Paladim Compiler

Alexander Worm Olsen - bdj816

Chi Dan Pham - vqr853

Troels Thompsen - qvw203

Group Project Milestone

December 6 2013

Departmen of Computer Science
University of Copenhagen

Contents

1 Introduction **2**

2 Parser.grm **2**

2.1 Definitions . . . 2

2.1.1 Terminals . . . 2

2.1.2 Precedence Rules . . . 3

2.1.3 Non-Terminals . . . 4

2.2 Grammar . . . 5

2.2.1 Program structure . . . 5

2.2.2 Statements, Values and Expressions . . . 6

2.2.3 Parameters and Procedures . . . 6

2.3 Driver.sml . . . 7

2.4 Lexer.lex . . . 7

2.5 Functionality of multiplication . . . 8

3 Testing **10**

1 Introduction

For the milestone assignment we were asked to implement the grammar production rules for the paladim language, which is described in the groupproject description document.

In order to implement the grammar, we have edited the empty file *Parser.grm*, which was already included in the source code handout. We also edited the *Driver.sml* and the *Lexer.lex* source code file, to use our new parser structure instead of the LL1Parser which was included in the handout.

For inspirational purposes we used the file *example.pdf* which we found on absalon, and the groupproject description document. In these documents we found examples for creating type precedence, how to construct terminals and non-terminals, and how to use them correctly in the grammar.

The following sections describe the changes made to each file respectively.

2 Parser.grm

2.1 Definitions

2.1.1 Terminals

The first definition in the parser, is the definition of terminals, which in mosmlyac is called tokens.

```
%token <AbSyn.Pos> TProgram TFunction TProcedure TVar TBegin TEnd TIf TThen TElse TWhile TDo TReturn TArray
TOf TInt TBool TChar TAnd TOr TNot TAssign TPlus TMinus TTimes TSlash TEq TLess TLParen TRParen TLBracket
TRBracket TLCurly TRCurly TComma TSemi TColon TEOF
%token <bool*AbSyn.Pos> TBLit
%token <int*AbSyn.Pos> TNLit
%token <char*AbSyn.Pos> TCLit
%token <string*AbSyn.Pos> TSLit TId
```

Figure 1: Parser.grm Tokens

Here we use the abstract syntax type definitions to define the types of our tokens. The tokens themselves are defined in the *Lexer.lex* file. We simply found all tokens in the lexer, and defined tokens for them.

2.1.2 Precedence Rules

Next we define our precedence rules, to prevent shift / reduce conflicts. The rules at the bottom take the highest precedence.

```

%nonassoc LowPrec
%nonassoc TElse
%right TOr
%right TAnd
%nonassoc TNot
%nonassoc TEq TLess
%left TPlus TMinus
%left TTimes TSlash
```

Figure 2: Parser.grm Precedence

We use left associative precedence for arithmetic operations, ranking *TTimes* and *TSlash* higher than *TPlus* and *TMinus*.

We define comparison operations *TLess* and *TEq* as non-associative, since it is not specified whether comparison is associative in Paladim in the project description.

The logical operation *TNot* is defined as non-associative because it is unary. We define the other logical operations *TAnd* and *TOr* as left-associative (they can be either left- or right-associative, but should not be non-associative). Here *TNot* takes precedence over *TAnd* which takes precedence over *TOr*.

The last two precedence rules are defined to resolve the shift / reduce conflict, which occurs when trying to write productions for If-Then-Else and If-Then. This conflict occurs in the following scenario.

```
if a > b then
    if c = d then
        return a
    else
        return b
```

In this scenario the parser does not know whether to match the production for If-Then-Else, or the production for If-Then. If it shifts, the else belongs to the inner if-statement, but if it reduces, the else belongs to the outer if-statement. In this scenario, we actually always wants to shift, because the else should belong to the closest previous if-then-statement. In order to accomplish this, we define TElse to take precedence over "LowPrec". We later assign "LowPrec" to the If-Then production grammar, and thus resolve the shift / reduce conflict.

2.1.3 Non-Terminals

In this section we define our start symbol, and the non-terminals for our productions. The types used for the non-terminals are defined in *AbSyn.sml*. The actual non-

```
/* types returned by rules below */
%type <AbSyn.Prog> Program
%type <AbSyn.Prog> Prog
%type <AbSyn.Prog> FunDecs
%type <AbSyn.FunDec> FunDec
%type <AbSyn.StmtBlock> Block
%type <AbSyn.Stmt list> SBlock
%type <AbSyn.Stmt list> StmtSeq
%type <AbSyn.Stmt> Stmt
%type <AbSyn.LVAL> LVal
%type <AbSyn.Exp option> Ret
%type <AbSyn.Exp> Exp
%type <AbSyn.Dec list> PDecl Params
%type <AbSyn.Dec> Dec
%type <AbSyn.Dec list> Decs
%type <AbSyn.Type> Type
%type <AbSyn.Exp list> CallParams Exps

%%
```

Figure 3: Parser.grm

terminals are defined in figure 3 in the group project definition document. The start symbol "Program" was defined by us, in order to handle end of file.

2.2 Grammar

In this section we define the grammar used by the parser. For easier analysis we have divided the grammar into sections. In general we built the grammar by looking at the production definitions in figure 3 in the group project definition. The type we return in the productions, are defined in the *AbSyn.sml*.

2.2.1 Program structure

```
Program: Prog TEOF                          { $1 }
;

Prog :
    TProgram TId TSemi FunDecs              { $4 }
;

FunDecs :
    FunDecs FunDec                          { $1 @ [$2] }
  | FunDec                                  { [$1] }
;

FunDec :
    TFunction TId TLParen PDecl TRParen TColon Type Block TSemi
                                            { AbSyn.Func($7, #1 $2, $4, $8, $1) }
  | TProcedure TId TLParen PDecl TRParen Block TSemi
                                            { AbSyn.Proc(#1 $2, $4, $6, $1) }
;

Block :
    SBlock                                  { AbSyn.Block ([], $1) }
  | TVar Decs SBlock                        { AbSyn.Block ($2, $3) }
;

SBlock :$
    TBegin StmtSeq TSemi TEnd               { $2 }
  | Stmt                                    { [$1] }
;

StmtSeq :
    StmtSeq TSemi Stmt                      { $1 @ [$3] }
  | Stmt                                    { [$1] }
;
```

Figure 4: Parser.grm

The most important change in this section, is the Block production. To begin with we had a DBlock production, as described in figure 3 in the group project definition. This caused a shift / reduce conflict, which we resolved by combining the DBlock with the Block production.

2.2.2 Statements, Values and Expressions

```
Stmt :
      TId TLParen CallParams TRParen          { AbSyn.ProcCall (#1 $1, $3, #2 $1) }
    | TIf Exp TThen Block TElse Block         { AbSyn.IfThEl ($2, $4, $6, $1) }
    | TIf Exp TThen Block %prec LowPrec       { AbSyn.IfThEl ($2, $4, AbSyn.Block([],[]), $1)}
    /* prec gives precedence as LowPrec */
    | TWhile Exp TDo Block                    { AbSyn.While ($2, $4, $1) }
    | TReturn Ret                             { AbSyn.Return ($2, $1) }
    | LVal TAssign Exp                        { AbSyn.Assign ($1, $3, $2) }
;

/* L-VALUES AND EXPRESSIONS */
LVal :
      TId                                     { AbSyn.Var (#1 $1) }
    | TId TLBracket Exps TRBracket            { AbSyn.Index (#1 $1, $3) }
;

Ret :
      Exp                                     { SOME $1 }
    |                                         { NONE }
;

Exp :
      TNLit                                   { AbSyn.Literal (AbSyn.BVal(AbSyn.Num(#1 $1)), #2 $1) }
    | TBLit                                   { AbSyn.Literal (AbSyn.BVal(AbSyn.Log(#1 $1)), #2 $1) }
    | TCLit                                   { AbSyn.Literal (AbSyn.BVal(AbSyn.Chr(#1 $1)), #2 $1) }
    | TSLit                                   { AbSyn.StrLit (#1 $1, #2 $1) }
    | TLCurly Exps TRCurly                    { AbSyn.ArrLit ($2,$1) }
    | LVal                                    { AbSyn.LValue ($1, (0,0)) }
    | TNot Exp                                { AbSyn.Not ($2, $1) }
    | Exp TPlus Exp                           { AbSyn.Plus ($1, $3, $2) }
    | Exp TMinus Exp                          { AbSyn.Minus ($1, $3, $2) }
    | Exp TTimes Exp                          { AbSyn.Times ($1, $3, $2) }
    | Exp TSlash Exp                          { AbSyn.Div ($1, $3, $2) }
    | Exp TEq Exp                             { AbSyn.Equal ($1, $3, $2) }
    | Exp TLess Exp                           { AbSyn.Less ($1, $3, $2) }
    | Exp TAnd Exp                            { AbSyn.And ($1, $3, $2) }
    | Exp TOr Exp                             { AbSyn.Or ($1, $3, $2) }
    | TLParen Exp TRParen                     { $2 }
    | TId TLParen CallParams TRParen          { AbSyn.FunApp (#1 $1, $3, # 2$1) }
;
```

Figure 5: Parser.grm

The most important change in this section, is the *TIf Exp TThen Block %prec LowPrec* production. As described in the non-terminal definitions, we need to assign lower precedence to the If-Then production. This is done by adding %prec LowPrec at the end of the production.

The Exp LVal production in this section might be problematic. This production needs to return an AbSyn.LValue(LVal, Pos). However this is not possible, because we call another production which only returns an LVal without a position, therefore we just pass (0,0) on.

2.2.3 Parameters and Procedures

At the end we have our simple definitions for parameters and declarations.

```
PDecl :
    Params                              { $1 }
  |                                     { [] }
;

Params :
    Params TSemi Dec                    { $1 @ [$3] }
  | Dec                                 { [$1] }
;

Dec :
    TId TColon Type                     { AbSyn.Dec (#1 $1, $3, #2 $1) }
;

Decs :
    Decs Dec TSemi                      { $1 @ [$2] }
  | Dec TSemi                           { [$1] }
;

Type :
    TInt                                { AbSyn.Int ($1) }
  | TChar                               { AbSyn.Char ($1) }
  | TBool                               { AbSyn.Bool ($1) }
  | TArray TOf Type                     { AbSyn.Array ($3,$1) }
;

/* FUNCTION AND PROCEDURE PARAMETERS AND INDEX LISTS */
CallParams :
    Exps                                { $1 }
  |                                     { [] }
;

Exps :
    Exp TComma Exps                     { $1 :: $3 }
  | Exp                                 { [$1] }

%%
```

Figure 6: Parser.grm

2.3 Driver.sml

In the driver file we have uncommented the two lines which was meant for the LL1 parser and instead inserted the appropriate ones for our parser (line 57 and 69). Furthermore we have uncommented the Parser.ParseError, because this error message will be caught by the Parsing.ParseError, and instead only created compile errors. These changed are shown in figure 7.

2.4 Lexer.lex

In the *Lexer.lex* we have changed all instances of LL1parser to Parser, thereby including our newly created Parser.grm instead of the handout. Please note that

```
fun compile arg path =
  let
    val inpath = path
    val outpath= Path.base path ^ ".asm"
    val lexbuf = createLexerStream (BasicIO.open_in inpath)
  in
    let
      (*val pgm = LL1Parser.parse Lexer.Token lexbuf*)
      (* COMMENT LINE ABOVE AND UNCOMMENT  *)
      (* THE LINE BELOW TO USE YOUR PARSER *)
      val pgm = Parser.Program Lexer.Token lexbuf
    in case arg of
      "-ti" => typedInterpret (typeCheck pgm)
    | "-c"  => compileNormal pgm outpath
    | other => print ("'" ^ other ^ "': Unknown mode of operation.\n")
    end
    handle
      Parsing.yyexit ob => errorMsg "Parser-exit\n"
    | Parsing.ParseError ob =>
       (* errorMsgAt "Parsing error" (Lexer.getPos lexbuf) *)
        (* COMMENT LINE ABOVE AND UNCOMMENT  *)
        (* THE LINE BELOW TO USE YOUR PARSER *)
         errorMsgAt "Parsing error" (Lexer.getPos lexbuf)

  (* | Parser.ParseError s =>
       errorMsgAt ("Parse error: " ^ s) (Lexer.getPos lexbuf) *)

    | Lexer.LexicalError (mess, pos) =>
        errorMsgAt ("Lexing error: "  ^ mess) pos
```

Figure 7: Driver.sml

this code is left out.

2.5 Functionality of multiplication

As task 2 stated we were to implement the functionality of multiplication, division, OR and NOT in the parser. These operations have not been implemented in the rest of the compiler, so in order to test our implementation of arithmetic precedence, we implemented multiplication. This included changes to the following files: *Compiler.sml*, *TpInterpret.sml*, *Type.sml*, *TpAbSyn.sml*.

```
    | compileExp( vtable, Times (e1, e2, _), place ) =
        let val t1 = "times1_" ^ newName()
            val c1 = compileExp(vtable, e1, t1)
            val t2 = "times2_" ^ newName()
            val c2 = compileExp(vtable, e2, t2)
        in c1 @ c2 @ [Mips.MUL (place,t1,t2)]
        end

    (* Task 2: Some code-generation of operators should occur here. *)

(*    | compileExp( vtable, Times(e1, e2, pos), place ) =
        raise Error ( "Task 2 not implemented yet in code generator ", pos ) *)
    | compileExp( vtable, Div(e1, e2, pos), place ) =
        raise Error ( "Task 2 not implemented yet in code generator ", pos )
```

Figure 8: Compile.sml

```
      | LValue   of LVAL              * Pos
      | Plus     of Exp * Exp         * Pos      (* e.g., x + 3 *)
      | Minus    of Exp * Exp         * Pos      (* e.g., x - 3 *)
      | Times    of Exp * Exp         * Pos      (* e.g., x * 3 *)
  (*  | Div      of Exp * Exp         * Pos      (* e.g., x / 3 *)   *)
      | Equal    of Exp * Exp         * Pos      (* e.g., x = 3 *)
```

Figure 9: TpAbSyn.sml

```
    | pp_exp (And   (e1, e2, _))    = "( " ^ pp_exp e1 ^ " & " ^ pp_exp e2 ^ " )"
    | pp_exp (Times (e1, e2, _))    = "( " ^ pp_exp e1 ^ " * " ^ pp_exp e2 ^ " )"
```

Figure 10: TpAbSyn.sml

```
    | typeOfExp ( And     (_,_,_) ) = BType Bool
    | typeOfExp ( Times   (a,b,_) ) = typeOfExp a
```

Figure 11: TpAbSyn.sml

```
    | posOfExp  ( Map     (_,_,p) ) = p
    | posOfExp  ( Times   (_,_,p) ) = p
```

Figure 12: TpAbSyn.sml

```
  | evalExp ( Times(e1, e2, pos), vtab, ftab ) =
        let val res1   = evalExp(e1, vtab, ftab)
            val res2   = evalExp(e2, vtab, ftab)
        in  evalBinop(op -, res1, res2, pos)
        end
```

Figure 13: TpInterpret.sml

```
        (* Must be modified to complete task 3 *)
      | typeCheckExp( vtab, AbSyn.Times (e1, e2, pos), _ ) =
          let val e1_new = typeCheckExp(vtab, e1, UnknownType )
              val e2_new = typeCheckExp(vtab, e2, UnknownType )
              val (tp1, tp2) = (typeOfExp e1_new, typeOfExp e2_new)
          in  if  typesEqual(BType Int, tp1) andalso typesEqual(BType Int, tp2)
              then Times(e1_new, e2_new, pos)
              else raise Error("in type check minus exp, one argument is not of int type "^
                               pp_type tp1^" and "^pp_type tp2^" at ", pos)
          end

    (* Task 2 and 3: Some type-checking of operators should occur here. *)
   (* | typeCheckExp ( vtab, AbSyn.Times (_, _, pos), _ ) =
        raise Error ( "Task 2 not implemented yet in type-checker ", pos ) *)
    | typeCheckExp ( vtab, AbSyn.Div   (_, _, pos), _ ) =
        raise Error ( "Task 2 not implemented yet in type-checker ", pos )

```

Figure 14: Type.sml

3 Testing

We have made two test examples; one that covers the problem with nested *if then else* and another one that tests precedence with multiplication and addition. We ran the compiled programs in Mars and they returned the expected results ("If-then-else virker!!!" and "det virker jo").

```
program test;

procedure main()
var n : int;
    m : int;
    d : int;
    o : int;
begin
    n := 1;
    m := 2;
    d := 3;
    o := 4;
    if n = 1 then
        if d = 1 then
            write("Den var sand")
        else
            write("If-then-else virker!!!");
    end;
```

```
program test2;

function test() : bool
var m : int;
    o : int;
    v : int;
    p : int;
begin
    m := 2;
    o := 3;
    v := (4+m)*6100;
    p := 1+1*o*2;
    return v = 36600 and p = 7;
end;

procedure main()
begin
    if test() then
        write("det virker jo")
    else
        write("fail");
end;

```

Figure 15: Test.pal og Test2.pal