Theosim v1.0.17

by
Achim Kern
Erik Thiele
Hardy Kahl
Philipp Cimiano Lavin

July 29, 2000

Contents

Chapter 1

Introduction

Theosim is a package useful to theoretical computer scientists only. For being able to prove fundamental and advanced topics of computer science they have invented a few languages for theoretical reasons mainly. Those languages are not useful for real programming, but you can use them to prove things.

The languages include Pascalli, Pascallino, and RAM. Theosim allows you to compile $PASCALLI \Rightarrow PASCALLINO \Rightarrow RAM$. It also allows you to simulate RAM programs with a graphical user interface.

Chapter 2

A Sample Session

- We want to calculate the faculty of 5.
- First we start theosim
- We open a new file named "faculty.pli" (.pli means Pascalli)
- We type the following code:

```
function faculty(x);
begin
  if (x = 0) then begin
    faculty := 1
  end else begin
    faculty := (x * faculty ((x - 1)))
  end
end
```

- We select Tasks/Pascalli->Pascallino
- Now we see the resulting Pascallino program. We notice its complexity, and believe in its truth.
- We do Tasks/Pascallino->RAM
- Now we see the RAM program. This time we really *have to* believe in its truth, because it is very complex.
- We do Tasks/Ramsim.
- Ramsim pops up. We press SetReg and set the register 0 to the value 5.
- Press the Run button.
- Once finished, you will see the faculty of 5 in Register 0. (The faculty of 5 is 1*2*3*4*5=120)

Chapter 3

The Languages

There are three languages used in the theosim package. These are:

3.1 PASCALLI

```
function example(arg1, arg2);               declaration of the
                                            function "example" with
                                            two formal parameters
begin                                       begin of a commandblock
                                            (function-body)
  variable:=5;                              assignment of the value "5"
                                            to the variable "variable"
  for count:=1 to variable do               for-loop
  begin                                     begin of a commandblock
                                            (for-loop-body)
   if (count = 1) then                      if-else-statement
     begin                                  begin of a commandblock
                                            (if-body)
       array[1]:=sum(arg1,arg2)             call of the function sum
                                            with arg1 and arg2 as
                                            arguments and assignment
                                            of the return-value to
                                            the field "1" of the
                                            array "array"
     end                                    end of the if-body
   else                                     begin of a commandblock
     begin                                  (else-body)
       while (count <=5) do                 while-loop
       begin                                begin of a commandblock
                                            (while-loop-body)
         array[count]:=(array[count]*arg1)  assign the product of
                                            array[count] and arg1
                                            to array[count]
       end                                  end of the while-loop-body
     end                                    end of the else-body
    end                                     end of the if-else-body
  end;                                      end of the for-loop-body
  example:=array[1]                         assign the value of array[1]
                                            to the variable "example"
```

```
                                            as return-value of the
                                            function "example"
end                                         end of the function "example"

function sum(arg1, arg2);                   declaration of the function
                                            "sum" with two formal
                                            parameters
begin                                       begin of a commandblock
                                            (function-body)
  sum:=(arg1+arg2)                          assign the sum of arg1 and
                                            arg2 to the variable "sum"
                                            as return-variable of the
                                            function "sum"
end
```

Pascalli is a high-level programming language originally defined by K.W. Wagner: "Einführung in die Theoretische Informatik - Grundlagen und Modelle", Springer 1994.

It is rather simple and useful to analyze the computational power of high-level programming languages.

Though our version of Pascalli is based on Wagner we have tried to create a more realistic programming language than he does by adding some restrictions.

As Pascalli is a language with no explicit variable declarations, and for this there is no possibility of adjoining a concrete variable to a certain type (variable/array), we have decided to forbid the use of an identifier "declared" (that means: first used) as a simple variable as an array and viceversa.

In this sense we also expect functions to be unique, meaning that they are to be "declared" only once with a fixed number of formal parameters. We decided to forbid the "call" of functions that are not declared and expect functions to have a return value.

We also defined the main function to be the first function in a Pascalli source and not the last one as defined by Wagner.

The syntax of Pascalli is as follows (EBNF):

```
PASCALLI_PROGRAM ::= FUNCTION_DECLARATION  PASCALLI_PROGRAM
                   | FUNCTION_DECLARATION

FUNCTION_DECLARATION ::= "function" VARIABLE
                       "(" PARAMETERLIST ")" ";" COMMAND
```

Note:

1. A VARIABLE is a non-empty finite sequence of symbols, that can be digits 0, ..., 9 or characters a, ..., z. However, the first symbol must be a character. It is forbidden to use the following key-words as a VARIABLE:

   begin, end, function, if, then, else, for, to, do, while, not, and, or

2. It is not allowed to "redeclare" a function.

3. The PARAMATERLIST can be empty.

4. As the function is expected to have a return value there has to be an assignment of the form: VARIABLE := ... within COMMAND

```
PARAMETERLIST ::= VARIABLE "," PARAMETERLIST
                | VARIABLE
                | epsilon
```

Note: It is not allowed to use a VARIABLE twice in the PARAMETERLIST.

```
COMMAND ::= VALUE_ASSIGNMENT
          | COMMANDBLOCK
          | IF_COMMAND
          | FOR_LOOP
          | WHILE_LOOP

VALUE_ASSIGNMENT ::= VARIABLE ":=" EXPRESSION
                   | ARRAY    ":=" EXPRESSION
```

Note: It is not allowed to use as VARIABLE an identifier that has been used within an ARRAY before.

```
COMMANDBLOCK ::= COMMAND ";" COMMANDBLOCK
               | COMMAND

IF_COMMAND ::= "if" CONDITION "then" COMMAND "else" COMMAND
             | "if" CONDITION "then" COMMAND

FOR_LOOP ::= "for" VARIABLE ":=" EXPRESSION "to"
             EXPRESSION "do" COMMAND
```

Note: The VARIABLE mustn´t appear on the left side of an assignment within COMMAND

```
WHILE_LOOP ::= "while" CONDITION "do" COMMAND

EXPRESSION ::= VARIABLE
             | CONSTANT
             | ARRAY
             | "(" EXPRESSION "+" EXPRESSION ")"
             | "(" EXPRESSION "-" EXPRESSION ")"
             | "(" EXPRESSION "*" EXPRESSION ")"
             | "(" EXPRESSION "/" EXPRESSION ")"
             | FUNCTION_CALL
```

Note:

1. It is not allowed to use an identifier as a VARIABLE that has been used as an ARRAY before.
2. A CONSTANT is a non-empty finite sequence of digits 0, . . . , 9

```
ARRAY ::= VARIABLE "[" EXPRESSION "]"
```

Note: It is not allowed to use an identifier as VARIABLE that has been used as a "simple" variable before.

```
FUNCTION_CALL ::= VARIABLE "(" EXPRESSIONLIST ")"
```

Note: It is not allowed to "call" a function that has not been "declared" before.

```
EXPRESSIONLIST ::= EXPRESSION "," EXPRESSIONLIST
                 | EXPRESSION
                 | epsilon
```

Note: The EXPRESSIONLIST can be empty.

```
CONDITION ::= "(" EXPRESSION "<"   EXPRESSION ")"
            | "(" EXPRESSION "<="  EXPRESSION ")"
            | "(" EXPRESSION "="   EXPRESSION ")"
            | "(" EXPRESSION "#"   EXPRESSION ")"
            | "(" EXPRESSION ">="  EXPRESSION ")"
            | "(" EXPRESSION ">"   EXPRESSION ")"
            | "(" CONDITION  "and" CONDITION ")"
            | "(" CONDITION  "or"  CONDITION ")"
            | "not" "(" CONDITION ")"
```

3.2 PASCALLINO

```
function example(arg1, arg2);     definition of the function "example"
                                  with two formal parameters
begin                             begin of a commandblock (function-body)
  variable:=arg1;                 the value of "arg1" is assigned to the
                                  variable "variable"
  one:=1;                         the constant "1" is assigned to the
                                  variable "one"
  while (variable#0) do           while-loop
  begin                           begin of a commandblock (while-loop-body)
    array[variable]:=arg1;        assign the value of arg1 to the field
                                  "variable" of the array "array"
    variable:=(variable-one)      decrement "variable"
  end;                            end of the while-loop-body
  example:=array[one]             assign the value of array[one]
                                  to the variable "example" as
                                  return-value of the function
                                  "example"
end                               end of the function-body
```

Pascallino is a simplified version of Pascalli. It is quite similar to the RAM-language and useful as a middle step in the compilation of Pascalli to RAM.

As in Pascalli we expect identifiers to be uniquely adjoined to a type (variable/array) and also the program to have a return value. We also decided to extend the language definition of Wagner and allow assignments of the form VARIABLE := VARIABLE because there seems to be no reason to forbid such a construct as RAM does allow it. Furthermore, it turns out that the task of compiling Pascalli into Pascallino is simplified by this.

The syntax of Pascallino is as follows (EBNF):

```
PASCALLINO_PROGRAM ::= "function" VARIABLE
                       "(" PARAMETERLIST ")" ";" COMMAND
```

Note:

1. A VARIABLE is a non-empty finite sequence of symbols that can be digits 0, ..., 9 or characters a, ..., z. However, the first symbol must be a character. It is forbidden to use the following key-words as a VARIABLE:

   begin, end, function, while, do

2. The PARAMETERLIST can be empty.

3. As the function is expected to have a return value there has to be an assignment of the form: VARIABLE := ... within COMMAND

```
PARAMETERLIST ::= VARIABLE "," PARAMETERLIST
                | VARIABLE
                | epsilon
```

Note: It is not allowed to use a VARIABLE twice in the PARAMETERLIST.

```
COMMAND ::= CONSTANT_ASSIGNMENT
          | VALUE_ASSIGNMENT
          | COMMANDBLOCK
          | WHILE_LOOP
```

```
CONSTANT_ASSIGNMENT ::= VARIABLE ":=" CONSTANT
```

Note:

1. It is not allowed to use an identifier as a VARIABLE that has been used in an ARRAY before.
2. A CONSTANT is a non-empty finite sequence of digits 0, . . . , 9

```
VALUE_ASSIGNMENT ::= VARIABLE ":=" VARIABLE
                   | ARRAY    ":=" VARIABLE
                   | VAIABLE  ":=" ARRAY
                   | VARIABLE ":=" "(" VARIABLE "+" VARIABLE ")"
                   | VARIABLE ":=" "(" VARIABLE "-" VARIABLE ")"
```

Note: It is not allowed to use an identifier as a VARIABLE that has been used in an ARRAY before.

```
COMMANDBLOCK ::= COMMAND ";" COMMANDBLOCK
               | COMMAND
```

```
WHILE_LOOP ::= "while" CONDITION "do" COMMAND
```

```
CONDITION ::= "(" VARIABLE "#" "0" ")"
```

Note: It is not allowed to use an identifier as a VARIABLE that has been used in an ARRAY before.

```
ARRAY ::= VARIABLE "[" VARIABLE "]"
```

Note: Here it is not allowed to use an identifier as the first VARIABLE that has been used as a "simple" VARIABLE before. It is also forbidden to use an identifier as the second VARIABLE that has been used to identify an array before.

3.3 RAM

RAM is a language that describes a machine somewhat like a real assembler language. The difference is that you have an infinite amount of registers, and each register is of infinite size. There is no memory, and the code is not in the registers, but in a seperate space. The code is fix. The program cannot change its own code. The code is described using the RAM language.
Documented example RAM program:

```
1 R0 <- 0               R0 is initialized with 0
2 R1 <- 1               R1 is initialized with 1
3 R2 <- 1764            R2 is initialized with 1764
4 IF R2 = 0 GOTO 9      perform a loop until R2 is equal to 0
```

```
5 R2 <- R2 - R0          decrement R2 by R0
6 R0 <- R0 + R1          increment R0 by 1
7 R2 <- R2 - R0          decrement R2 by R0
8 GOTO 4                 goto beginning of loop
9 STOP                   this terminates the program
```

This program calculates the square root of 1764. The result is stored in R0. The algorithm works because all numbers raised to power of 2 can be listed by starting with 1 and adding 1+2, 2+3, 3+4, 4+5, 5+6, . . . This program decrements the given number by these numbers until 0 is reached. The result is the square root if it is an integer, or the next higher integer value. There are a few more commands in the RAM language. Here is a complete list of all available commands:

syntax	semantic
Rx<–Ry	copy value of register y in register x
Rx<–RRy	copy value of register with the number stored in Ry in register x
RRx<–Ry	copy value of register y in register with the number stored in Rx
Rx<–k	set the value of register x to the constant k
Rx<–Ry+Rz	add the values of register y and register z and store the result in register x
Rx<–Ry–Rz	calculate md(Ry, Rz) and store the result in register x
GOTO m	continue program execution in line m
IF Rx=0 GOTO m	if value of register x is equal to 0 continue program execution in line m
IF Rx>0 GOTO m	if value of register x is greater than 0 continue program execution in line m
STOP	terminate program execution

md(x, y) is called the modified difference of x and y and is calculated like the normal difference as long as x is greater than y. In other case the result is 0.

Chapter 4

Tools in the theosim package

Theosim consists of several programs. These are:

4.1 Theosim - the GUI

Theosim is an Integrated Development Environmend (IDE) to make work with the compilers and the simulator easier.

The user interface is straightforward. By doing the following steps, you should learn it alltogether.

1. Choose File/Open.
2. Type "adder.pli" and press ENTER.
3. Now type in the following program:

```
function adder(x);
begin
  adder := (x+x)
end
```

4. Choose Tasks/Pascalli->Pascallino.
5. You now see the compiled program in Pascallino.
6. Choose Tasks/Pascallino->RAM.
7. You now see the compiled program in RAM.
8. Choose Tasks/Ramsim.
9. (Read ramsim documentation).
10. Set Register 0 with value 40.
11. Press Run.
12. see that Register 0 has value 80 (=x+x).
13. Choose Buffer/* and continue your work.

When selecting an item in the Tasks menu, theosim calls an external program. The output is shown in a window. You can interrupt the external program by pressing the "Interrupt" button. When the external program is done, crashed or interrupted, you see its status above the Ok button.

When you ran a compiler, the resulting program is automatically loaded and popped into the front.

4.2 Plic - the Pascalli to Pascallino compiler

Plic is a tool which allows you to compile a Pascalli program into a Pascallino program.

You can call this compiler by using the following syntax:

```
plic <infile> <outfile>
```

At a first step the compiler will parse the program <infile> and build a parse-tree in order to compile it to Pascallino at a further step.

During the parsing the compiler will report all the errors found in the source one by one, which means that you are supposed to run the compiler, wait for the first error to be reported, correct it and run the compiler again searching for further errors.

If there are no more errors the compiler will report the message:

```
Compilation finished at <time>
```

otherwise it will report the error and the message:

```
Compilation exited abnormally at <time>
```

Here you have a list of all possible errors the plic compiler will return; please check the Pascalli syntax for more details.

4.2.1 Possible errors and their solution

"parse error"

the program to be compiled doesn't match the syntax of Pascallino

Example program:

```
function sum(a,b)
begin
  sum:=(a+b);
end
```

Reported Error:

```
At line 4: parse error
  end
   *^*
```

Solution: consult the syntax of Pascalli to correct the error; in the example you just have to add a ';' after the head of the function definition:

```
function sum(a,b);
begin
  sum:=(a+b)
end
```

"Function already declared!!!"

Pascalli doesn't allow "redefining" functions

Example program:

```
function sum(a,b);
begin
  sum:=(a+b)
end
```

```
function sum(a,b,c);
begin
  sum:=((a+b)+c)
end
```

Reported Error:

```
At line 6: Function already declared!!!
  function sum(a,b,c);
            *^*
```

Solution: rename the other function; in the example:

```
function sum(a,b);
begin
  sum:=(a+b)
end

function sum3(a,b,c);
begin
  sum3:=((a+b)+c)
end
```

"Function has no return-value!!!"

a defined function has no return-value
Example program:

```
function sum(a);
begin
  b:=a
end
```

Reported Error:

```
At line 4: Function has no return-value!!!
  end
   *^*
```

Solution: assign a return-value to the function-identifier

"Function-identifier not allowed as a formal parameter!!!"

it is not allowed in Pascalli to use the function-identifier as a formal parameter
Example program:

```
function sum(sum);
begin
  sum:=sum
end
```

Reported Error:

```
At line 1: Function-identifier not allowed as formal parameter!!!
  function sum(sum);
                *^*
```

Solution: rename the formal parameter

"Function <function-name> used but not declared!!!"

it is not allowed in Pascalli to call a function which has not been "defined"

Example program:

```
function prod(a,b);
begin
  while(a>0) do
  begin
    c:=sum(c,b);
    a:=(a-1)
  end
  prod:=c
end
```

Reported Error:

```
Function sum used but not declared!!!
```

Solution: "define" a function before calling it; in the example:

```
function prod(a,b);
begin
  while(a>0) do
  begin
    c:=sum(c,b);
    a:=(a-1)
  end;
  prod:=c
end

function sum(a,b);
begin
  sum:=(a+b)
end
```

Comment: Note that the line-number of the error is not reported

"Number of parameters in call to function <function-name> does not match declaration!!!"

it is an error in Pascalli to call a function with a bad number of formal parameters

Example program:

```
function prod(a,b);
begin
  while(a>0) do
  begin
    c:=sum(c,b,0);
    a:=(a-1)
  end;
  prod:=c
end

function sum(a,b);
begin
  sum:=(a+b)
end
```

Reported Error:

```
Number of parameters in call to function <function-name> does not
match delcaration!!!
```

Solution: call the function with the correct number of formal parameters

```
function prod(a,b);
begin
  while(a>0) do
  begin
    c:=sum(c,b);
    a:=(a-1)
  end
  prod:=c
end

function sum(a,b);
begin
  sum:=(a+b)
end
```

Comment: Note that the line-number of the error is not reported.

"Formal parameter used before!!!"

one of the formal parameters in the parameter list is duplicated
Example program:

```
function sum(a,a);
begin
  sum:=(a+a)
end
```

Reported Error:

```
At line 1: Formal parameter used before!!!
  function sum(a,a)
                *^*
```

Solution: eliminate the duplicated formal parameters; in the example:

```
function sum(a)
begin
  sum:=(a+a);
end
```

Comment: note that there is obviously no sense in duplicating formal parameters

"Variable used as array before!!!"

A variable has been "defined" as an array before.
Example program:

```
function sum(a);
begin
  b[a]:=a;
  sum:=(a+b)
end
```

Reported Error:

```
At line 4: Variable in expression used as array before!!!
  sum:=(a+b);
          *^*
```

Solution: rename the variable using a new identifier; in the example:

```
function sum(a);
begin
  b[a]:=a;
  sum:=(a+c)
end
```

"Variable used as function before!!!"

an variable has been "defined" as a function before

Example program:

```
function sum(a);
begin
  b:=double(a);
  sum:=(a+double)
end
```

Reported Error:

```
At line 4: Variable used as a function before!!!
  sum:=(a+double)
               *^*
```

Solution: rename the variable using a new identifier

"Counter-variable used at the left side of an assignment!!!"

the counter-variable is assigned a value within the commandblock of the for loop, which is not allowed in Pascalli

Example program:

```
function prod(a,b);
begin
  for counter:=1 to a do
  begin
    c:=sum(c,b);
    counter:=a;
  end
  prod:=c;
end
```

Reported Error:

```
At line 7: Counter-Variable used at the left side of an assignment!!!
  end
   *^*
```

Solution: Don't use the counter-identifier at the left side of an assignment

"Variable in index used as array before!"

the variable b used in an array of the form a[b] has been "defined" as an array before
Example program:

```
function sum(a);
begin
  b[a]:=a;
  c[b]:=a;
  sum:=(a+a)
end
```

Reported Error:

```
At line 4: Variable in index used as array before!
  c[b]:=a
    *^*
```

Solution: do not use variable-identifiers as the index of an array

"Array used as variable before!"

an array has been "defined" as a variable before
Example program:

```
function sum(a);
begin
  b:=a;
  b[a]:=a
  sum:=(a+a)
end
```

Reported Error:

```
At line 4: Array used as variable before!
  b[a]:=a
    *^*
```

Solution: rename the array using a new identifier; in the example:

```
function sum(a);
begin
  b:=a;
  c[a]:=a
  sum:=(a+a)
end
```

"Array used as a function before!"

an array has been "defined" as a function before
Example program:

```
function sum(a);
begin
  b:=c(a);
  b[a]:=a;
  sum:=(b[a]+c)
end
```

Reported Error:

```
At line 4: Array used as a variable before!!!
  b[a]:=a;
    *^*
```

Solution: rename the array

"Function used as variable before!!!"

a function-identifier has been used as a variable-identifier before

Example program:

```
function prod(a,b);
begin
  while(a>0) do
  begin
    sum:=sum(sum,b);
    a:=(a-1)
  end
  prod:=sum;
end
```

Reported error:

```
At line 5: Function used as a variable before!!!
  sum:=sum(sum,b);
             *^*
```

Solution: rename the function

"Function used as array before!!!"

a function-identifier has been used as an array-identifier before

Example program:

```
function prod(a,b);
begin
  while(a>0) do
  begin
    sum:=sum(sum,b);
    a:=(a-1)
  end
  prod:=sum;
end
```

Reported error:

```
At line 6: Function used as an array before!!!
  c:=sum(c,b);
         *^*
```

Solution: rename the function

"Function-identifier used as an assignable expression!!!"

the function-identifier is not an assignable expression, its only purpose is to store the return-value of the program

Example program:

```
function sum(a);
begin
  b:=sum;
  sum:=(a+b)
end
```

Reported Error:

```
At line 3: function-identifier used at right side of an assignment!
  b:=sum;
      *^*
```

Solution: Don't use the function-identifier as an assignable expression

4.3 Ploc - the Pascallino to RAM compiler

Ploc is a tool which allows you to compile a Pascallino program into a RAM program.

You can call this compiler by using the following syntax:

```
ploc <infile> <outfile>
```

At a first step the compiler will parse the program <infile> and build a parse-tree in order to compile it to RAM at a further step.

During the parsing the compiler will report all the errors found in the source one by one, which means that you are expected to run the compiler, wait for the first error to be reported, correct it and run the compiler again searching for further errors.

If there are no further errors the compiler will report the message:

```
Compilation finished at <time>
```

otherwise it will report the error and the message:

```
Compilation exited abnormally at <time>
```

Here you have a list of all the possible errors the ploc compiler will return; please check the Pascallino syntax for more details.

4.3.1 Possible errors and their solution

"parse error"

the program to be compiled doesn't match the syntax of Pascallino.

Example program:

```
function sum(a,b)
begin
  sum:=(a+b)
end
```

Reported Error:

```
At line 4: parse error
  end
   *^*
```

Solution: consult the syntax of Pascallino to correct the error; in the example you only have to add a ';' after the head of the function definition:

```
function sum(a,b);
begin
  sum:=(a+b)
end
```

"Formal parameter used before!!!"

one of the formal parameters in the parameter list is duplicated

Example program:

```
function sum(a,a)
begin
  sum:=(a+a)
end
```

Reported Error:

```
At line 1: Formal parameter used before!!!
  function sum(a,a)
                *^*
```

Solution: eliminate the duplicated formal parameters; in the example:

```
function sum(a)
begin
  sum:=(a+a);
end
```

Comment: there is obviously no sense in duplicating formal parameters

"Variable used as array before!!!"

a variable has been "defined" as an array before

Example program:

```
function sum(a)
begin
  b[a]:=a;
  sum:=(a+b)
end
```

Reported Error:

```
At line 4: Variable in expression used as array before!!!
  sum:=(a+b);
          *^*
```

Solution: rename the variable using a new identifier; in the example:

```
function sum(a);
begin
  b[a]:=a;
  sum:=(a+c)
end
```

"Function-identifier used as an assignable expression!!!"

the function-identifier is not an assignable expression, its only purpose is to store the return-value of the program

Example program:

```
function sum(a);
begin
  b:=sum;
  sum:=(a+b)
end
```

Reported error:

```
At line 3: function-identifier used at right side of an assignment!
  b:=sum;
      *^*
```

Solution: Don't use the function-identifier as an assignable expression

"Variable in index used as array before!"

the variable b used in an array of the form a[b] has been "defined" as an array before

Example program:

```
function sum(a);
begin
  b[a]:=a;
  c[b]:=a;
  sum:=(a+a)
end
```

Reported Error:

```
At line 4: Variable in index used as array before!
  c[b]:=a
    *^*
```

Solution: Don't use array-identifiers as the index of an array

"Array used as variable before!"

an array has been "defined" as a variable before

Example program:

```
function sum(a);
begin
  b:=a;
  b[a]:=a;
  sum:=(a+a)
end
```

Reported Error:

```
At line 4: Array used as variable before!
  b[a]:=a;
    *^*
```

Solution: rename the array using a new identifier; in the example:

```
function sum(a);
begin
  b:=a;
  c[a]:=a
  sum:=(a+a)
end
```

"No return value!"

the function has no return value

Example program:

```
function sum(a);
begin
  b:=a
end
```

Reported Error:

```
At line 4: No return value!
```

Solution: assign a return-value to the function-identifier

4.4 Ramsim

Ramsim can simulate a RAM program. It gives you a nice graphical user interface (GUI) which you can use like a normal debugger. You can inspect registers, manipulate them, and run the program either in steps or until a certain point or as a whole.

The user interface is straightforward. By doing the following steps, you should learn it alltogether.

1. Choose File/Open and load a .ram file. You could also have passed it on the commandline as is done if ramsim was run from theosim.
2. Now you probably want to set register 0 to some value, since register 0 is usually the input. To do this, press the SetReg button.
3. You could run the complete program by pressing the Run button.
4. if running takes too long you could press the interrupt button and see what's happening. Pressing Run again does continue, not restart.
5. If you press the Step button, only one command is executed.
6. If you press the "Run slowly" button, you can watch the program running.
7. The Reset button is like reopening the file.
8. You can also select a line of code and press the Until button. The machine will process until it reaches your point.

If you write a pascalli program, compile it to pascallino and compile it to .ram afterwards, you'll get an enourmously inefficient program. It might happen that it "never" terminates. This is due to the theoretical and impractical way the compilation is done.

The compilation is easy to understand and easy to prove. But it really produces terribly inefficient code.

Chapter 5

Internals

If you are interested, you can now learn about the internals of our sources!

5.1 Theosim

Theosim uses the libQT library for its graphical user environment. See *www.troll.no*. Theosim just stacks some QMultiLineEdit into a QWidgetStack object and supplies them in the menubar. The call to external programs is done by a special handcoded widget. Stdout and Stderr are shown in a window and once the external program terminates, its termination status is checked and displayed. The code should be straightforward and if you are interested in further details it should be best to consult the sourcecode.

5.2 Plic

5.2.1 How the parser works

The plic parser is implemented using the GNU-tools lex and yacc.

The file pascalli_lexer.ll contains the specification of the valid tokens lex will use in order to build a lexer.

The file pascalli_parser.yy contains the specification of the grammar as well as the code to be executed once a rule is reduced yacc will need to create a bottom-up parser.

Note that in spite of the fact that yacc handles left-recursive rules more efficiently than right-recursive ones, all the recursive rules are implementes right-recursive. The reason for this is that we want to parse all the constructs from "left to right". For example, if we want to parse an argument-list (a,b,c) it would be rather reasonable to parse first the "a", then the "b" and then the "c". For this reason the rule to parse an argument-list is implemented as follows:

```
arglist: arglist ',' IDENTIFIER
                   {
                       ...
                   }
```

The way the parser works is quite simple to explain:

1. the lexer reads the input-file and feeds yacc with all the found TOKENS
2. the parser "shifts" all these TOKENS until it is able to "reduce" a rule; then it will execute the associated code
3. the steps 1) and 2) are repeated until the parser is able to reduce the rule with the top-catgeory on the left side

Note: Consult the files pascalli_lexer.ll and pascalli_parser.yy for further details

5.2.2 Construction of the parse-tree

The file pascalli_parser.yy includes the header-file frontend.h in order to create a parse-tree.

In this file the object-hierarchy of the different syntactical constructs of Pascalli is defined.

The way the parser builds up a tree is rather simple to explain: Once the parser is able to reduce a rule it creates a new object corresponding to the category on the left side of the rule and involving the categories or TOKENS of the right side of the rule in the construction (see the implementation of the constructors in frontend.h for more details). This process is iterated until the parser is able to reduce the rule with the category "program" at the left side and create a xProgram which will serve as argument in the call of xTranslator (the compiler) at a further step.

5.2.3 Symboltables & Identifiers

The parser uses five symboltables:

1. Variable_Symboltable (xIdentifierSet): the symboltable storing the variables used withing the function being parsed
2. Array_Symboltable (xIdentifierSet): the symboltable storing the arrays used within the function being parsed
3. Declared_Functions_Symboltable (xIdentifierSet): the symboltable storing the functions declared in the pascalliprogram
4. Used_Functions_Symboltable (xIdentifierSet): the symboltable containing the functions called in the pascalliprogram
5. Local_Functions_Symboltable (xIdentifierSet): the symboltable containing the function-identifiers used within the function being parsed

The explanation for the use of all these symboltables is quite easy: The Variable_Symboltable, the Array_Symboltable and the Local_Functions_Symboltable allow the parser to find out if a variable has been "defined" as array before or if a variable has been used as a function before etc.

Note that these symboltables are destructed and initialized once the function has been parsed.

The Declared_Functions_Symboltable and the Used_Functions_Symboltable have the purpose of checking if every called function has been delcared.

The parser also uses a global pointer Function_Name (xIdentifier *); this pointer has the purpose of storing the function-identifier. This turns out to be very useful as the parser can easely verify if a variable-identifier or an array-identifier is equal to the function-identifier, which is very important for error-reporting.

5.2.4 Error reporting

There are two functions defined in pascallino_lexer.ll for the purpose of error reporting:

1. void yyerror(char *s) : reports the error with the corresponding line and position
2. void yyerror2(char *s): reports the error without line and position

It is worth to have a look at the way errors are reported by the parser. For this, we will note that there are basically two types of errors:

1. context-free errors (parse errors)
2. context-sensitive errors

Context-free errors are always reported using the function void yyerror(char* s); the location of this type of error is always exact, which means that the parser reports an error at the end of the TOKEN that does not match the syntax of Pascallino.

Context-sensitive errors are reported mostly using the function void yyerror (char *s); however the location of the error is rather unprecise. The reason for this is that the parser reports an error at the end of the "right side" of the rule it is reducing and the code it is executing.

Other context-sensitive errors are reported with the function void yyerror2 (char *s); for this type of errors it would make no sense to report the line in which they where produced.

Aditionally the parser defines a global boolean variable Value_Assigned (TBool) and initializes it with "FALSE". Once the parser finds an assignment in which the function_identifier appears on the left side, he sets Value_Assigned "TRUE"; so the parser is always able to check if a function has a return-value.

The parser also defines a function TBool used_but_not_declared() which checks if all the called functions have been defined by searching all the functions of the Used_Functions_Symboltable in the Declared_Functions_Symboltable; it returns "TRUE" if one of the functions called is not in the Declared_Functions_Symboltable, otherwise it returns "FALSE".

5.2.5 Deallocation of the parse-tree

For the purpose of deallocating the parse-tree six global lists are created:

1. Free_IdentifierLists (list containing the lists of xIdentifiers to deallocate)
2. Free_ExpressionLists (list containing the lists of xExpressions to deallocate)
3. Free_FunctionLists (list containing the lists of xFunctions to deallocate)
4. Free_CommandLists (list containing the lists of xCommands to deallocate)
5. Free_Identifiers (list containing the xIdentifiers to deallocate)
6. Free_Nodes (list containing the xNodes to deallocate, that means: the parse-tree)

The deallocation takes place in the function void free_parsetree(), which is called at the end of the program. This function deallocates all the above mentioned lists.

But it is worth to have a look at the mechanism how the parse-tree is stored in the Free_Nodes-list in order to be deallocated at a further step: when the parser finds a new object it is inserted into the Free_Nodes- list. However, when the parser creates a new object, that means, calls the constructor of this object with the subordered (in the object-hierarchy) objects it eliminates these subordered objects from the Free_Nodes-list.

We could call this way of deallocating the parse-tree as a shift/reduce- deallocation, meaning that the objects created are "shifted" on the Free_Nodes- list but also removed and substituted by the object corresponding to the father-category (in the object-hierarchy) if the parser is reducing the corresponding rule with this "father-category" on the left side.

Thus, the deallocation works top-down; however, the advantage of this shift/reduce-deallocation is that the parser will also be able to deallocate top-down non-complete parse-trees corresponding to syntactically non-correct Pascalli-programs.

5.2.6 Call of xTranslator

Once the parser has reduced the rule with the "program"-category at the left side it will call the xTranslator with the TopNode (xNode *) of the parse-tree an the <outfile> (xString) as arguments.

5.3 Ploc

5.3.1 How the parser works

The ploc parser is implemented using the GNU-tools lex and yacc.

The file pascallino_lexer.ll contains the specification of the valid tokens lex will use in order to build a lexer.

The file pascallino_parser.yy contains the specification of the grammar as well as the code to be executed once a rule is reduced yacc will need to create a bottom-up parser.

Note that the grammar provided to yacc is nearly the one specified in BNF on chapter [...].

Note also that in spite of the fact that yacc handles left-recursive rules more efficiently than right-recursive ones, all the recursive rules are implemented right-recursive. The reason for this is that we want to parse all the constructs from "left to right". For example, if we want to parse an argument-list (a,b,c) it would be rather reasonable to parse first the "a", then the "b" and then the "c". For this reason the rule to parse an argument-list is implemented as follows:

```
arglist: arglist ',' IDENTIFIER
                 {
                    ...
                 }
```

The way the parser works is quite simple to explain:

1. the lexer reads the input-file and feeds yacc with all the found TOKENS
2. the parser "shifts" all these TOKENS until he is able to "reduce" a rule; then he executes the associated code
3. the steps 1) and 2) are repeated until the parser is able to reduce the rule with the top-catgeory on the left side

Note: consult the files pascallino_lexer.ll and pascallino_parser.yy for further details

5.3.2 Construction of the parse-tree

The file pascallino_parser.yy includes the header-file frontend.h in order to create a parse-tree.

In this file the object-hierarchy of the different syntactical constructs of Pascallino is defined.

The way the parser builds up a tree is rather simple to explain: Once the parser is able to reduce a rule he creates a new object corresponding to the category on the left side of the rule and involving the categories or TOKENS of the right side of the rule in the construction (see the implementation of the constructors in frontend.h for more details). This process is iterated until the parser is able to reduce the rule with the category "program" at the left side and create a xProgram which will serve as argument in the call of xPascallinoVisitor (the compiler) at a further step.

5.3.3 Symboltables & Identifiers

The parser includes the header xset.h in order to create a xVariableSet-class (see the file pascallino_parser.yy) for more details). The parser will use two symboltables:

1. Variable_Symboltable (xVariableSet): the symboltable storing the used variables
2. Array_Symboltable (xVariableSet) : the symboltable storing the used arrays

This proceeding turns out to be of great importance when error-reporting is concerned as the parser is be able to check if a variable has been "defined" as an array before and viceversa.

Thus, always when the parser parses an identifier (variable or array) he creates the corresponding object (xVariable or xArray) and searches for it in one of the symboltables (Variable_Symboltable or Array_Symboltable). If the parsed identifier is new, which means that it is neither in one of the symboltables, the identifier is inserted into the corresponding symboltable and a xMemoryObject is created, otherwise no xMemoryObject is created.

The parser also uses a global pointer Function_Name (xVariable *); this pointer has the purpose of storing the function-identifier. This turns out to be very useful as the parser can easely verify if a variable-identifier or an array-identifier is equal to the function- identifier, which is very important for error-reporting.

Aditionally the parser defines a global boolean variable Value_Assigned (TBool) and initializes it with "FALSE". Once the parser finds an assignment in which the function_identifier appears on the left side, he sets Value_Assigned "TRUE". By this the parser is able to check if the function has a return-value.

5.3.4 Error reporting

There are two functions defined in pascallino_lexer.ll for the purpose of error reporting:

1. void yyerror(char *s) : reports the error with the corresponding line and position
2. void yyerror2(char *s): reports the error without line and position

It is worth to have a look at the way errors are reported by the parser. For this, we will note that there are basically two types of errors:

1. context-free errors (parse errors)
2. context-sensitive errors

Context-free errors are always reported using the function void yyerror(char *s); the location of this type of error is always exact, which means that the parser reports an error at the end of the TOKEN that does not match the syntax of Pascallino.

Context-sensitive errors are reported mostly using the function void yyerror (char *s); however the location of the error is rather unprecise. The reason for this is that the parser reports an error at the end of the "right side" of the rule he is reducing and which code he is executing.

Other context-sensitive errors are reported with the function void yyerror2 (char *s); for this type of errors it would make no sense to report the line in which they were produced.

5.3.5 Deallocation of the parse-tree

For the purpose of deallocation of the parse-tree two global xLists are created:

1. Free_List : a xList containing pointers to objects of the Pascallino-object-hierarchy to be deallocated
2. Free_List2: a xList containing pointers to xMemoryObjects to be deallocated

Always when the parser creates an object (xVariable, xArray, ...) it is added to the Free_List and when he creates a xMemoryObject it is added to the Free_List2. At the end all the objects of the Free_List and the Free_List2 are deallocated iteratively.

5.3.6 Call of xPascallinoVisitor

Several global variables are created for the purpose of calling the compiler (xPascallinoVisitor):

1. Number_Functions (TIndex)
2. Number_Parameters (TIndex)

3. Number_Arrays (TIndex)
4. Number_Variables (TIndex)

Note that the Number_Functions is initialized with "1" as a Pascallino- program consists of exactly one function; for this reason Number_Functions is not changed by the parser. On the other hand, Number_Parameters, Number_Arrays, Number_Variables are increased always the parser finds a corresponding object (a formal parameter, an array or a variable).

5.4 Ramsim

Ramsim uses the libQT library for its graphical user environment. See *www.troll.no*. The window is divided into two subwindows. Custom widgets were created to show RAM code and the register database. The execution of RAM code is delegated to the common classes. The code should be straight forward and if you are interested in further details it should be best to consult the sourcecode.

5.5 common classes

In many classes, there are a few operators overloaded. Some of them have almost the same meaning within every class:

operator˜ returns the length/size.

operator! returns the current position/pointer.

5.5.1 auto.h (memory allocation)

There are two classes. xAllocator allocates memory for one object and xArrayAllocator for an array of given type. You can specify an initial length for the allocated array. If you have sepcified the initial lenght too small no problem, there is operator() wich resizes the array length. With operator[] you have access to each element in the array. Using these allocator classes, you don't have to care about deallocating memory. It is done automatically when destroying the instance of an allocator class.

5.5.2 base.h (basic type definitions)

There are definitions for ON, OFF, TRUE, FALSE, PI, EULER . . . and some types like TPosition, TSize . . . Here is not much to say. Look in the file and see for yourself.

5.5.3 bcode.h (byte code definitions)

Some defines for easier understanding byte code. Each RAM command is assigned a unique number.

RAM code	byte code number
Rx<–Ry	0
Rx<–RRy	1
RRx<–Ry	2
Rx<–2	3
Rx<–Ry+Rz	4
Rx<–Ry–Rz	5
GOTO m	6
IF Rx=0 GOTO m	7
IF Rx>0 GOTO m	8
STOP	9

5.5.4 compram.h (RAM compiler)

Here is a function that compiles RAM code into byte code (for internal use only). There may occur an error while compiling RAM code. In this case an exception is thrown (see 5.5.7 how exceptions work). CompileRAM2BC uses regular expressions with xStrings for compiling (see 5.5.16 and 5.5.18).

5.5.5 ctbase.h (basic container definitions)

xContainer is a base class used by ctlist.h (see 5.5.6). There also is an exception class xContainerException.

5.5.6 ctlist.h (some list classes)

xList is a simple list that uses xArrayAllocator to store its elements. Access to each element is done by using operator[]. You may insert/delete an element with operator+= or operator−=. You don't have to care about the length of the list. It is automatically incremented when using operator+=. If you store elements that must be deallocated when destroying the list, you have to overload the Kill method. Example:

```
{
  xList<int> list;
  list += 1;
}
```

In this case all memory that has been reserved is deallocated after this block. You don't have to overload anything.

```
{
  xList<int *> list;
  list += new int;
}
```

In this case you have to tell xList to deallocate each element in the list. This can be done by overloading the Kill method like this:

```
void MyList::Kill(int * &object)
{
  delete object;
}
```

The Create method follows the same rules.

5.5.7 exbase.h (basic exception definitions)

xBaseException is the base class for all exceptions used in this project. Here is an example:

```
try {
  ...
}
catch (xBaseException &ex) {
  char buf[200];
  cout << ex.GetDescription(buf) << endl;
}
catch (...) {
  cout << "unknown exception" << endl;
  throw;
}
```

Somewhere in the try block an exception is thrown. If it is our own exception with base class xBaseException it is caught in the first catch block. You can get some description of the error by using the method GetDescription, but you have to provide some space where the description can be stored.

If you want to write your own exception, the only thing you have to do is to overload the GetText method and the constructor. GetText should reurn a short description for the error code stored in Error. (see exgen.h how it works)

Have a look at the macros for throwing exceptions. This may seriously shorten your throw statemend.

5.5.8 exgen.h (some common exceptions)

Here are some predefined exceptions (general and memory).

5.5.9 numeric.h (some numeric functions/macros)

Here are some more or less useful functions/macros around arithmetics.

5.5.10 prgbase.h (basic program definitions)

xProgram is a class is a xList with two aditional methods Read and Write to read and store date from/in a stream. It is used to manage any source code. You have easy access to any line of your source and the ability to read/write your source from/to a file.

5.5.11 prgbc.h (byte code program)

Each command is coded in a structure (xByteCode) which contains a number specifying the command type (see bcode.h), and three strings (xString) representing parameters. Since some commnad types haven't got three parameters, it is possible that e.g. only one string contains a parameter and the others are empty. Everytime the first available string is used. It is not possible that the first string is empty and the second string contains a parameter. Here is a complete list of RAM - byte code relations:

RAM code	byte code number	Parameter
R1<–R2	0	”1”, ”2”, ””
R1<–RR2	1	”1”, ”2”, ””
RR1<–R2	2	”1”, ”2”, ””
R1<–2	3	”1”, ”2”, ””
R1<–R2+R3	4	”1”, ”2”, ”3”
R1<–R2–R3	5	”1”, ”2”, ”3”
GOTO 1	6	”1”, ””, ””
IF R1=0 GOTO 2	7	”1”, ”2”, ””
IF R1>0 GOTO 2	8	”1”, ”2”, ””
STOP	9	””, ””, ””

Since each command is an element in the list, you have easy access by using operator[]. When reading/writing from/to a stream, you may use the operator>> and operator<<.

5.5.12 prgcplx.h (Complex program definitions)

Here is a container for complex programs like Pascalli or Pascallino. Since each line of a Pascalli/Pascallino program is an element in the list, you have easy access to lines by using operator[]. When reading/writing from/to a stream, you may use the operator>> and operator<<.

5.5.13 prgram.h (RAM program definitions)

Each line of a RAM program is assigned a line number starting with 1. The order of line numbers must be correct and without holes. That is why xRAMProgram does a line checking when reading data from a stream and throws an exception in case of incorrectness. As line numbers can be reconstructed from indexes, they are not stored in the list. When writing the program to a stream, the line numbers are automatically generated.

Since each command is an element in the list, you have easy access by using operator[]. When reading/writing from/to a stream, you may use the operator>> and operator<<.

If you want to generate a RAM program, you should use the functions in xRAMCode. These functions generate the right syntax for each command with the given parameter(s).

5.5.14 ram.h (ram interpreter)

The class xRAM loads and interprets RAM programs. RAM code will not be interpreted directly. At first it will be compiled into a byte code for easier interpretation. (see section 5.5.3 for byte code specification) Byte code is for internal use only. You need not take care of it, if you are only using the RAM interpreter. The class "xRAM" has a function "Load" to load RAM programs. Whithin this function the RAM program passed as parameter will automatically be compiled into byte code. To interpret the program you may use one of the three execute functions.

- Execute() interprets the program until the "STOP" command appears.
- Execute(TCount) interprets the secified number of lines.
- ExecuteTo(TIndex) interprets until the specified line is reached.

The command register stores the line number of the next command to be interpreted and can be read with the function "ReadCommandRegister()". All other registers can be read with the function "ReadRegister(xString)", giving the number of the desired register as parameter.

5.5.15 register.h (register simulator)

Since numbers stored in registers have no upper limit, dynamically growing strings ("xString") are used. The amount of register also has no upper limit. That's why a dynamically growing list is necessary ("xList") too. There is no need to call memory allocation functions, when you want to use a new register. If the first access is a read access, the register will have the zero value. Each register is named by a unique number (Name in xNamedRegister). All operators needed for interpreting byte code are overloaded, that are operator=, operator+ and operator–. Name and value of a xNamedRegister can be read using the GetName and GetValue methods.

xRegisterManager is responsible for creating and managing all the registers you are using. To save memory the register number is not equal to the index in the register list, so you can access register no. 500000 and only memory for one register will be reserved. In order to have easy access to registers, the operator[] is overloaded. Example:

```
xRegisterManager manager;
manager[3000] = 2;              //register no. 3000 has now the value 2
manager[8000] = "100200300400"; //strings may be used for very long numbers
manager[7411] = manager[666];   //register no. 666 and no. 7411 have zero value
```

5.5.16 xset.h (set definitions)

A set is a list of elements where each element is unique. operator+= inserts an element if not already in the list. operator–= deletes an element. operator() returns the index of given element. Contains checks wheater an element is in the list or not.

5.5.17 xstring.h (string manipulating)

The class xString has everything to work with strings. xStrings are dynamically growing, you can use almost all operators like $<$, $>$, $=$, $+$ ...and they have a lot of methods for manipulation. There are always two methods for one function. One of them looks like name(...) and the other one like nameThis(...). Methods ending with "This" change the data of their object, while the other ones return a new xString containing the manipulated data. All methods are selfexpnaining, see for yourself.

5.5.18 xstrio.h (xString io operators)

To read/write xStrings from/to a stream, it is necessary to overload the operator>> and operator<<. Writing is done one by one. Exactely the string data is written, nothing more or less. Reading is done some different. Always the whole line will be read and line feed and return characters are stripped.

5.5.19 xstrre.h (regular expressions with xString)

Regular expression matching is needed while compiling RAM code. xRegexString has not the full perl5 functionality, but it is satisfying for use in this project. After having asigned a regular expression to an instance of xRegexString, you have to call the Parse method before matching strings. Simply use Match or MatchAt to see if the given string matches the regular expression. GetMatch returns the string of the last match.