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. Grammar transformation for LL(1)

Terminals: `id ( ) ,`

Analyse the given grammar G_{Tup} for tuple expressions for LL(1) parsing:

$$G_{Tup}: S \rightarrow id \quad (1)$$
$$S \rightarrow (\ T\ ) \quad (2)$$
$$T \rightarrow S\ ,\ T \quad (3)$$
$$T \rightarrow S \quad (4)$$

a) Give a short reason why this grammar is not LL(1), and transform the grammar (using a well-known transformation) to obtain a grammar G'_{Tup} suitable for LL(1) parsing.

b) For G'_{Tup}, determine FIRST sets for all right-hand sides.

c) Add a start production $S' \rightarrow S\$$ and determine FOLLOW sets for all nonterminals.

c) Determine the look-ahead sets for all productions and point out that G'_{Tup} is LL(1).

2. Extend the equality operation in Paladim to work on arrays.

At the moment, the **Paladim** equality operation = can only compare basic types. Your task is to extend equality to also work for arrays.

Obviously, when comparing two arrays, they should have the same type (for instance, `array of array of array of char`). If the arrays have different shape (which should be allowed), they are not equal. If the arrays have the same shape, the contents of the two array should be compared element-wise to determine if they are the same.

- Extend the type-checking routine for the equality operation to allow two arrays of the same type as the arguments.
- Extend the code generator (interpreter not required), implementing the comparison as described above.

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

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

- receives as parameters (i) a function (name) that takes two arguments of basic type α and produces a result also of basic type α, and (ii) an arbitrary array expression,
- and semantically flattens the array to a one-dimensional array of basic-type elements, and reduces the flat array to an element of basic type α.

Thus, the type of `reduce` is: $((\alpha * \alpha) \rightarrow \alpha,\ Array(n,\ \alpha)) \rightarrow \alpha$, where α is a basic type, i.e., $\alpha \in \{int, char, bool\}$, and $Array(n,\ \alpha)$ is the type of an array of rank n and basic type α.

If `a` is an array of arbitrary shape (and basic type), and `{`$\mathtt{a}_1$`, ..., `$\mathtt{a}_n$`}` is the flattened version of `a`, i.e., an one-dimensional array enumerating the basic-type elements of `a`, *then* the semantics of `reduce` is: `reduce(f, a)` $\equiv$ `f (... f( f(a1, a2), a3 ) ..., an)`.

For example, in program `FlatReduceExample` of the nearby Figure, the flattened version of array `a` is `{1,2,3,4,5,6}`, and `reduce(plus, a)` evaluates to `(((((1+2)+3)+4)+5)+6) = 21`.

Your task is to implement and to report the implementation of the flat-reduce function throughout the compiler, i.e., type checker and code generator (interpreter not required):

Example code for `reduce`

```
program FlatReduceExample;
function  plus(x : int; y : int) : int
  return (x + y);

procedure main()
var x : int; a : array of array of int;
begin
  a := {{1,2,3},{4,5,6}};
  x := reduce(plus, a);
  write(x); // prints 21
end;
```

- No changes are required to the compiler's lexer and parser, since the untyped representation of `reduce`, 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 ("reduce", args, pos), etp )=...` Note that the *typed representation* of `reduce` in file `TpAbSyn.sml` uses `Exp`'s type constructor `datatype Exp = ... | Red of FIdent * Exp * Pos`, where the first argument of `Red` corresponds to the function, the second to the array expression and the third to the position of the `reduce` 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 `reduce`, and how this was implemented.
- For machine-code generation, fill in the file `Compiler.sml` the implementation of `compileExp( vtable, Red( (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 with 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 switch statement for Paladim

Extend the **Paladim** language with a `switch` statement over `int`, `char` and `bool` values.

The example on the side illustrates the syntax and semantics of the new construct, by switching over two possible `int` values (`0` and `2`) and a `default`.

The parser should check that all cases of the `switch` statement use literals (but `switch` may use different base types). Therefore, literals should be an own production in the grammar. Another property which is syntactically checked is that every `switch` has a `default` clause. In contrast, any number of `case` clauses can occur, including none.

Example code for `switch`

```
program Switchexample;
procedure main()
var i : int;
begin
  i := read();
  switch i+i:
   case 0: write("i is 0!");
   case 2: var c : char;
           begin
             write("i is 1");
             c := ord(32*i);
             write(c); // prints '!'
           end;
   default: write("is is neither 0 nor 1.")
   write("\nbye!\n");
end;
```

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

Stmt	→	**switch** *Exp* : *Cases*
Cases	→	*Case* ; *Cases*
Cases	→	**default** : *Block*
Case	→	**case** *Lit* : *Block*

Lit	→	NUM
Lit	→	TRUE
Lit	→	FALSE
Lit	→	CHARLIT

The semantics of `switch` is to evaluate the given expression, and compare the values to all literals in the `case` clauses from top to bottom. If the expression evaluates to the value of one of the given literals, the adjacent code block is evaluated and the statement ends. If no literal has the value of the expression, the block in the `default` clause is executed. The `case` clauses are checked from top to bottom. If the same literal occurs in more than one `case` clause, only the block in the first clause is executed.

Implement the `switch` 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 *)
   |  Switch of Exp * Case list * Pos

and Case = Case        of BasicVal * StmtBlock * Pos
         | DefaultCase of StmtBlock * Pos
```

(with a list of cases with the literals as basic values, and known to contain at least the default case *at the end*).