A Compiler for the **Paladim** Language

Group Project for the Compiler Course

Block 2, Winter 2013/14

Contents

1 Introduction

This is the description of the project for the Compilers Course (Oversættere) in block 2 of 2013-2014. The project should be solved in groups of (up to) three students, and will be evaluated as passed or failed (without a mark). You have to pass the project task to get access to the final exam of the course.

The task is to complete the implementation of a compiler for the **Paladim** language[1], described in detail in a later section. The **Paladim** programming language is a pascal-like imperative language with functions and procedures, which uses multidimensional regular arrays.

The task is **available from Tuesday, 19 Nov. 2013**. A **first part** of the solution, from now on called the ***milestone***, is **due on Friday, 6 Dec. 2013**. The **final solution** must be **handed in by Friday, 20 Dec. 2013, 2:00pm**.

[1] **Paladim** stands for "Pascal and large array dimensions, introducing main".

2 The Project Task

2.1 Task Summary

A partial implementation of a **Paladim** compiler has been prepared as a hand-out for you. Your task will be to add specific language features and parts of their implementation, to produce a compiler for the full language. Some small test programs are also provided, but you should add more of them to test your implementation.

In brief, you need to implement the following features in the assignment:

- A new parser for **Paladim**, using the tool `mosmlyac`.
  This subtask *should be solved and described at the milestone.*
- Integer multiplication and division, and boolean operators (`or`, `not`);
- Simple type inference and type checking for special functions `read` and `new`;
- code generation for indexing into arrays;
- call-by-value-result semantics for procedure arguments.

The project tasks are described in more detail in Section 3.3, after the language description.

2.2 Solving the Tasks and Submitting the Solution

The project should be solved in groups of (up to) three students. Solutions should be handed in by uploading code and a written report to the Absalon course pages.

Midway through the project, a *milestone submission* should be delivered. The milestone consists of a solution to one of the subtasks, and a short status report on the other tasks. Its purpose is to enable early feedback from teaching assistants about report writing style, testing methodology, and implementation work.

The *final solution* consists of the compiler code and a project report, which describes your solutions to all subtasks (and will thus include material submitted earlier for the milestone). The final report will be assessed as passed or failed, without a mark.

Use the group submission function in Absalon for both hand-ins, and indicate the names of all group members in the reports. Approval of this task is a prerequisite for participation in the final exam (in addition to approval of at least four of the five weekly assignments), and *cannot be resubmitted.*

2.2.1 Important Dates

Tuesday, 19 Nov. 2013: The **task is available** on Absalon
Friday, 6 Dec. 2013: A **milestone submission** should be handed in.
Friday, 20 Dec. 2013, 2:00pm: Final solution must be uploaded to Absalon.

2.2.2 Milestone Submission

The milestone submission should contain a *complete solution to the first subtask*, i.e. a parser for **Paladim** generated using the *mosmlyac* parser generator, including test programs. The parser should be able to parse the new kinds of expressions introduced by task 2. Furthermore, you should submit a written status report (2-5 pages) describing the implementation of the parser and how you have tested it, as well as your ideas and partial work on the other subtasks.

You should hand in two files by uploading them to Absalon:

1. A `zip` or `tar.gz` archive containing the *parser grammar file* (input file for mosmlyac), the rest of the compiler source code (which should now use the new generated parser), and the tests you have used. Please use the same directory structure as in the code which was handed out.

2. A written *status report* describing the parser implementation, its integration in the compiler and how it was tested, and ideas and status for the other subtasks. This report may be brief (2-5 pages), but should contain all necessary technical information.

Please use the group submission function in Absalon (i.e. only one submission per group), and write the names of all group members on the title page of the report.

2.2.3 Final Submission

The final solution for the project consists of a *complete implementation of all subtasks*, the *test data* you have used to test them, and a *technical report* describing your implementation and tests. Please upload to Absalon the following two files:

1. A `zip` or `tar.gz` archive containing the full implementation of the compiler and tests. Please use the same directory structure as the code which was handed out, and make sure that the provided compilation script `compile.(bat|sh)` works and produces your compiler as `BIN/Paladim`.

2. A written *report* describing and evaluating your work and the main design decisions you took. The report should not exceed 16 pages of text, and must describe your work in sufficient technical detail (see below).

Again, please use the group submission function in Absalon (i.e. only one submission per group), and write the names of all group members on the title page of the report. Your solution should demonstrate competence in the entire curriculum, understanding of all compiler phases and the ability to thoroughly document your solution. Partial solutions will be considered if they are convincing and well-documented.

Writing the Report: The technical report about your implementations shall include justification of the changes that were made in every compiler stage, i.e., scanner, parser, interpreter, type checker, code generator.

You should not include the whole compiler code in the content of your report, but you *must* include the parts that were either added, i.e., new code, or substantially

changed. Add them as Figures, or code listings in Annexes or the like. If you refer to implementation code, this must also be included in the report document.

The report shall describe how your modifications were tested, describing how test programs were selected, and their expected behaviour and/or output. If any of your tests do not succeed, possible reasons should be given.

Known shortcomings in type checking and compilation must be described, and, whenever possible, you need to make suggestions on how these might be corrected.

Your report should include all important information about your solution: all major design decisions should be presented and justified, and all deficiencies must be described. Ideally, we should not need to read your source code. However, be concise in your description and do not include irrelevant details, and keep the report below 16 pages. In the end, it is largely your decision what you think is important enough to include in the report, as long as the explicit requirements above are met.

Tools to Use: Please use the Moscow ML[2] compiler, `mosmlc`, to compile the code which is handed out and your modifications. For lexical and syntactical analysis, Moscow ML's tools `mosmllex` and `mosmlyac` will be used.
To run the programs compiled by the **Paladim** compiler, you should use the Mars simulator[3]. Mars simulates a MIPS processor, and is written in Java, so you need a Java Runtime Environment to use it.
We have tested our implementation and will evaluate yours with these tools.

Accepted Limitations: It is perfectly acceptable that the scanner, parser, type checker and code generator stop at the first error encountered. It can be assumed that the translated program is small, so that target addresses for jump and branch instructions always fit into constant fields of MIPS jump instructions (i.e. label addresses fit into 16 bit, for branch instructions). It is not necessary to free memory in the heap while running the program. You do not need to consider stack or heap overflow in your implementation.

3 Technical Description

3.1 Compiler Module Structure

A partial implementation of the **Paladim** compiler has been prepared as a starting point for your work, as well as a number of input programs and expected output for tests. This partial implementation can be found in the archive ***Paladim****.zip*. The archive contains four subfolders:

`SRC/`: contains the implementation of the compiler.

`DATA/`: contains test inputs and expected output files.

`DOC/`: contains documentation, e.g., this document and `mips-module.pdf`.

`BIN/`: will contain the compiler executable (binary) file.

[2]Moscow ML is available at `http://mosml.org` [1], the latest version is 2.10.
[3]Mars is available at `http://courses.missouristate.edu/kenvollmar/mars/` [2].

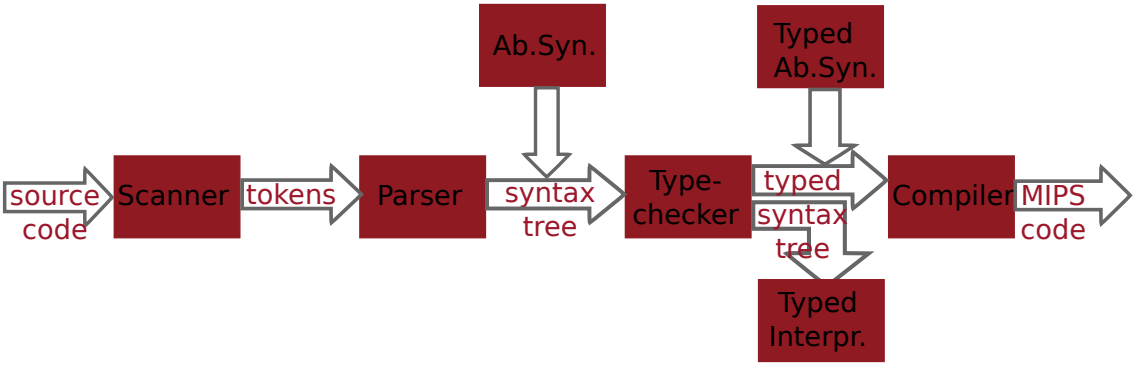

Figure 1: Compiler Modules of the **Paladim** Compiler

To complete the task you will need to modify files in the `SRC` folder, your tests should be added to the `DATA` folder.

Files `SRC/compile.sh` and `SRC/compile.bat` contain commands to rebuild all modules and the compiler under Linux and Windows, respectively. The handed-out code does not generate any compiler warnings. Try to keep your changed code free of warnings or errors as well. The bundle also contains a Makefile, but you do not need to understand or use it.

The **Paladim** compiler consists of a number of modules which form a processing chain, as depicted in Figure 1. The following modules are part of the compiler:

`Lexer.lex`: Definition of **Paladim** lexical tokens, used by `mosmllex` to generate the **Paladim** scanner.

`LL1Parser.sml`: A top-down parser for **Paladim** programs, producing the abstract syntax tree of the input program. This parser will be *replaced* in task 1.

`AbSyn.sml`: The abstract syntax data type for **Paladim** programs, along with some helper functions to format and print entities of a **Paladim** program.

`Type.sml`: Type-checker for **Paladim**, produces a type-checked abstract syntax.

`TpAbSyn.sml`: The abstract syntax data type for type-checked **Paladim** programs, along with helper functions for printing and for retrieving type information from **Paladim** programs.

`TpInterpret.sml`: An interpreter for type-checked **Paladim** programs (`TpAbSyn`).

`Compiler.sml`: Translator from **Paladim** to MIPS assembler. The translation is done directly from **Paladim** to MIPS, i.e., without passing through a lower-level intermediate representation of the code.

`Driver.sml`: The main program which defines the compilation chain (and file I/O).

There are two more modules in the compiler: a register allocator (`RegAlloc.sml`), and functions and types for MIPS instructions (`Mips.sml`). Both provide general

infrastructure and are not specific to the **Paladim** language implementation. It will not be necessary to modify these files for your project.

The main program `Driver.sml` is the driver of the compiler: it runs the scanner and parser, and type-checks the program. Depending on the given program options (`-ti` or `-c`), it then either interprets the program, or else compiles it to MIPS code. Processing stages after the type checker assume that the program is type-correct and use type information in code generation and interpretation.

3.2 The Paladim Language

Paladim is a Pascal-like imperative language with simple control structures and data types. A small example program is given here, which computes a Fibonacci number using an array of intermediate results (dynamic programming).

3.2.1 Lexical and Syntactic Structure

Paladim programs consist of a number of procedures and functions, the latter returning values while the former do not. There should always be a `main` procedure.

Functions and procedures consist of a top-level *block* with (optional) variable declarations followed by statements. Variables can have a base type (`char, int, bool`) or an array type (`array of ..`), which can also be nested.

The detailed syntax of **Paladim** is given by the grammar in Figure 3. Lexical entities are characterised in the following.

DATA/fibArray.pal

```
program fibonacciArray;

function fillFib(int n) : array of int
var r : array of int;
    i : int;
begin r    := new(n+1);
      r[0] := 0; r[1] := 1;
      i    := 1;
      while (i < n) do
         begin i  := i + 1;
               r[i] := r[i-1] + r[i-2];
         end;
      return r;
end;

procedure main()
var n : int;
    a : array of int;
begin n := read();
      if n < 1 then return;
      a := fillFib(n);
      write(a[n]); write('\n');
end; // main
```

Figure 2: **Paladim** Program to Compute Fibonacci Numbers

Identifiers in **Paladim** must begin with a letter (ranging from A to Z and from a to z), followed by any number of letters or digits or underscore. Some words (`if, then, function,...`) are reserved keywords and *cannot* be used as identifiers.

Numeric constants are positive whole decimal numbers, formed from digits 0 to 9. Numeric constants are limited to numbers representable as positive integers in Moscow ML. Negative number literals are unsupported, one must write `0-1` for `-1`).

Character literals are surrounded by single quotes (`'`). A character literal can be either a character with ASCII code between 32 and 126 *except* for characters `'`, `"` and `\`; or else, an *escape sequence*, consisting of character `\`, and one of the following characters: `a, b, f, n, r, t, v, ?, ', "`.

String literals consist of a sequence of characters surrounded by double quotes (`"`). Escape sequences as described above can be used in string literals.

Program structure:

Prog	→	**program** ID ; *FunDecs*	
FunDecs	→	*FunDecs FunDec*	
FunDecs	→	*FunDec*	
FunDec	→	**function** ID(*PDecl*) : *Type Block* ;	
FunDec	→	**procedure** ID (*PDecl*) *Block* ;	
Block	→	*DBlock SBlock*	
DBlock	→	**var** *Decs*	
DBlock	→	ε	
SBlock	→	**begin** *StmtSeq* ;**end**	
SBlock	→	*Stmt*	
StmtSeq	→	*StmtSeq* ; *Stmt*	
StmtSeq	→	*Stmt*	

Variable and Parameter Declarations, Types

PDecl	→	*Params*	
PDecl	→	ε	can be empty
Params	→	*Params* ; *Dec*	
Params	→	*Dec*	
Dec	→	ID : *Type*	
Decs	→	*Decs Dec* ;	
Decs	→	*Dec* ;	cannot be empty
Type	→	**int**	
Type	→	**char**	
Type	→	**bool**	
Type	→	**array of** *Type*	may be nested

Statements

Stmt	→	ID(*CallParams*)	procedure call
Stmt	→	**if** *Exp* **then** *Block*	
Stmt	→	**if** *Exp* **then** *Block* **else** *Block*	
Stmt	→	**while** *Exp* **do** *Block*	
Stmt	→	**return** *Ret*	maybe empty
Stmt	→	*LVal* := *Exp*	assignment

Function and Procedure Parameters and Index Lists

CallParams	→	*Exps*	
CallParams	→	ε	may be empty
Exps	→	*Exp* , *Exps*	
Exps	→	*Exp*	never empty

L-Values and Expressions

LVal	→	ID	
LVal	→	ID[*Exps*]	index into an array
Ret	→	*Exp*	return statement may or...
Ret	→	ε	...may not include an expression
Exp	→	NUMLIT	
Exp	→	LOGICLIT	
Exp	→	CHARLIT	
Exp	→	STRINGLIT	
Exp	→	{ *Exps* }	Special array literal, never empty
Exp	→	*LVal*	simple and indexed identifiers
Exp	→	NOT *Exp*	boolean negation
Exp	→	*Exp* OP *Exp*	binary operations
Exp	→	(*Exp*)	
Exp	→	ID (*CallParams*)	function call
OP	→	+ | - | * | / | = | < | AND | OR	

Figure 3: Syntax of **Paladim**

Operators are the usual arithmetic, comparison, and boolean binary operators.

All binary operators are *left-associative*, and have the usual precedence: Addition and subtraction bind weaker than multiplication and division, comparisons bind weaker than arithmetic, and boolean operators bind weaker than comparisons.

Line comments start by `//`, and make the scanner filter out the rest of the line.
Whitespace is irrelevant for **Paladim**, and no lexical atoms contain whitespace.

3.2.2 Meaning of Paladim Programs

Paladim is a typical typed imperative language: one can evaluate arithmetic and boolean expressions (with the usual operations), and assign values of appropriate type to (previously declared) variables in statements. All **Paladim** expressions have a *statically determined type*, and there is *no automatic conversion* between types. For example, a variable of type `int` cannot be used as a `bool` without explicit conversion. Conversion between different types is realised by comparison operations and special predefined functions (see below).

A statement for conditional execution (`if`) and a loop statement (`while`) realise control flow in the small, while functions (returning values) and procedures (not returning anything) provide the larger structure of a **Paladim** program.

At the large scale, a **Paladim** program consists of a set of functions and procedures, which may take up to nine arguments. The functions and procedures used in a program may be declared in any order, but must reside in a single file (there is no module system). Every valid **Paladim** program must contain a procedure called `main`; a **Paladim** program is executed by invoking it.

Paladim provides some **predefined functions** for special tasks:

`chr` takes an integer argument and (returns) converts it to a char value, e.g., `chr(97)` returns character `'a'`.

`ord` takes an character argument and (returns) converts it to an integer value, e.g., `ord('a')` returns integer `97`.

`read` is a function without parameters which reads a value of basic type `int`, `bool` or `char` from the console and returns it. The (basic) type of the result is determined by the type checker from the context in which `read` is used (read is *polymorphic*). Booleans are read/written as `int` values (0 is *false*, 1 is *true*).

`write` is a polymorphic procedure which takes exactly one argument, and prints out the value of the argument on the console. For the moment, `write` supports arguments of basic type (int, char, or bool).

`new,len` are special arrays functions described in the following section.

3.2.3 Arrays in Paladim

Variables in a **Paladim** program may be declared to have array type, where the arrays can also be nested. For example, identifier `a` can be declared to hold a 3-dimensional array of `int`s, by declaration `a : array of array of array of int`. The number of dimensions of an array is called the *rank*, e.g., the rank of `a` is 3.

The special function `new` allocates a new array. The statement `a := new(3,4,5)` will allocate a new array with $3 \cdot 4 \cdot 5 = 60$ integer elements, where all elements are initialized to zero, and assign it to a variable `a`. A runtime error is raised if any of the arguments (sizes for each array dimension) is less or equal to zero.

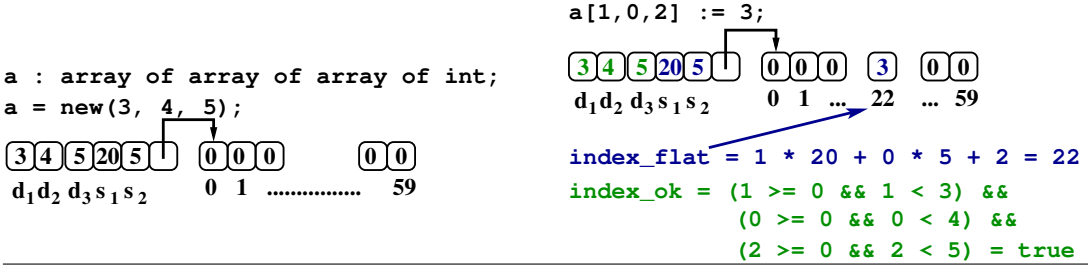

Figure 4: Array Layout.

The left-hand side of Figure 4 shows the array layout of the created array (in the **Paladim** heap): The header of an n-dimensional array consists of $2 \cdot n$ words (one word is 4 bytes): The first n, generically denoted by $d_1, \ldots, d_n$ store the size in each dimension, in our case `3`, `4`, and `5`. The next $n-1$ words, denoted by $s_1, \ldots, s_{n-1}$ are

the strides[4] of the first $n-1$ dimensions. They are computed recursively from the last dimension: $s_n = 1$, $s_i = s_{i+1} \cdot d_{i+1}$. In our example, $s_3 = 1$ (not represented), $s_2 = 1 \cdot 5 = 5$ and $s_1 = 4 \cdot 5 = 20$. Finally, the last word of the array is a pointer to the array's *data area* in memory. The size of this data area is $d_1 \cdot d_2 \cdot \ldots d_n \cdot s$ bytes, where s is the size of the array base type (4 Bytes for `int`, 1 Byte for `bool` and `char`).[5] The size in the example $3 \cdot 4 \cdot 5 \cdot 4 = 240$ Bytes (60 `int` elements).

Subsequently, one can assign a new value to an element of `a` via an *indexed LValue* assignment, e.g., `a[1,0,2] := 3`, and also read and use a value inside `a` via indexing, e.g., `write(a[1,0,2])`. The address computation for indexing into an existing array is illustrated on the right of Figure 4. The flat index of the respective indexed element inside the array's data area is computed by summing up the products of index and stride in each dimension, i.e., $i_1 \cdot s_1 + \ldots + i_n \cdot s_n$ (where s_n is always 1). In our example, the index `a[1,0,2]` corresponds to an offset of $1 \cdot 20 + 0 \cdot 5 + 2 \cdot 1 = 22$ from the start of the array, hence the 22^{nd} element in the array's content is set to 3. If any given index is negative or greater than the corresponding dimension size (e.g. if for `a[`i_1`,...,`i_n`]`, there is a $i_k > d_k - 1$ or $i_k < 0$), execution stops with a runtime error.

The *size in one dimension* of an array can be queried at runtime by the special function `len`. In our example, `len(0,a)` would return 3, the size in the first dimension. Similarly, `len(1,a)` and `len(2,a)` would return 4 and 5. If the first argument of `len` is negative or greater than the rank of the array, a runtime error is raised (for instance, both `len(-1,a)` and `len(4,a)` result in runtime errors).

Note that both `new` and `len` are polymorphic function: `new` receives an *arbitrary number* `n` of integer-type arguments, and the type checker must determine the *type of the result array* from the context where `new` is used. `len` takes two arguments: an integer value and an array *of arbitrary type*, and returns an integer, i.e., the size of the `n`th dimension of array `a`.

3.3 Project Tasks

1. A Parser for Paladim Using the *mosmlyac* Parser Generator

The handed-out code comes with a top-down parser for **Paladim**. This parser does not support some of the necessary syntax for subtask 2, and it is hard to maintain.

Your task is to replace this parser by one which is generated using the *mosmlyac* parser generator, and which supports all necessary syntax for the other tasks. This parser's `Parser.grm` file should be written from scratch.

Note that you must change `Driver.sml` to use *your* implementation instead of the top-down parser. This is done by uncommenting the appropriate lines in the body of the `compile` function.

The solution to this subtask should be delivered with the milestone.

[4] The stride of dimension i can be intuitively described as the number of elements which are jumped in a row-major array layout when increasing only the i^{th} index by one.

[5] For a char or bool array, the allocated area is rounded up to the next multiple of 4, to preserve alignment.

2. Integer Multiplication and Division, and Boolean Operators

The delivered compiler only implements addition and subtraction. Your task is to implement full support for arithmetic operations, i.e. multiplication and division (which have precedence over addition and subtraction).

Furthermore, the compiler only implements the boolean `and` operator. You should implement support for the missing boolean operators, i.e. `or` and `not`. Conjunction (`and`) has precedence over disjunction (`or`), but binds weaker than the unary negation (`not`). As an example, the expression $b = c$ **or not** $a = b$ **and** $a = c$should be parsed as $(b = c)$ **or** ((**not** $(a = b)$) **and** $(a = c)$).

The parser of your milestone submission should be able to parse expressions with these operators correctly.

3. Simple Type Inference and Type Checking the `new` Function

The special functions `read` and `new` require the type checker to determine their particular type from the calling context. This task is about enabling this inference, requiring you to modify the function `typeCheckExp` in file `Type.sml`.

To type-check calls to `read`, type-checking for expressions must, whenever possible, determine an *expected type* from the context of the expression, and pass this expected type down as an inherited attribute when type-checking any sub-expressions). For example, if a plus-expression is type-checked, its two subexpressions are expected to be `int`s. The inherited attribute allows for type-checking expressions like `read() + read()` because the context determines that both `read()` should return `int` values. Similarly, one can type-check `chr(read()) = read()` because the context determines that (a) the first `read` should return an `int`, since function `chr` takes an `int` argument, and (b) the second `read` should return a `char`, since the arguments of a comparison must have the same type, and `chr` returns `char`.

For simplicity, it is allowed that `read() = chr(read())` fails to type-check, i.e. it is fine to use (only) the resulting type of the *left* sub-expression as an expected type for the *right* sub-expression, rather than implementing all possible cases. Finally, an expression like `read() = read()` will fail to type-check because there is no unique way of determining the return type of the two `read()`, e.g., they may both return a boolean value, or a character value, or an integer value.

As a second, related task, you should implement type-checking for the predefined (polymorphic) function `new`, which was explained in Section 3.2.3. Function `new` requires checking that all arguments are of integer type, and the expected type attribute is needed to determine the type of the resulting array (and especially the element size to allocate).

See Appendix B for more hints.

4. Type Checking and Code Generation for Array Indexing

As described in Section 3.2.3, a variable holding an array can be used in indexing expressions, both to use the stored value in an expression, and to assign a new value to the location in an assignment. This array indexing is not implemented in the type checker and in the compiler, your task is to implement it.

Type-check an indexed array by adding the corresponding case to the function `typeCheckExp` in file `Type.sml`. You must check that all indices are integers, and that the number of indices matches the array's rank, i.e., full indexing.

Generate machine code to compute the address of a certain array element by adding the corresponding case to function `compileLVal` in file `Compiler.sml`. You should review the array layout description in Section 3.2.3 to find out how the respective address offset is computed, involving the stored *strides* (and the type information about the array elements). Your implementation should raise a runtime error when an index is out of bounds in any of the dimensions.

Bonus: The computation of the flat index into the array's content can be accomplished in a manner of similar efficiency to the one described in Section 3.2.3, but which uses only the array's dimension sizes (and *not* its strides). A bonus will be offered if you present how this may be accomplished and/or if you actually implement it (instead of the suggested solution).

See Appendix C for more hints.

5. Call-by-Value-Result Semantics for Procedures

In many languages, procedures are used when it is necessary to return more than one value from a subroutine. Procedure calls in the **Paladim** language should therefore copy back their argument values, in a *call-by-value-result* policy.

For example, the program in Figure 5 should output `3 4`, i.e., `x` and `y` are *copied in* to `a` and `b` at the beginning of the call to `f`, and upon return from `f`, the values of `a` and `b` are *copied out* into `x` and `y`, respectively.

DATA/proctest.pal

```
program proctest;
procedure f(a : int; b : int)
begin a := a + 1; b := b + 1; end;

procedure main()
var x : int; y : int;
begin
   x := 2; y := 3;
   f(x, y);
   write(x); write(' '); write(y);
end;
```

Figure 5: **Paladim** example program using call-by-value-result

Your task is to implement a *call-by-value-result* policy for **Paladim** procedures in the interpreter and in the MIPS code generator, by modifying the `callFun` function in `TpInterpret.sml` and functions `compileF` and `compileStmt` (case for procedure call) in `Compile.sml`.

Interpreter Hint: Remember that each function/procedure call maintains its own symbol table that associates a variable name with its value. For any actual-argument expression that is a variable, you need to (i) get the final value, i.e., after the call, of the corresponding formal argument of the procedure, and (ii) to update the corresponding entry in the symbol table of the caller procedure.

Code-Generator Hint: In the compiler, code already exists that moves actual arguments, stored in symbolic registers, to named (caller) registers when calling procedures and functions. The code generated for a procedure needs to be modified to move the symbolic registers associated to the formal arguments back into their corresponding named (caller) registers at the very end of the function execution. The code generated for a procedure *call* need to be modified so that immediately after the call, the symbolic registers holding the actual arguments are updated with the values of their corresponding (named) caller registers.

A Reading Material

[1] Moscow ML – a light-weight implementation of Standard ML (SML). http://mosml.org, 2013. New release 2.10 in 2013 with substantial improvements.

[2] MARS, a MIPS Assembler and Runtime Simulator, version 4.4. http://courses.missouristate.edu/kenvollmar/mars/, 2002-2013. Manual: http://courses.missouristate.edu/KenVollmar/MARS/Help/MarsHelpIntro.html.

[3] David A. Patterson and John L. Hennessy. *Computer Organization & Design, the Hardware/Software Interface*. Morgan Kaufmann, 1998. Appendix A is freely available at http://www.cs.wisc.edu/~larus/HP_AppA.pdf.

[4] Torben Ægidius Mogensen. *Introduction to Compiler Design*. Springer, London, 2011. Previously available as [6].

[5] Jost Berthold. MIPS, Register Allocation and MARS simulator. Based on an earlier version in Danish, by Torben Mogensen. Available on Absalon., 2013.

[6] Torben Ægidius Mogensen. *Basics of Compiler Design*. Self-published, DIKU, 2010 edition edition, 2000-2010. First published 2000. Available at http://www.diku.dk/~torbenm/Basics/. Will not be updated any more.

B Type Checking Hints for Tasks 2, 3 and 5

Paladim is a strongly (statically) typed language, i.e., each expression, function, etc., has a unique type that is computed at compile time. The type checking phase, implemented in file `Type.sml` via `typeCheck*` functions, receives as input an untyped program, i.e., that uses the representation of file `AbSyn.sml`, and verifies that all type rules are satisfied and produces a typed program, i.e., that uses the representation of file `TpAbSyn.sml`.

The untyped representation, i.e., `AbSyn.sml`, targets ease of parsing, e.g., the array type is defined recursively as `datatype Type = ... | Array Type * Pos`, and provides very limited type support.

The typed representation, i.e., `TpAbSyn.sml`, is more suited for implementing code transformations such as code generation. For example the array type is defined as `Array of int * BasicType`, where the first parameter is the array's rank and `BasicType` is one of `Int`, `Char` or `Bool`. More importantly, the typed representation retains whatever (type) information is necessary to easily compute the type of values, expressions, statements and function by means of `typeOfVal`, `typeOfExp`, `typeOfStmt`, and `typeOfFun` functions, respectively. For example, the representation of a variable identifier contains its type, i.e., `type Ident = string * Type`,

and the representation of a function identifier contains the function signature, i.e., `type FIdent = string * (Type list * Type option)`, where the list of types corresponds to the arguments' types and the type option corresponds to the return type of a function (and is `NONE` for procedures).

Tasks 2, 3, and 4 requires implementing parts of function `typeCheckExp`, whose signature is presented below.

```
typeCheckExp ( vtab: VTab, f: AbSyn.Exp, etp: ExpectType ): TpAbSyn.Exp

Type checks an expression:
  Input : 1. the vtable, which associates a variable with its type
          2. an untyped expression, i.e., 'AbSyn.Exp',
          3. an expected type,derived from the context (inherited atrib)

  Output: a typed expression, i.e. 'TpAbSyn.Exp'.
```

Note that `typeCheckExp` returns an expression, and thus we are able to use `typeOfExp` if we need the types of the subexpressions. (hint: have a look at the pattern matching case in `typeCheckExp` for `Plus`.)

The expected type of the expression, `etp`, is used for type checking polymorphic expressions, such as calls to `read()`, which cannot be uniquely typed otherwise. If `etp` is unknown at such points then type checking fails by throwing an exception, e.g., see the pattern matching case in `typeCheckExp` for `AbSyn.FunApp ("read"...)`.

The datatype of `etp` is `ExpectType` which is also located in `Type.sml` and looks like this:

```
datatype ExpectType = SomeArray of TpAbSyn.BasicType
                    | KnownType of TpAbSyn.Type
                    | UnknownType
```

Constructor `KnownType` is used when the expected type is fully known (while `SomeArray` should be used when only the basic type of an array is known, but not the array's rank).

Non-trivial examples of how the expected type is computed/propagated are, for example, (i) the type checking of a function call, in which the expected types of the actual arguments are determined from the function's signature, see implementation of `typeCheckExp( vtab, AbSyn.FunApp (fid, args, pos), _)`, or (ii) the type checking of assignment statements, in which the expected type of the right-hand-side expression is determined from the type of the left-hand side variable, see `typeCheckStmt( vtab, AbSyn.Assign(AbSyn.Var(id), e, pos) )`.

C Code Generation Hints for Task 4

Code generation is implemented in file `Compiler.sml` by a set of mutual recursive functions (`compile*`) that correspond to the constructs of the abstract syntax tree. For example, generating code for a function (`compileF`) requires generating code for all statements in the function body (`compileStmt`), which requires generating code

for the statement's expressions (`compileExp`), etc.

Task 4 requires you to implement the following case of function `compileLVal`:

```
compileLVal ( vtab : VTab, Index((n,t),inds) : LVAL, pos : Pos ) :
            ( Mips.mips list * Location )
```

where detailed comments are provided at that point in file `Compile.sml`.
The arguments of `compileLVal` are:

`vtab` A symbol table that maps any variable name to a register name, i.e., the register that will record the variable's value.

`Index ((n,t),inds)`
The index which needs to be compiled, where `n` is the name of the array, `t` is the type of array-variable `n`, and `inds` is the list of integer-expression indices.

`pos` The position of the `LVal` in the source code, for use in error messages.

Function `compileLVal` returns a list of MIPS instructions, i.e., `Mips.mips list`, that computes the (memory address) location of the indexed element, and the name of the register that records this location (`Location`).

A location is represented via one of the two constructors below:

```
datatype Location = Reg of string (* in given register *)
                  | Mem of string (* at memory address held in register *)
```

in order to distinguish between registers holding a value and registers holding a memory address. More precisely, when compiling an l-value that corresponds to a variable we use constructor `Reg`, and when compiling an array index we use constructor `Mem`. For code examples that illustrate how `Location` is used, please take a look at the implementation of an assignment statement in `compileStmt`.