dOvs - Parsing

Group 9
Miran Hasanagić - 20084902
Jakob Graugaard Laursen - 20093220
Steven Astrup Sørensen - 201206081

September 23, 2015

1 Introduction

This report describes our approach to develop the parser, which is the first component in the compilation process. This parser is implemented using the ML-Yacc, as it is suggested by the book and the project description.

The first three section describe different parts of the parser development. First, important parts of the tiger grammar are introduced with respect to the ML-Yacc. Secondly, it is described how the semantics actions are used to construct the Abstract Syntax Tree (AST) for the program. Then it is shown how conflicts where detected, and what was done in order to resolve them.

Following these sections an overview of the problems encountered and experience gain during the work process is provided. Finally, five tiger programs are provided as test cases for different aspects of the developed parser.

2 The tiger grammar

The tiger grammar was implemented in the file `tiger.grm`. The parser development was divided into two separate parts; first the tiger grammars was implemented, and next the semantics action where added. These two parts reflect the two *program* section for the chapters three and four, respectively. Using this approach it was possible to divide the problem in two parts. First it could be ensured that we did not have any conflicts in the grammars, and afterwards the semantics actions could be implemented in order to create the AST.

In order to specify the grammar and rules Appendix A was used. However, using this grammar directly provided some conflicts and problem, which is discussed below.

3 The abstract syntax

The abstract syntax was provided with the handout file `absyn.sml`. This file provided the types of the different nodes that can be used when creating an AST from a tiger program. Hence the key work was to use this provided structure for the abstract syntax, when creating the semantic action for the parser using ML-YACC. Also it shall be noted that both the function declaration data and type declaration were made as list inside `absyn.sml`. The reason was in order to sort mutual recursive functions and type declaration, which is allowed in tiger.

4 Conflict management

4.1 lvalue

lvalue was the primary example of association and gramma not always matching, as the following code:

```
lvalue : ID (...)
           | subscript (...)
           | fieldExp (...)

subscript : lvalue LBRACK exp RBRACK (...)

fieldExp : lvalvue DOT ID (...)
```

This however produces a shift/reduce error, so we will need to fix it by making the lvalue nonterminal right associative. And we will do this by introducing a tail.

```
lvalue : ID ltail (...)

ltail : subscript ltail (subscript::ltail)
      | fieldexp ltail (fieldexp::ltail)
      | ([])
```

this will work, as we're merely now able to create the same lvalue, just from another direction, now going from left to right, rather than from right to left. As an example, I will make the string ID, sub, sub, field, sub. first using the first implementation of lvalue, and the second time with the updated version with a tail.

```
lvalue -> subscript  = lvalue2 subscript
lvalue2 -> fieldExp  = lvalue3 fieldExp subscript
lvalue3 -> subscript = lvalue4 subscript fieldExp subscript
lvalue4 -> subscript = lvalue5 subscript subscript fieldExp subscript
lvalue5 -> ID        = ID subscript subscript fieldExp subscript
```

And now with the updates Version

```
lvalue -> ID tail1
tail1  -> subscript = ID subscript tail2
tail2  -> subscript = ID subscript subscript tail3
tail3  -> fieldExp  = ID subscript subscript fieldExp tail4
tail4  -> subscript = ID subscript subscript fieldExp subscript tail5
tail5  -> NONE      = ID subscript subscript fieldExp subscript
```

As seen, the two final strings are the same, and doesn't provide a shift/reduce conflict. However as a final note, while we now make the string from left to right, when actually resolving the lvalue, we will still need to read it from right to left.

4.2 Mutual recursive function declarations

Mutual recursive types are implemented as they're explained in the appendix, which also proved to be quite the task, as they function in quite a specific way. Functions, Variables and Type are mutually recursive as long as they're declared by a consecutive sequence of the same kind.

To implement this we did the following: 1. Identify the kind of element that's found first in the let expression. 2. Make a list of the consecutive elements of the same kind. 3. pattern match to check whether a new kind of element is found next, if so, begin a new list with this kind. 4. Repeat 2 and 3 until no more elements left.

to implement this in our parser, we did the following code:

TODO: Insert correct code

```
letexp : LET deccon IN seqsexps END (...) )

deccon : fundeclbegin (...)
       | vardeclbegin (...)
```

```
        | tydeclbegin (...)

fundeclbegin : fundeclist ([fundeclist])
             | fundeclist tydeclbegin (fundeclist::tydeclbegin)
             | fundeclist vardeclbegin (fundeclist::vardeclbegin)

vardeclbegin : vardec ([vardec])
             | vardec fundeclbegin (vardec::fundeclbegin)
             | vardec tydeclbegin (vardec::tydeclbegin)
             | vardec vardeclbegin(vardec::vardeclbegin)

tydeclbegin : tydeclist (tydeclist)
            | tydeclist vardeclbegin (tydeclist::vardeclbegin )
            | tydeclist fundeclbegin (tydeclist::fundeclbegin )

fundeclist : fundec ([fundec])
           | fundec fundeclist (fundec :: fundeclist)

tydeclist : tydec ([tydec])
          | tydec tydeclist (tydec :: tydeclist)
```

Note however that vardec isn't made into a list, this is because value doesn't follow the same rules as Function and Types, as also seen in the absyn.sml file. As vardecldata is not a list.

```
decl     = FunctionDec of fundecldata list
             | VarDec of vardecldata
             | TypeDec of tydecldata list
```

4.3 If-Else shift/conflict

It can also be noted that the tiger grammar will have a dangling else conflict. However, this conflict was already solved in the provided code by making else right associative, which makes it use shift instead of reduce. Another solution can be found in the book page 67, in which an auxiliary nonterminal can be used in order to remove this conflict.

5 Problems encountered & experience gained

5.1 Solving conflicts

A key problem encountered was the implementation of tiger grammar, when a shift/reduce conflict was reported. In this part experience was gain in understanding the tiger grammar better, and understanting how to rewrite the part of the `tiger.grm` file.

5.2 Creating the abstract syntax

Another main problem encountered was the notion of mutual recursive functions and type declarations. This was not part of the grammar defintion, but still had to be addressed, when creation the specification when using ML-Yacc to create the parser. As explained above this was solved by introducing an intermediate representation in order to being able to sort the relevant mutual recursive declarations. Tiger does in have a specific construct for creating mutual recursive function declaration, as can be achieved in ML using the `and` keyword. From Appendix A in the book, the following is stated about mutual recursive functions. " Mutually recursive functions and procedures are declared by a sequence of consecutive function declarations (with no intervening type or variable declarations.)

5.3 Debugging

Quite some time was also used on debugging the code, especially when an operator was used with the wrong input. A key problem was that some errors where reported as part of the generated parser file `tiger.grm.sml`. Then we had to read this error and try to find which error in the `tiger.grm` this was related to. Also it helped that we could use the provided function `(PrintAbsyn.print (TextIO.stdOut, Parse.parse(file)))` in order to see that we create the correct tree.

5.4 Error handling

When using ML-Yacc it uses the error correction method Burke-Fisher. However, it is also possible to add some manual error correction. This we tried to do after we had created some tiger programs with syntax errors.

6 5 tiger programs

These five tiger programs presented here are created simple in order to focus on different aspects of the implemented parser. These tests contain different syntax errors as required by the project description. In order to ensure that this parser works it is run on the provided testcases by the course. Additionally, some intersting experiments where made in order to test some of the more challenging aspects of this parser. This includes creating test for the `lvalue`, test that exponent binds tighter then unary minus[1] and that the mutual recursive function are sorted correctly according to the tiger programming language.

6.1 test1.tig - Forgot to close parenthesis

```
let var h :=  (5+4
in
h end
```

This test the default error correction, and the missing parser inserts it automatically.

6.2 test2.tig - Wrong array declaration

```
let var x := aArry  of 0
in
  0
end
```

In this test case the parser will try to insert a plus according to the standard error correction.

6.3 test3.tig - Forgot end in a let exp

```
let
var a :=5
a
```

In this part the % change is tested to test that the in and end are inserted.

6.4 test4.tig - Forgot using then in a if

```
if a=0 1 else 2
```

In this case the then should be insert automatically in order to make it work.

[1]This is changed according to Aslan's new requirement

6.5 test5.tig - Using an equals instead of an assign

```
let var  b=3
 in b
end
```

In this test the parser should try to insert en assign instead of the equals according to the error correction added.

7 Conclusion

This report showed the implementation of the parser. Furthermore some intersting problem encountered where described. Finally, some tests provided confidence that this parser is working correctly and also reports correct syntax errors.