Group Project

Oversættere

Christoffer Horn
Kristian Holm
Damir Gorovic NDC516

18. december 2014

Introduction

This report covers extensions and adjustments for the Fasto compiler language in the group project. We will describe the most significant changes and adjustments we have made in the different files, and will briefly discuss some of the tests and test results.

Indhold

1 Getting started

PERHAPS INTRO HERE

1.1 The lexer

The lexer we were given in the start of the group project contained most of the common tokens/keywords needed. We did not need to change any data which was already there, but we needed to add some new tokens and keywords which were needed for the Fasto language (such as true, false, not, if and so on). The keywords we have added has been done in the following way;

```
fun keyword (s, pos) =
    case s of
        "if"           => Parser.IF pos
      | "then"         => Parser.THEN pos
      | "else"         => Parser.ELSE pos
      | "bool"         => Parser.BOOL pos
      | "true"         => Parser.BOOLVAL (true, pos)
      | "false"        => Parser.BOOLVAL (false, pos)
      | "char"         => Parser.CHAR pos
      | "fun"          => Parser.FUN pos
```

This is just a small example of how it is implemented, but it gives a general idea of how we have approached it.
Other than these keywords we also needed to add some tokens which were not known for the Fasto. A small example is shown here;

```
  | `+`                { Parser.PLUS      (getPos lexbuf) }
  | `*`                { Parser.TIMES     (getPos lexbuf) }
  | "&&"               { Parser.AND       (getPos lexbuf) }
  | `<`                { Parser.LTH       (getPos lexbuf) }
  | `)`                { Parser.RPAR      (getPos lexbuf) }
  | eof                { Parser.EOF       (getPos lexbuf) }
  | _                  { lexerError lexbuf "Illegal symbol in input" };
```

The code shown above should give a decent understanding of how we have approached some of the tokens needed, and everything which we do not

know.[1] A minor note can be added about the fact that we use the "sign for AND because we need to define it as a string since it needs 2 tokens to be recognized.

1.2 The parser

Inside of the Parser.grm file we need to define how we use the tokens which we made in the lexer. First we needed to define *how* out tokens looked when they were in use, so our parser can recognize them. This was done in the following way;

```
%token <(int*int)> PLUS MINUS TIMES DIVIDE DEQ EQ LTH AND OR
%token <(int*int)> IF THEN ELSE LET IN INT BOOL CHAR EOF
%token <string*(int*int)> ID STRINGLIT
```

The "(int*int)"figure shown above is used to get the location of where these tokens are used, so we can send a helpful error message when something goes wrong. the ID and STRINGLIT tokens are konwn as a string, which also needs to have a position in the document.

Next up we need to define the precedence of our different evaluating expressions. This was done quickly and with not much trouble, as we were told in the handed out document which expressions had which precedence and associativity.

Lastly we need to define how we use the different expressions/types/parameters and so on in our language. Here is a few examples of how this was implemented;

```
Exp :     Exp PLUS  Exp  { Plus ($1, $3, $2) }
        | Exp DIVIDE Exp { Divide($1, $3, $2) }
        | FILTER LPAR FunArg COMMA Exp RPAR
                         { Filter ($3, $5, (), $1) }
Type :    INT            { Int }
        | LBRACKET Type RBRACKET { Array $2 } ;
```

[1] All of the tokens which we do not know and have not defined here, we will not recognize (and are therefore not valid) and therefore raise an error

2 Interpreting

After the parser and lexer have been implemented and told how to handle the tokens and identifiers, we move on to the interpreter. In this document we define how to evaluate the different expressions. The following example shows some examples of how this is done;

```
| evalExp ( Minus(e1, e2, pos), vtab, ftab ) =
        let val res1   = evalExp(e1, vtab, ftab)
            val res2   = evalExp(e2, vtab, ftab)
        in  evalBinopNum(op -, res1, res2, pos)
        end
| evalExp ( Negate(e1, pos), vtab, ftab ) =
        let val res1   = evalExp(e1, vtab, ftab)
        in  evalopNum(op ~, res1, pos)
        end
```

The interesting part of these implementations are the inclusion of vtab and ftab. What this basically does is making our code dynamic, in the sense that we are able to look up our previously used functions and variables.
Other than these "simple"implementations, we also implemented the anonymous function (commonly known as Lambda), but we will get back to this later.

3 Typechecking

After we have told our

4 Tests

In this following section we will show relevant tests to prove the correctness of our implementations across the Fasto language.
Apart from the gives tests we received with the Fasto folder (which all work and run correctly), we have implemented a few of our own to test some of the important extensions we have made.

4.1 Precedence

To test that our precedence of expressions are correct we have used the following program;

```
fun int main() =
write(3 + 9 - 15 * 2 / 10)
```

This program returns the correct output "9". It does the multiplication first, then the division and thereafter the addition and subtraction.

4.2 And/Or

To test that our And/Or expressions are handled correctly we have used the following program;

```
fun bool main() =
write(true && true || false)
```

The program returns "True"which is what we expect and want from it. If we choose to remove the and expression and do "write(true || false)"it will return false, which shows that both are expressions are working as intended.

4.3 Not

The program we use to test out "not"expressions looks like this;

```
fun bool main() =
write(not false)
```

This returns true which is the result we expect.