UNIVERSITY OF COPENHAGEN
Department of Computer Science
Jost Berthold Cosmin Oancea
berthold@diku.dk 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) Parameter-Passing and Scoping.

Calling Mechanisms

```
int f(int a, int b, int c) {
    a = c + a;
    b = 2 * a;
    c = 2 * c;
    return (a + b + c);
}

int main() {
    int x = 5;
    int y = 10;
    int r = f(x, x, y);
    print(x); print(y); print(r);
}
```

With the C-like program on the side, what is printed if:

1.a) call-by-value is used? (Explain briefly.)

1.b) call-by-reference is used? (Explain briefly.)

1.c) call-by-value-result is used? (Explain briefly.)

Scoping Mechanisms

```
int x = 33;  // global x

void f()      { print(x);                          }
void g(int y) { int x = y * 3; f();                }
void h(int x) { if( x = 3 ) g(x-1); else f(); }
```

With the C-like program on the side, what do `h(3)` and `h(11)` print, respectively, if

1.d) static scoping is used? (Explain briefly.)

1.e) dynamic scoping is used? (Explain briefly.)

2. Add a raise-to-power operator for integers to Paladim.

The power-operator is written as a `^` symbol (circumflex) in infix notation, has type `int * int` $\rightarrow$ `int`, and is not defined when the exponent argument is negative.
For example, `x ^ 0` $\equiv$ `1`, `x ^ 3` $\equiv$ `x * x * x`, and `x ^ (0-1)` raises a runtime error.

- Add the grammar rule $Exp \rightarrow Exp$ `^` Exp, where `^` is left associative and binds tighter than multiplication. Add a new AbSyn node:
  `datatype Exp = ... | Pow of Exp * Exp * pos`
- Implement the new raise-to-power operation throughout the compiler, i.e., type checker and code generator (interpreter not required). Do not forget to signal an error when the exponent evaluates to a negative integer.

3. A Flat-ZipWith Higher-Order Function in Paladim

Extend the **Paladim** language with the higher-order function `zipWith` that:

- receives as parameters a function (name) and two array expressions,
- raises a runtime error if the input arrays do not have identical shapes, and
- returns an array of the same rank and shape as the input arrays whose
- basic-type elements are obtained by applying the function parameter to each pair of basic-type element located at the same position in the input arrays.

Thus, the type of `zipWith` is: $(\alpha * \beta \rightarrow \gamma,\ Array(n,\ \alpha),\ Array(n,\ \beta)) \rightarrow Array(n,\ \gamma)$, where α, β and γ are basic types, i.e., $\alpha, \beta, \gamma \in \{int, char, bool\}$, and $Array(n,\ \mu)$ is the type of an array of rank n and basic type μ.

Example code for zipWith

```
program FlatZipWithExample;
function  f(a : int, b : char) : int
  return a + ord(b);

procedure main()
var a : array of array of int;
    b : array of array of char;
    c : array of array of int;
begin
  a := {{1,2,3},{4,5,6}};
  b := {{'e','f','g'},{'h','i','j'}};

  c := zipWith(f, a, b);
  write(c[1,2]); // prints 112
end;
```

For example, in the nearby Figure, the call to `zipWith` in program `FlatZipWithExample` will succeed because arrays `a` and `b` have identical ranks, i.e., 2, and shapes, i.e., the sizes of the outer and inner dimension are `2` and `3`, respectively. Next, the two input arrays `a` and `b` are (semantically) flattened to one-dimensional arrays of `int` and `char` basic-type elements, i.e., `{1,2,3,4,5,6}` and `{'e','f','g','h','i','j'}`. The result of `zipWith` is obtained by applying `f` to each pair of elements located at the same index in `a` and `b`: `{f(1,'e'), f(2,'f'), f(3,'g'), f(4,'h'), f(5,'i'), f(6,'j')}`, and finally, the result array is given the shape of the input arrays, i.e., `c = zipWith(f, a, b) = {{f(1,'e'), f(2,'f'), f(3,'g')}, {f(4,'h'), f(5,'i'), f(6,'j')}} = {{102,104,106},{108,110,112}}`.

Your task is to implement and to report the implementation of the flat-`zipWith` function 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 the untyped representation of `map`, i.e., the one in `AbSyn.sml`, is simply a function call.
- For type checking, fill in the file `Type.sml` the implementation of `typeCheckExp( vtab, AbSyn.FunApp ("zipWith", args, pos), etp )=...` Note that the *typed representation* of `zipWith` in file `TpAbSyn.sml` uses `Exp`'s type constructor `datatype Exp = ... | ZipWith of FIdent * Exp * Exp * Pos`, where the first argument of `zipWith` corresponds to the function, the second to the array expression and the third to the position of the `zipWith` call. **Remember**: (i) that the type checker transforms an untyped representation into a typed one, in which, for e.g., the type of any expression `e` can be queried with `typeOfExp(e)`, (ii) that `FIdent` contains both the function's name and its signature (see `TpAbSyn.sml`), and (iii) to explain in your report the type rule of map, and how this was implemented.
- For machine-code generation, fill in the file `Compiler.sml` the implementation of `compileExp( vtable, ZipWith( (id,signat),arr1,arr2,pos ), place ) =...`. Remember to raise a runtime error in the case when the shapes of the two input arrays (of `zipWith`) differ. Marks will be subtracted if your implementation is inefficient. Show and explain in your report the most important code snipets of your implementation, and discuss why you think it is efficient (in comparison with alternative solutions).
  Hint: to call a function of identifier (`FIdent`) `f` on a set of symbolic-register parameters `rps`, one may use the function `mkFunCallCode(f, regps, vtab, regres)` in `Compiler.sml`, where `vtab` is the variable symbol table, and `regres` is the register where the result of the function call will be stored.

4. Dijkstra's guarded command statements for Paladim

Extend the **Paladim** language with a "guarded command" (Dijkstra[1]).

The example on the side illustrates the syntax and semantics of the new construct. The `do` construct will *repeatedly* check whether *any* of the given boolean expressions evaluates to true (top to bottom), and execute the respective block as soon as one expression evaluates to true, and then repeats from the start. Execution ends when none of the expressions was true.

Example code for guarded `do`

```
program Guardexample;
procedure main()
var i : int;
begin
  i := read();
  do i < 10 : begin
                i := i + 1;
                write('.');
              end;
  [] 10 < i : begin
                write("Too big, again\n");
                i := read();
              end;
  [] 20 < i : write("never reached!");
  done;
  write("Now i should be 10.\n")
end;
```

In the example program, a number is read in and then repeatedly re-read if it was above 10. If it is below 10, it will be incremented up to 10 while printing dots at the same time. This results in printing $10 - i$ dots. As soon as 10 is reached, the guarded statement exits and the program terminates.

The parser checks that there is at least one guarded branch. Branches are separated by the special symbol combination `[]` (without spaces in between), and the statement ends with a special `done` keyword. And as customary, all blocks are terminated by semicolon. The **Paladim** grammar is extended with the following productions:

Stmt	→	**do** *Exp* : *Block* ; *Branches*
Branches	→	**[]** *Exp* : *Block* ; *Branches*
Branches	→	**done**

The expressions inside a guarded `do` loop are checked from top to bottom. If one of these expressions is true, the block adjacent to the expression is executed, and then execution proceeds from the top again. If several expressions would evaluate to true, only the block adjacent to the first one is executed.

Implement the guarded `do` statement throughout the entire **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 *)
    |  GuardedDo of (Exp * StmtBlock) list * Pos
```

(where the list of condition expressions is stored in original order).

[1] Dijkstra, Edsger W. *Guarded commands, nondeterminacy and formal derivation of programs.* Communications of the ACM 18.8 (1975): 453-457.