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) SLR(1) Parser Construction

Terminals: `id` `*` `=` , and `$` denoting the end of input.

$$
\begin{array}{llr}
G_{LR}: S' \to S\ \$ & & (0) \\
S \to L \texttt{ = } R \mid R & & (1,2) \\
L \to \texttt{*}R \mid \texttt{id} & & (3,4) \\
R \to L & & (5)
\end{array}
$$

Perform SLR(1) parser construction as shown in lecture and exercises with the grammar G_{LR} defined here:

a) Construct a combined NFA with items from all rules, then use subset construction to obtain an equivalent DFA.

b) Compute the FOLLOW sets and determine the reduce actions.

c) The grammar has a shift-reduce conflict under SLR(1) analysis. Give the SLR parse table and point out the conflict.

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

In the G-assignment, **Paladim** expressions were extended by an `and` operator for `bool`. In this task, you should extend the operator to also work on the types `int` and `char`. As a result, the `and` operator now has type `a * a` $\to$ `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 $\texttt{and}_{bit}$ (defined on the right side).

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

For example, the base-2 representation of `17` is $\texttt{(10001)}_2$, and of `11` is $(01011)_2$, hence
`17 and 11 =` $\texttt{(10001)}_2$ `and` $\texttt{(01011)}_2$ `=` $\texttt{(00001)}_2$ `= 1`.
For characters (`char`), the operation should work in a similar way on the 8-bit representation of the character.

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

- Implement support for the new `and` 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* $\to$ *Exp* `and` *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 `and` expression must now be kept flexible, whereas they were known to be `bool` before.

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

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

- receives as parameters a function (name) and an array expression, and
- returns an array of the same rank and shape as the input array that
- is obtained by applying the function parameter to each basic-type element of the input array.

Thus, the type of `map` is: $(\alpha \to \beta,\ Array(n,\ \alpha)) \ \to\ Array(n,\ \beta)$, where α and β are basic types, i.e., $\alpha, \beta \in \{int, char, bool\}$, and $Array(n,\ \alpha)$ is the type of an array of rank n and basic type α.

Example code for `map`

```
program FlatMapExample;
function  f(a : int) : char
  return chr(100+a);

procedure main()
var a : array of array of char;
begin
  a := map(f, {{1,2,3},{4,5,6}});
  write(a[1,2]); // prints 'j'
end;
```

For example, the program `FlatMapExample` in the nearby Figure maps each element of the array literal `{{1,2,3},{4,5,6}}` via function `f` and returns a two-dimensional array of characters, `a`, whose outer and inner dimension sizes are 2 and 3 respectively, i.e., `a` $\equiv$ `{ {'e', 'f', 'g'}, {'h', 'i', 'j'} }`. Intuitively, the input array is flattened, i.e., to a one-dimensional array of int (basic type) elements `{1,2,3,4,5,6}`, then each element is mapped via function `f`, and the result array is then given the shape of the original array.

Your task is to implement and to report the implementation of the flat-map 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 ("map", args, pos), etp )=...` Note that the *typed representation* of `map` in file `TpAbSyn.sml` uses `Exp`'s type constructor `datatype Exp = ...  | Map of FIdent * Exp * Pos`, where the first argument of `Map` corresponds to the function, the second to the array expression and the third to the position of the `map` 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, Map( (id,signat), arr_exp, pos ), place ) =...`. Marks will be subtracted if your implementation is inefficient. Show and explain in your report the most important code snippets of your implementation, and discuss why you think it is efficient (in comparison to alternative solutions).
  Hint: A helper function `mkFunCallCode(f, regps, vtab, regres)` is provided in `Compiler.sml` which can be used to call a function of identifier (`FIdent`) `f` on a set of symbolic-register parameters `regps`, where `vtab` is the variable symbol table, and `regres` is the register where the result of the function call will be stored.

4. A `For` Loop in Paladim

Extend the **Paladim** language with a `for`-loop.

The **Paladim** syntax is extended with the following productions:

Stmt	→	**for** ID = *Exp LoopDir Exp* **do** *Block*
LoopDir	→	**downto**
LoopDir	→	**to**

where ID corresponds to the loop induction variable, e.g., `i`, which is initialized to the value of the first *Exp* in the execution of the first iteration, and *Block* corresponds to the block of declarations and statements that implements the body of the loop.

- If the loop declaration uses keyword `to`, as in `for i = 0 to N-1 do` ..., then, at the end of each iteration: (1) the induction variable is incremented by one and (2) the loop terminates if `i` is greater than the value of the second *Exp*, i.e., `N-1`.
- Otherwise, if the loop declaration uses keyword `downto`, as in `for i = N-1 downto 0 do` ..., then (1) the induction variable is decremented by one at the end of each iteration, and (2) the loop terminates if `i` is less than the value of the second *Exp*, i.e., `0`.

The example on the side illustrates the syntax and semantics of the new construct. The first loop traverses in order the elements of array `a`, and sets each element to the value of its index plus one, while the second loop traverses the array in reverse order and subtracts one from the value of each element of `a`. (Hence at the end of the program invariant `a[i] = i` holds ∀ `i`∈ $\{0, \ldots, N-1\}$).

Example code for `for`

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

  for i = 0 to N-1 do
  begin
    a[i] := i + 1;
  end;

  for i = N-1 downto 0 do
    a[i] := a[i] - 1;

  write( a[N-1] = (N - 1) ); // true
end;
```

Your task is twofold:

- Implement the `for`-loop 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 *)
     | ForLoop of Ident * Exp * bool * Exp * StmtBlock * Pos
  ```

  where `Ident` is the loop induction variable, the first and second `Exp` are as before, the (third) boolean parameter is `false` (`true`) if the loop uses the `downto` (`to`) keyword, and finally the `StmtBlock` parameter is the body of the loop.

- Some languages, e.g., Fortran, require the induction variable to be read-only inside the loop body, for example to guarantee loop termination. Explain how this check could be performed statically in **Paladim** (essay question).