Swi prolog." http://www.swi-prolog.org/, 2014.

[2] S. Prolog, "Predicate arg/3." http://www.swi-prolog.org/pldoc/man?predicate=arg/3, 2014.