Université libre de Bruxelles

info-f-403

LLVM IR code generation for the Imp language

Stéphane Sercu

December 29th, 2017

1 Introduction

This report describes the last part of the project, which consists in building an Abstract Syntax Tree from the parser tree generated in the last part of the project and then in the generation of LLVM IR code.

The paper is structured as such: first it describes how each of the (most important) blocs of the parser tree were transformed into an AST. Then it describes how those lead to a valid LLVM IR code. Finally, it presents the new features added to the Imp language, namely, the variable scoping, the variable typing and the support of functions (including recursion).

2 Abstract Syntax Tree

First of all, the parser tree was modified to handle `Symbol` instances instead of just the name of the grammar variables/terminals (in `String` format). This makes it possible for the parser to track the syntactic/semantical errors more precisely (by keeping the line/column of the symbol as long as possible in the process).

As expected by the provided class `Symbol`, the non terminal variables were represented with a null-value for their lexical unit attribute. Their value was set to the name of the variable in the grammar. This allows to print the produced trees and helped during the debugging phases.

This allowed to build the AST from the parser tree. This was achieved by recursively browse the parser tree (from top to bottom) and remove the non-terminals and useless symbols.

2.1 Basic blocks

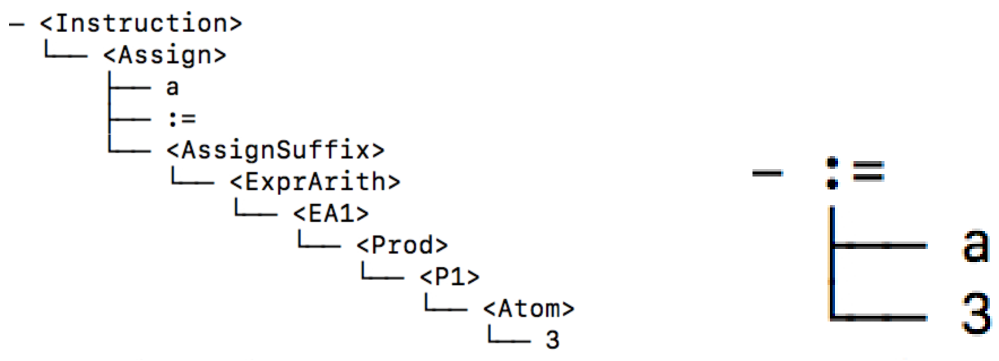
The conversion, for each of the basic blocs, is briefly described in the following subsections (and figures).

2.1.1 Assignation

The node used to mark an assignation in the AST is the `ASSIGN` symbol. It's meant to have two children: its first one is the target variable and its second one the assigned value. The later can be any valid kind of node in accordance with the grammar.
An example is presented in the figure 1.

2.1.2 Arithmetic expressions

The order of the operations is defined by their position in the AST: the deeper they are, the bigger their priority is. Indeed, the code for the bottom nodes will be generated first, and the result will be used to compute the parent operations.
As a consequence of this, the parenthesis are useless (and thus removed) in the AST.



```
─ <Instruction>
  └── <Assign>
      ├── a
      ├── :=
      └── <AssignSuffix>
          └── <ExprArith>
              └── <EA1>
                  └── <Prod>
                      └── <P1>
                          └── <Atom>
                              └── 3
```

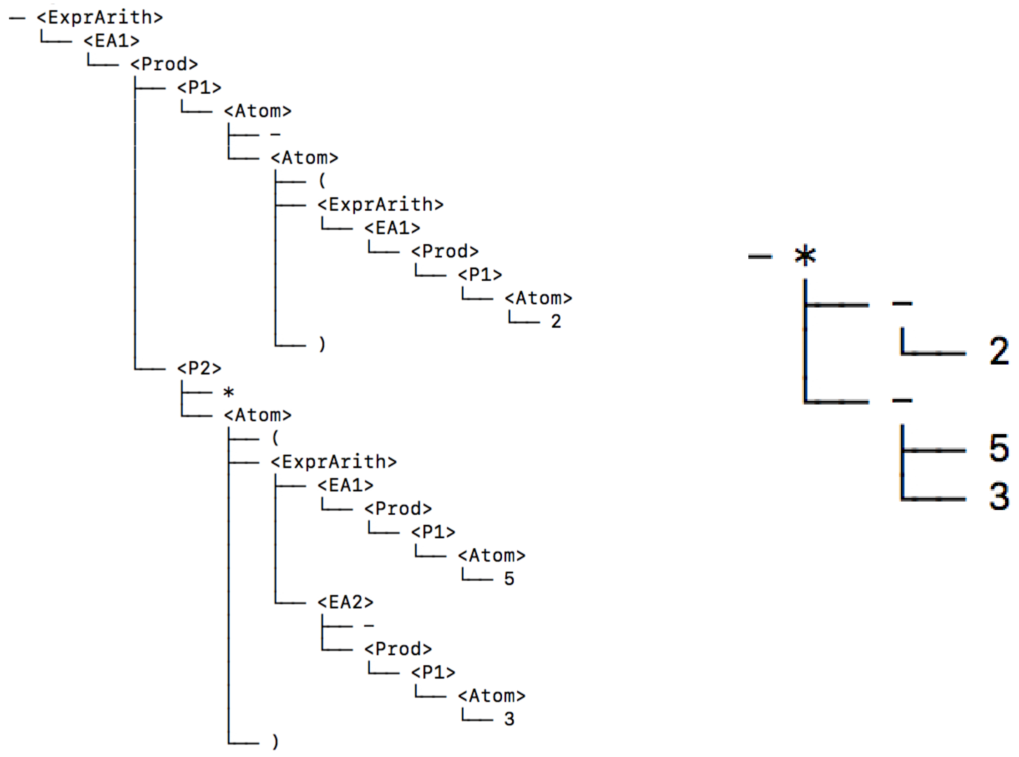
```
─ :=
  ├── a
  └── 3
```

(a) Parser Tree (b) AST

Figure 1: Assignation (`a := 3`) converted from the parser tree to the AST

The structure of an AST for an arithmetic can summarized as such: each node is either an arithmetic atom ([*VarName*] or `[Number]`) or an operation. In the later case, it has two children which can be either another arithmetic expression or an atom.
The only exception is for the **minus** operator: it can either represent the operator for a subtraction or the opposite of another expression. In the AST, its semantic is determined by the number of children it has: two for the subtraction operation, one otherwise.

The figure 2 gives an example.



```
─ <ExprArith>
  └── <EA1>
      └── <Prod>
          ├── <P1>
          │   └── <Atom>
          │       ├── -
          │       └── <Atom>
          │           ├── (
          │           ├── <ExprArith>
          │           │   └── <EA1>
          │           │       └── <Prod>
          │           │           └── <P1>
          │           │               └── <Atom>
          │           │                   └── 2
          │           └── )
          └── <P2>
              ├── *
              └── <Atom>
                  ├── (
                  ├── <ExprArith>
                  │   ├── <EA1>
                  │   │   └── <Prod>
                  │   │       └── <P1>
                  │   │           └── <Atom>
                  │   │               └── 5
                  │   └── <EA2>
                  │       ├── -
                  │       └── <Prod>
                  │           └── <P1>
                  │               └── <Atom>
                  │                   └── 3
                  └── )
```

```
─ *
  ├── -
  │   └── 2
  └── -
      ├── 5
      └── 3
```

(a) Parser Tree (b) AST

Figure 2: Arithmetic expression (converted from the parser tree to the AST

2.1.3 Boolean expressions

The AST for the boolean expressions is derived in the exact same way than for the arithmetic operations.

2.1.4 If-While-For blocs

An `if` node in the AST has 3 children: the first one contains the condition and the two others are the sub-ASTs of the instructions list in the body of the `if` and the `else`.

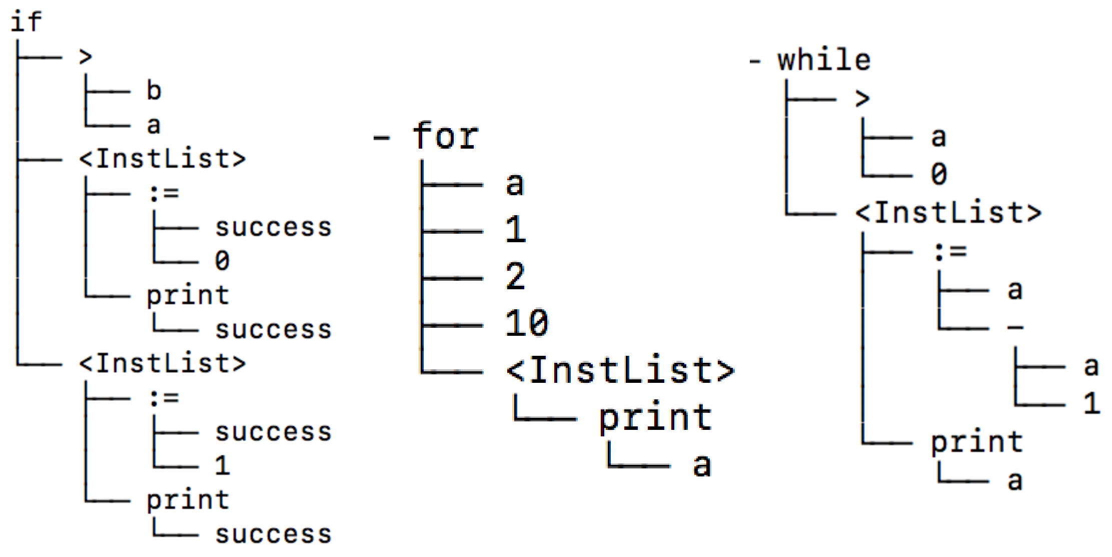
A `for` node has 5 children:

1. the counter (a [VarName])
2. the initial value (the arithmetic expression after the keyword `from`)
3. the incrementation value (the arithmetic expression after the keyword `by`)
4. the final value (the arithmetic expression after the keyword `to`)
5. the body of the loop. (an <InstList>)

If the source file doesn't specify a "`by` value", a 1 is added in the AST by default.

Finally, a `while` node has only two children: one for the condition and the other for the instructions list in its body.

The figure 3 gives an example for each of those blocs.



```
if
├── >
│   ├── b
│   └── a
├── <InstList>
│   ├── :=
│   │   ├── success
│   │   └── 0
│   └── print
│       └── success
└── <InstList>
    ├── :=
    │   ├── success
    │   └── 1
    └── print
        └── success
```

```
- for
  ├── a
  ├── 1
  ├── 2
  ├── 10
  └── <InstList>
      └── print
          └── a
```

```
- while
  ├── >
  │   ├── a
  │   └── 0
  └── <InstList>
      ├── :=
      │   ├── a
      │   └── -
      │       ├── a
      │       └── 1
      └── print
          └── a
```

(a) If:

```
if a>b then
success := 0; print(success)
else
success := 0; print(success)
endif
```

(b) For:

```
for a from 1 by 2 to 10 do
print(a)
done
```

(c) While:

```
while a > 0 do
a := a-1; print(a)
done
```

Figure 3: Flow control blocs converted from the parser tree to the AST

2.1.5 print - read

The ASTs for `print` and `read` statements are pretty obvious: they contain the `PRINT` or `READ` symbol and have one child: the variable to print the value for.
Though, the grammar for the `print` instruction has been updated to handle strings and arithmetic expressions. This is discussed in section 4.

3 Code generation

This section briefly describes how each Imp instruction is translated from the AST to LLVM IR code.

3.1 LLVM package

To deal with the generation of LLVM IR code, a bunch of classes were built. Those deal with the generation of well formatted LLVM IR instructions/groups/functions and handle the generation of valid (unique) variable/group names. The last part is achieved by storing counters (one for group names, another for variable names) that are incremented each time a new name is requested. Also, for the group names, a custom prefix can be added (typically `if`/`for`/`while`/`endif`/...) to make the IR code more readable.

Also, this package is widely independent from the syntax and the semantic of the Imp language. Only the class `LLVMGenerator`, which handles the conversion of the AST to LLVM IR code, is proper to the Imp language.

3.2 Basic blocks

This section describes how the sub-ASTs of each basic blocs are translated to LLVM code.
Some of the block a pretty trivial and their translation in IR code is obvious. Others though are less straightforward. Those are discussed in the following subsections.

3.2.1 Print/Read

To read on the input, the C `scanf` function is used.[1]
For the `print` statement, the `println` function used during the practical session on LLVM was reused.

Those two declaration are added in the final code only if it's necessary, i.e. only if there's a `read` or `print` statement in the Imp source code.

3.2.2 If

A `if` block (in Imp language) such as the one showed in figure 4, are implemented through 4 LLVM's basic blocks:

- The evaluation of the condition and the conditional jump to one of the two following blocks is added directly in the "current" bloc (i.e. in the block the previous instruction terminates);
- `if`: the second bloc contains the code inside the `if` block (executed if the condition is true);
- `else`: the third one is for the `else` block (only if there's a `else` statement in the source code);
- `endif`: the last basic block contains the code that follows the `if` block. The other two basic blocks (unconditionally) jump both to this basic block when all the code they contain is executed.

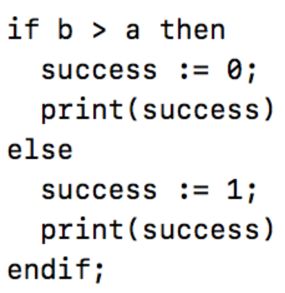
The figure 4 gives an example of translated `if` statement.

3.2.3 While

The idea behind the conversion of a `while` loop is similar to the one behind the `if` statement, with the main difference being that in the body of the `while`, instead of jumping out and continuing the execution of the program, the flow is redirected to reevaluate the condition.
In more details, the resulting LLVM IR code contains 3 basic blocks:

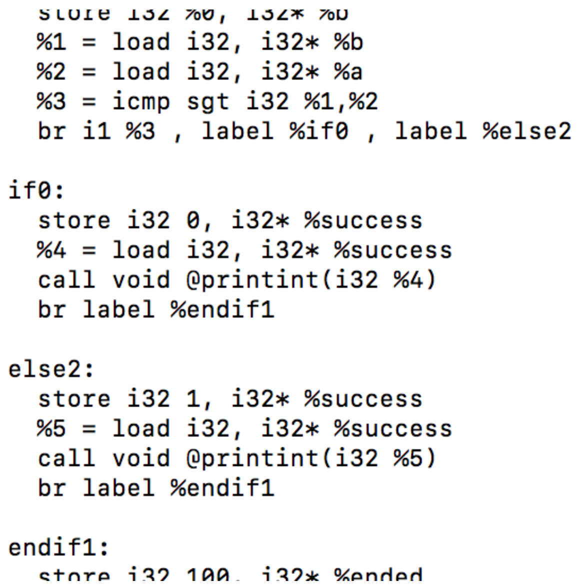
- `while`: there's one block for the content of the loop. At the end of this block, there's an unconditional jump to the `whilecond` block;
- `endwhile`: another block for the code that follow the loop;

[1] It's worth noting that to make it work on our machine, we had to change the declaration that was used during the practical session, from `@__isoc99_scanf` to `@scanf`.



```
if b > a then
  success := 0;
  print(success)
else
  success := 1;
  print(success)
endif;
```

(a) Source Imp



```
  store i32 %0, i32* %b
  %1 = load i32, i32* %b
  %2 = load i32, i32* %a
  %3 = icmp sgt i32 %1,%2
  br i1 %3 , label %if0 , label %else2

if0:
  store i32 0, i32* %success
  %4 = load i32, i32* %success
  call void @printint(i32 %4)
  br label %endif1

else2:
  store i32 1, i32* %success
  %5 = load i32, i32* %success
  call void @printint(i32 %5)
  br label %endif1

endif1:
  store i32 100, i32* %ended
```

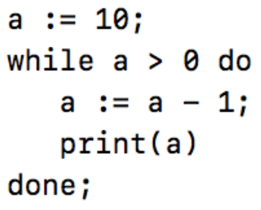
(b) LLVM IR

Figure 4: Example of LLVM IR code generated for an `if-else` statement

- `whilecond`: the last block is for the evaluation of the condition. At the end of this one, if the condition is true, the execution flow will jump in the `while` basic block. Otherwise it will leave the loop and jump in the `endwhile` basic block.

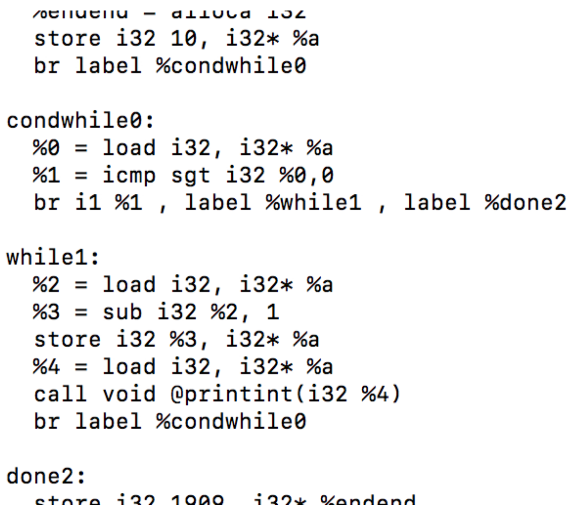
Obviously, to enter the `while`, the LLVM IR code start by jumping into the `whilecond` basic block.

An example is showed in figure 5



```
a := 10;
while a > 0 do
   a := a - 1;
   print(a)
done;
```

(a) Source Imp



```
  %endend = alloca i32
  store i32 10, i32* %a
  br label %condwhile0

condwhile0:
  %0 = load i32, i32* %a
  %1 = icmp sgt i32 %0,0
  br i1 %1 , label %while1 , label %done2

while1:
  %2 = load i32, i32* %a
  %3 = sub i32 %2, 1
  store i32 %3, i32* %a
  %4 = load i32, i32* %a
  call void @printint(i32 %4)
  br label %condwhile0

done2:
  store i32 1000, i32* %endend
```

(b) LLVM IR

Figure 5: Example of LLVM IR code generated for a `while` statement

3.2.4 For

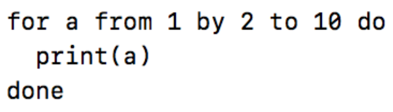
A for loop is implemented in a very similar way to the `while` loop: there are 3 basic blocks too: `for`, `endfor` and `incfor`. The first two are equivalent to their counter-sides in the `while` loop, the last one is equivalent to the `whilecond` where the condition to evaluate is the following:

```
(by>0 and a+by < to) or (by<0 and a+by>to)
```

where `a` is the incremented variable, `by` and `to` are respectively the increment and the max value. Also, the left-hand side of the `or` handles the cases where the increment (`by`) is positive, the other side is for the cases where it's negative.

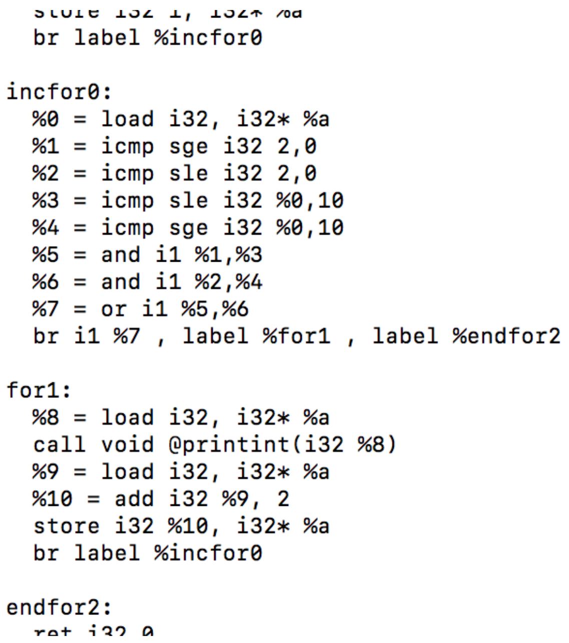
If this condition is true, then it runs the body of the loop one more time, otherwise it jumps outside the loop (in the `endfor` group).

The result is illustrated in the figure 6



```
for a from 1 by 2 to 10 do
  print(a)
done
```

(a) Source Imp



```
  store i32 1, i32* %a
  br label %incfor0

incfor0:
  %0 = load i32, i32* %a
  %1 = icmp sge i32 2,0
  %2 = icmp sle i32 2,0
  %3 = icmp sle i32 %0,10
  %4 = icmp sge i32 %0,10
  %5 = and i1 %1,%3
  %6 = and i1 %2,%4
  %7 = or i1 %5,%6
  br i1 %7 , label %for1 , label %endfor2

for1:
  %8 = load i32, i32* %a
  call void @printint(i32 %8)
  %9 = load i32, i32* %a
  %10 = add i32 %9, 2
  store i32 %10, i32* %a
  br label %incfor0

endfor2:
  ret i32 0
```

(b) LLVM IR

Figure 6: Example of LLVM IR code generated for a `for` loop

4 Amelioration

This section presents all the ameliorations that were added to the compiler presented until now.

4.1 Scoping and undeclared variable detection

The variable scoping was achieved through a tree of identifier tables and was implemented directly in the parser.

Before digging in the way the parser build the idTable tree, let's point out some assumptions:

- A new variable can only be declared in an assignation or between a `FOR` and a `FROM`[2];
- There's a limited number of symbols affecting the scope: `FOR`, `DO`, `THEN` create a child scope, `ELSE` creates a new scope on the same level as the current one, `DONE` and `ENDIF` lead back to the parent scope[3].

A consequence of the first assumption is that a variable cannot be initialized in a `read` statement anymore. This is not a problem though, since it will be useful when implementing different types: it will tell the read statement what kind of input to expect[4].

Each time it matches a new symbol, the parser does the following things:

[2]A variable used in a `read` must have been declared first.

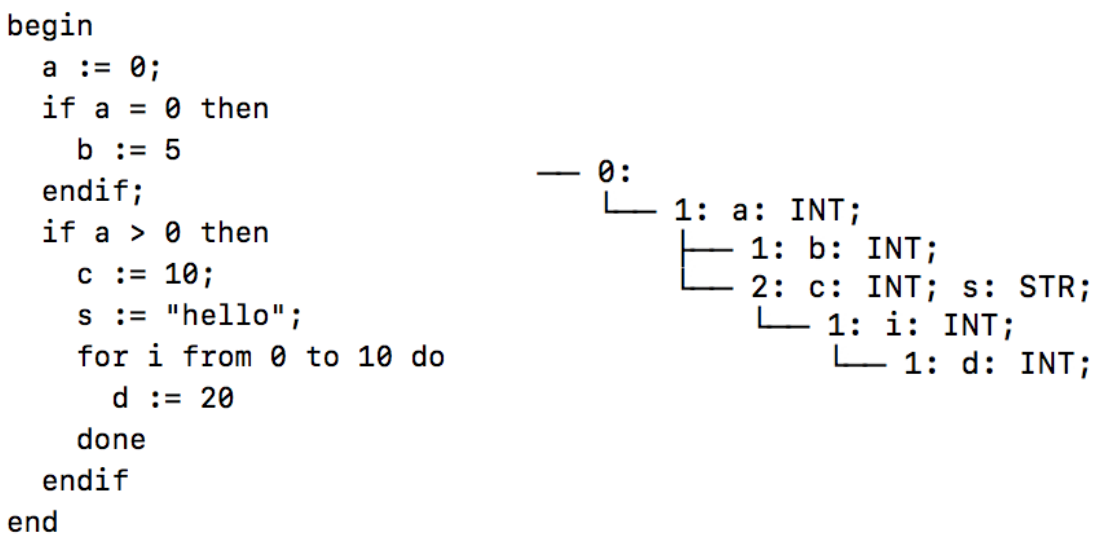
[3]Actually, to make the initialization of a new variable possible between a `FOR` and a `FROM`, a new (child) idTble is created for each matched FOR/DO and WHILE/DO and the match of a DONE leads to a double upward move in the scoping hierarchy. The new IdTable created by the matching of a while is useless though since no variable can't be declared between the `while` and the `done`.

[4]So this is not a bug, it's feature!

1. If the symbol is a `VARNAME`, it checks of the variable is declared in the current idtable or in all the parent id tables. If it's not the case, it checks if the following symbol is an `ASSIGN` or a `FROM`. If so, it declare the variable in the current scope. Otherwise, it yields an error.
2. Checks how the matched symbol affect the scope, and change the reference of the current node in the tree accordingly.

The figure 7 shows an example of scoping tree.

The figure 8 shows the implementation of the structure containing the tree. The `subtype` parameter allows the declaration of more complex structures such as functions, arrays, dictionaries, ... For example, the first subtype of a function would be the type of the value it returns and the others would be the types of its parameters. This is discussed in section 4.1.2.



```
begin
  a := 0;
  if a = 0 then
    b := 5
  endif;
  if a > 0 then
    c := 10;
    s := "hello";
    for i from 0 to 10 do
      d := 20
    done
  endif
end
```

(a) Example program

```
── 0:
   └── 1: a: INT;
       ├── 1: b: INT;
       └── 2: c: INT; s: STR;
           └── 1: i: INT;
               └── 1: d: INT;
```

(b) Resulting tree

Figure 7: Example of generated identifier table tree

Undeclared variable detection The scoping feature allows the compiler to detect if a variable is used before declaration. For example, the following program[5]:

```
begin
  a := 1;
  if a > 0 then
    b := 3
  endif;
  print(b)
end
```

will be detected as invalid and the compiler will output the following error:

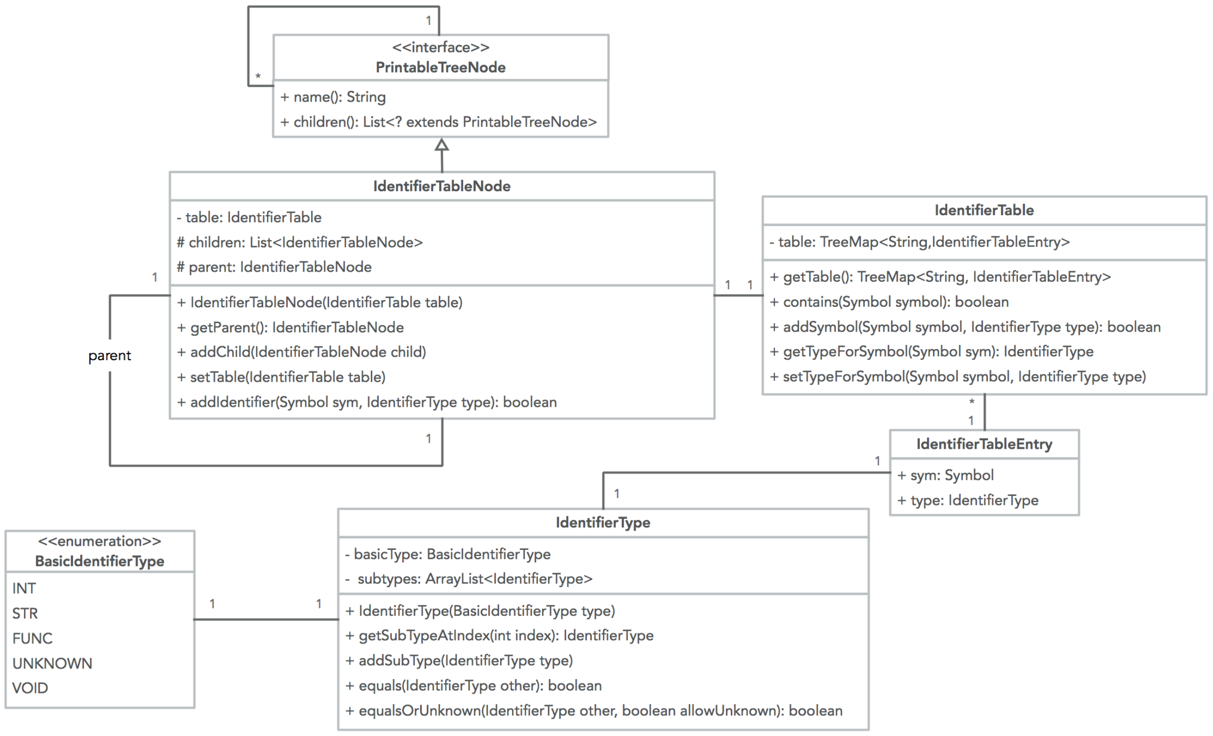
```
Error: variable b was used before initialization. Line 6 col 7
```

4.1.1 Strings and type checking

To add support for the strings, the following things were updated:

- support for new units in the class `LexicalUnit` and in the `Lexer`;
- modification of the grammar to handle strings in assignations and in `print` statements;

[5]This program is available in the test folder: `test/invalidScope.imp`



1
*
<<interface>>
PrintableTreeNode
+ name(): String
+ children(): List<? extends PrintableTreeNode>
IdentifierTableNode
- table: IdentifierTable
# children: List<IdentifierTableNode>
# parent: IdentifierTableNode
+ IdentifierTableNode(IdentifierTable table)
+ getParent(): IdentifierTableNode
+ addChild(IdentifierTableNode child)
+ setTable(IdentifierTable table)
+ addIdentifier(Symbol sym, IdentifierType type): boolean
parent
1
1
1 1
IdentifierTable
- table: TreeMap<String,IdentifierTableEntry>
+ getTable(): TreeMap<String, IdentifierTableEntry>
+ contains(Symbol symbol): boolean
+ addSymbol(Symbol symbol, IdentifierType type): boolean
+ getTypeForSymbol(Symbol sym): IdentifierType
+ setTypeForSymbol(Symbol symbol, IdentifierType type)
*
1
IdentifierTableEntry
+ sym: Symbol
+ type: IdentifierType
1
1
IdentifierType
- basicType: BasicIdentifierType
- subtypes: ArrayList<IdentifierType>
+ IdentifierType(BasicIdentifierType type)
+ getSubTypeAtIndex(int index): IdentifierType
+ addSubType(IdentifierType type)
+ equals(IdentifierType other): boolean
+ equalsOrUnknown(IdentifierType other, boolean allowUnknown): boolean
<<enumeration>>
BasicIdentifierType
INT
STR
FUNC
UNKNOWN
VOID
1
1

Figure 8: Class diagram for the scoping feature

- use of the new structure for the identifier table to differentiate `STR` from `INT` types;
- adaptation of the AST and the LLVM generator to handle strings.

Lexer To recognize strings, the lexer detects the opening/closing quotes `"` and jumps into a *STRING* state which handles the escaped characters (such as `"\\""`, or `"\\n"`). The whole process generates a lexical unit `STRING` which was added in the class `LexicalUnit`.

Grammar The modifications in the grammar to support the string are marked with the blue color in section .

By the same occasion, the grammar of the `print` statement has been updated to allow the print of <ExprArith> (which includes [VarName]) and of [String]. This was done with the inclusion of the new variable <PrintSuffix>.

LLVM To facilitate the manipulation of strings in the LLVM code, all the string literals are allocated and intermediary LLVM variables are initialized in the beginning of the program. After that, each time they are used in the code (in an assignation or in a `print` statement) they are loaded in the right (local) variable.

To achieve that, the part of the compiler handling the generation of the LLVM code for a `print` statement checks the type of the variable to print (in the case the parameter of the print is a variable) and produce an LLVM IR code with the right variable types.

Other features, such as function returning other types or taking other types as argument, or type verification (detecting a misuse of a variable, such as arithmetic operations with string variable) could easily be implemented in a similar way.
The current version of the compiler doesn't perform a complete type check, but the implementation of the IdentifierTable tree would allow the compiler to traverse the AST and deduce the type of the remaining undetermined variable and check for ambiguities.

4.1.2 Functions

Syntax The functions have been implemented to have the following syntax:

```
func funcName with var1 var2 var3 in
     <Code>
     [return <ExprArith>]
endfunc
```

Also, the syntax for calling a function is:
`[rtnValue = ] call funcName(v1 v2 v3);`

Finally, to keep the grammar as simple as possible, all the functions have to be declared before the `begin` of the program.

This lead to the modifications (marked with the red color) in the grammar presented in section A. Of course, the grammar was checked with the same methods as in Part 2 to make sure it's still an LL(1) grammar.

Changes in the scoping The scope gesture in a function is very similar to the one for the `for` loops.

The lexical unit `FUNC` is recognized as one that adds a level of scope, and the `ENDFUNC` one is recognized as one that goes one level back.

Also, it's now possible for variables to be declared before another variable[6] or before the `in` keyword (i.e. in the declaration of a function).

It's also worth noting that the scope gesture presented earlier makes it easy to check for duplicate parameters in a function declaration. For example, a function declared as[7]

```
func invalidFunc with var1 var1 var2 in
```

will yield to the following error in the compiler:

```
Duplicate variable name "var1".  Line 1 col 22.
```

Changes in the typing The name of a function is managed as a [VarName] and thus added in the identifier tables tree. The type that's attributed to it is `FUNC` and its subtypes are made of the type it returns and the types of its arguments.

Currently, and according to the grammar, only the type `INT` is supported as return type and argument type.

LLVM implementation For each Imp function, an LLVM function is generated with the same name, same return type (currently `i32` by default) and same parameters.

4.2 Executable

Finally, three optional arguments were added to the executable

- a path to an output file were the generated LLVM code will be written
- `--exec` will make the compile to run the generate LLVM code (as long as the `lli` command is defined on the host system)[8]

[6]this is ok since the only syntactical valid place where two variables are used the one in front of the other, is in a function declaration (... and in a function call, which means that undeclared variables in a function call will not be detected as invalid. This will be fixed in the fourth part of the project! ... oh wait..)

[7]such a function is available in the file `test/func/invalid/duplicate_arg.imp`

[8]note that the `read` statement doesn't work well if the program is executed like that.

- `--debug` will print the parser tree, the identifier tables (in a tree form) and the AST, as they are generated.

The final usage can be summarized by:

```
java -jar part3.jar file.imp [out.ll] [--debug] [--exec]
```

5 Tests

The compiler was tested through a wide collection of code examples that cover most of the possible (valid or wrong) cases. Those files are contained in the `test` folder and are sorted according to the feature they test and according to whether they implement valid use cases or invalid ones.

Additionally, some real programs/algorithms were implemented. For example

- `gcd.imp` implements a recursive algorithm to calculate the gcd of two integers;
- the folder `eulerproject` contains the implementation of the solutions of the 10 first problems for eulerproject
- `recursiveFactorial.imp` and `iterativeFactorial.imp` calculate the factorial of a number, each one with a different algorithm
- `isPrime.imp` implements and tests a function that tells if a number is a prime
- `guessingGame.imp` implements a version of the famous game;

A Modified grammar

(1)	<Func>	→ `func` [VarName] `with` <FuncSuffix>
(2)	<FuncSuffix>	→ `in` <Code> <ReturnStmt>
(3)		→ [VarName] <FuncSuffix>
(4)	<ReturnStmt>	→ `return` <ExprArith> `endfunc`
(5)		→ `endfunc`
(6)	<Program>	→ `begin` <Code> `end`
(7)	<Code>	→ ϵ
(8)		→ <InstList>
(9)	<InstList>	→ <Instruction> <InstListSuffix>
(10)	<InstListSuffix>	→ ϵ
(11)		→ `;` <InstList>
(12)	<Instruction>	→ <Assign>
(13)		→ <If>
(14)		→ <While>
(15)		→ <For>
(16)		→ <Print>
(17)		→ <Read>
(18)		→ <FuncCall>
(19)	<FuncCall>	→ `call` [VarName] `(` <FuncCallSuffix>
(20)	<FuncCallSuffix>	→ <ExprArithm> <FuncCallSuffix>
(21)		→ `)`
(22)	<Assign>	→ [VarName] `:=` <AssignSuffix>
(23)	<AssignSuffix>	→ ExprArith
(24)		→ [String]
(25)		→ <FuncCall>
(26)	<ExprArith>	→ <EA1><EA2>
(27)	<EA1>	→ <Prod>
(28)	<EA2>	→ `+` <Prod><EA2>
(29)		→ `-` <Prod><EA2>
(30)		→ ϵ
(31)	<Prod>	→ <P1><P2>
(32)	<P1>	→ <Atom>
(33)	<P2>	→ `*` <Atom><P2>
(34)		→ `/` <Atom><P2>
(35)		→ ϵ
(36)	<Atom>	→ [VarName]
(37)		→ [Number]
(38)		→ `-`<Atom>
(39)		→ `(` <ExprArith> `)`
(40)	<Cond>	→ <C1><C2>
(41)	<C1>	→ <BoolAnd>
(42)	<C2>	→ `or` <BoolAnd><C2>
(43)		→ ϵ
(44)	<BoolAnd>	→ <BA1><BA2>
(45)	<BA1>	→ <BoolAtom>
(46)	<BA2>	→ `and` <BoolAtom><BA2>
(47)		→ ϵ
(48)	<BoolAtom>	→ `not` <BoolAtom>
(49)		→ <ExprArith> <Comp> <ExprArith>
(50)	<If>	→ `if` <Cond> `then` <Code> <IfSuffix>
(51)	<IfSuffix>	→ `endif`
(52)		→ `else` <Code> `endif`

Table 1: Final grammar

(53)	<Comp>	→ `=`
(54)		→ `>=`
(55)		→ `>`
(56)		→ `<=`
(57)		→ `<`
(58)		→ `<>`
(59)	<While>	→ `while` <Cond> `do` <Code> `done`
(60)	<For>	→ `for` [VarName] `from` <ExprArith> <ForSuffix>
(61)	<ForSuffix>	→ `by` <ExprArith> `to` <ExprArith> `do` <Code> `done`
(62)		→ `to` <ExprArith> `do` <Code> `done`
(63)	<Print>	→ `print(` <PrintSuffix>
(64)	<PrintSuffix>	→ <ExprArith>`)`
(65)		→ [String]`)`
(66)	<Read>	→ `read(`[VarName]`)`