UNIVERSITY OF COPENHAGEN
Department of Computer Science
Jost Berthold berthold@diku.dk
Cosmin Oancea cosmin.oancea@diku.dk

13 January 2014, 9:00

Exam for Course "Compilers" ("Oversættere"), January 2014

This is the exam assignment for the course "Compilers". The exam will be assessed on the seven point grading scale (grades -3 to 12). Editing time starts on Monday, 13 January 2014, 9:00, and solutions must be handed in before **Friday, 17 January 2014, 14:00**.

Task Sets in the Exam The exam consists of four *task sets* to choose from, and each task set contains four tasks. The task sets have similar structure and overall difficulty, but contain different individual tasks. You should solve only one of the four task sets.
Students who worked together as a group on the G-assignment of this course are not allowed to choose the same task set. Please try to achieve mutual agreement. If this is not possible, the rule is that an older group member may choose before a younger one. In case of severe problems with this procedure, please contact the teachers via mail.

Tasks and Requirements for Your Solution Each task set consists of one theory question (counting with 25%) and three programming tasks (counting 75%).
The programming tasks relate to different areas: extensions to the expression language by new or modified operators; new control flow constructs (loops or branches); and array-specific functionality.
You do not need to modify the **Paladim** interpreter (but it might be a useful debugging tool). Your report should document your implementation to sufficient detail. In particular:

Adding or extending operators: Discuss how the new expression is type-checked and how machine code is generated; reflect about how the changes affect the existing implementation; test that operator precedences are as desired and that the operator works as expected.

Adding functionality specific to arrays: Discuss the main ideas of how the new construct should be implemented. Explain in some detail how it is type-checked and how code is generated.

Adding new control flow constructs: Discuss changes to scanner and parser, whether/how type-checking is affected, and how machine code is generated.

As in the G-assignment earlier you should test your implementation, and solutions without tests will get *lower grades*. Add your tests to the folder `DATA/`) and document them in the report.
In case of errors or unspecified details in the exam text, examinees should take a decision themselves and explain the assumptions for their solution. If any corrections are made to the exam tasks, they will be published in the course's discussion forum in Absalon.
Your report should document which observations you made and on which assumptions your solution depends, and justify the design decisions in your implementation. Four pages are estimated to be the minimum number of pages necessary, and the length should not exceed 16 pages (excluding an appendix for code). It is the *report* which is the basis of the grading. Therefore, it is important to include in the report the most important code that *you yourself wrote or modified*. To facilitate testing your solutions, you are asked to also upload your entire source code to Absalon. Remember to comment your source code so that it is easy to understand.

How to Hand In the Solution A solution to the exam consists of the following parts:

- A written report in pdf format which describes your implementation and discusses its essential parts (see above).
  Please include a cover page stating your KU user name and the chosen task set.
- The code you wrote to implement your solution (as a zip or tar.gz archive).

The deadline (17 January 2014, 14:00) is *strict*, both code and report have to be *uploaded to Absalon before the deadline* (we allow a few minutes grace to avoid technical problems, but the Absalon page will be closed shortly after the official deadline, so do not count on this extra time). To ensure against corrupt uploads or missing files, you should download and check your own submission after uploading it. You can update your uploaded solution until (but not after) the deadline.
Report and code should be uploaded as two files. The report should be provided in pdf format. The code should be uploaded as an archive containing the entire directory in the same structure as the compiler source code handed out.

Exam policy A take-home exam is like any other written exam; i.e. no exam-relevant communication between students is allowed. However, a general discussion of the course material is allowed, but **absolutely no** program sharing, no joint design of algorithms, no exchange of solutions, or the like. At most you can ask each other to clarify parts of the general course material (book, lecture slides).
If you use any material from other sources for your solution, those parts must be *clearly identified* and the authors must be named. Obviously, such parts *cannot count* towards your own grade. If the examiners discover that you received substantial help or went beyond a general discussion of the course material, it will count as cheating.

1. (Theory) Liveness-Analysis and Register-Allocation.

Perform register allocation with the intermediate code given here:

a) Show **succ**, **gen** and **kill** sets for every instruction in the program.

b) Compute **in** and **out** sets for every instruction, show the fix-point iteration.

c) Draw the interference graph for `a`, `b`, `c`, `d`, `e` and `f`.

d) Color the interference graph with 4 colors.

Show the stack, i.e., the three-column table of (Node | Neighbors | Color).

```
1:   LABEL loop
2:   b := a + c
3:   d := 0 - b
4:   e := d + f
5:   IF e = 0 THEN labt ELSE labf
6:   LABEL labt
7:   f := 2 * e
8:   GOTO endif
9:   LABEL labf
10:  a := d + e
11:  e := e - 1
12:  LABEL endif
13:  a := f + c
14:  IF a = 0 THEN loop ELSE endloop
15:  LABEL endloop
16:  RETURN a
```

2. Extend the `or` operator to work bitwise for `int` and `char` in Paladim.

In the G-assignment, **Paladim** expressions were extended by an `or` operator for `bool`. In this task, you should extend the operator to also work on the types `int` and `char`. As a result, the `or` operator now has type `a * a` $\rightarrow$ `a` where `a` can be any basic type. In case `a` is `int`, it should compute the integer result by pairwise combining the bits of the binary representations of the two operands with $\texttt{or}_{bit}$ (defined on the right side).

$\texttt{or}_{bit}$	0	1
0	0	1
1	1	1

For example, the base-2 representation of `17` is $\texttt{(10001)}_2$, and of `11` is $(01011)_2$, hence

$\texttt{17 or 11 = (10001)}_2 \texttt{ or (01011)}_2 \texttt{ = (11011)}_2 \texttt{ = 27}$.

For characters (`char`), the operation should work in a similar way on the 8-bit representation of the character.

The MIPS instruction for `or`ing two integer registers is `Mips.OR` (and `Mips.ORI` when the second argument is a constant).

- Implement support for the new `or` operation throughout the compiler, i.e., in type checker and code generator (interpreter not required).
- No changes are required to the grammar because **Paladim** already uses a grammar rule *Exp* $\rightarrow$ *Exp* `or` *Exp*.

  Discuss in your report whether this operation should get a different precedence and what the implications would be for the implementation.

Hint: Please note that the expected argument types of an `or` expression must now be kept flexible, whereas they were known to be `bool` before.

3. Polymorphic Functions `shape` and `view` in Paladim

Extend the **Paladim** language with the built-in polymorphic functions `shape` and `view`:

- `shape` has type $Array(n,\ \alpha) \rightarrow Array(1, int)$, i.e., it receives as argument an array of arbitrary rank n (with element type $\alpha \in$ `{int,bool,char}`), and returns its shape, i.e., returns a one-dimensional array containing n integers, which represent the sizes of the dimensions of the input array.

- `view` has type $Array(n, \alpha) * int_1 * \ldots * int_m \rightarrow Array(m, \alpha)$. It receives as argument an array of arbitrary rank n (with element type α), and an arbitrary number m of integer parameters, and returns an m-dimensional "view" of the input array. We denote the m integer parameters of `view` with $s_1, \ldots s_m$. The result array shares (aliases) the first $s_1 * \ldots * s_m$ basic-type elements of the input array but under a different shape, i.e., an array of rank m in which the sizes of the m dimensions are $s_1, \ldots, s_m$. If the total number of basic-type elements of the result array, i.e., $s_1 * \ldots * s_m$, is greater than the total number of basic-type elements of the input array, then a runtime error should terminate program execution.

For example, in the program `ShapeViewExample` aside, `a` is a two dimensional array of dimensions 2 and 5, respectively, which contains $2 * 5 = 10$ integers. The shape of `a` is therefore a one-dimensional array `s` with two elements: `{2, 5}`, i.e., the dimensions of array `a`.

The call `view(a, 2, 2, 2)` reshapes the first $2*2*2 = 8$ elements in the flat representation of array `a` into a three-dimensional array `b`, i.e., of rank 3, in which all dimensions of `b` have size 2. Note that array `b` aliases array `a`, hence updating an element of `b` will also update (a different index) in array `a`, e.g., `write(a[0,0]` prints 33 due to the previous assignment `b[0,0,0] := 33`.

Example code for `shape` *and* `view`

```
program ShapeViewExample;
procedure main()
var s : array of int;
    a : array of array of int;
    b : array of array of array of int;
begin
    // flat size of array a is: 2*5 = 10
  a := {{1,2,3,4,5},{6,7,8,9,10}};

  s := shape(a); // s = {2, 5}
  b := view(a, 2, 2, 2);
    // b := {{{1,2},{3,4}},{{5,6},{7,8}}}

  b[0,0,0] := 33;
  write(a[0,0]); //writes 33 because the first
                 // 8 elems of array a are
                 // shared with array b!
  b := view(a, 2, 2, 3); // runtime error:
              // 2 * 2 * 3 = 12 > 2 * 5 = 10!
end;
```

Your task is to implement and to report the implementation of the polymorphic `view` and `shape` functions throughout the compiler, i.e., type checker and machine code generator (interpreter not required):

- No changes are required to the compiler's lexer and parser, since `view` and `shape` are represented as (special) function calls.

- For type checking, extend function `typeCheckExp` to type-check the new functions.

  `typeCheckExp( vtab, AbSyn.FunApp ("view", args, pos), etp )=...` and
  `typeCheckExp( vtab, AbSyn.FunApp ("shape", args, pos), etp )=...`.
  Explain the type rules of `shape` and `view` in your report and describe how they were implemented.

- For machine-code generation, extend the function `compileExp` with the new cases.

  `compileExp( vtable, FunApp (("shape",signat),[arr],pos), place ) = ...`
  `compileExp( vtable, FunApp (("view", signat),arr:dims,pos),place)= ...`

Hints: (1) Note that *the content* of the array returned by `shape(a)` can be *the start address* of the representation of array `a`, i.e., you just need allocate two words, the first being set with the rank of `a`, the second with the start address of `a`. Explain why.

(2) The implementation of `view` resembles the implementation of `new`, with the difference that the array's content is *not* allocated and initialized, but rather the start-address of the content of the result array is the same as the start-address of the content of the input array. Explain why.

4. Repeat-Until, Break and Continue Statements in Paladim

Extend the **Paladim** language with a `repeat-until` loop, and with `break` and `continue` statements. A `repeat-until` loop is very similar to a `while` loop, but the body of the loop is executed *until* a condition holds, but at least once (the first time).

`break` and `continue` statements are only allowed inside loops. A `break` statement causes execution to immediately exit the loop. A `continue` statement causes execution to immediately proceed to the next iteration, i.e., to check the loop condition and either exit the loop or restart at the top.

The example on the side illustrates the syntax and semantics of the `repeat-until` loop and the `break` and `continue` statements. The loop body is executed at least once and repeated as long as `i` is unequal to `N` at the iteration's end, (or until a `break` statement is reached inside the loop). The body of the loop reads an integer in `t`:

- if `t`'s value is negative then the loop is exited due to the `break` statement,
- if `t`'s value is 0 then control jumps to just before testing the loop condition `i = N`,
- otherwise, `a[i]` is set to `t` and `i` is incremented.

Example code for `repeat-until`

```
program RepeatExample;
procedure main()
var i : int; t : int; N : int;
    a : array of int;
begin
  N := read();
  a := new(N);

  i := 0;

  repeat
  begin
    t := read();
    if(t < 0) then break
    else if(t = 0) then continue;
    a[i] := t;
    i := i + 1;
  end;
  until(i = N);

end;
```

The **Paladim** grammar is to be extended with the following productions:

Stmt → **repeat** *Block* **until** *Exp*
Stmt → **break**
Stmt → **continue**

where *Block* is the body of the loop and *Exp* is the loop termination condition.

Your task is to implement the `repeat-until`, `break` and `continue` statements throughout the **Paladim** compiler: scanner and parser, type-checker, and code generator (interpreter not required). Use the following representation in the abstract syntax:

```
and Stmt = ... (* as before *)
    | RepeatUntil of StmtBlock * Exp * Pos
    | Break       of Pos
    | Continue    of Pos
```

where `StmtBlock` is the body of the loop and `Exp` is the loop termination condition.