UNIVERSITÉ LIBRE DE BRUXELLES

DÉPARTMENT D'INFORMATIQUE

INFO-F403
INTRODUCTION TO LANGUAGE THEORY AND COMPILING

Project Report – Part 3

Author:
Hakim BOULAHYA

December 29, 2017

Contents

1 Introduction

As part of the course of *Introduction to Language theory and Compiling* we were asked to write a compiler of the IMP language. The goal of the third part of the project was to implement the *code generatar*.

2 Parser Improvements

To make the code generation easier to implement, the parser was modified to produce an AST (abstract syntax tree).

To do so, some modification were make on the parser of part 2. The derivation tree is constructed during the parsing *i.e.* using the stack implementation. The AST a post-process function, that modify the derivation tree. The method `LL1Parser.buildAST()` simplifies the derivation tree.

To simplify the derivation tree to an AST, *simplifying* methods has been implemented in `LL1Parser`. The resulitng format of the AST are shown in the Figures below. The trees must be read from to bottom *i.e.* the topmost node is considered as the leftmost node.

```
begin
    print(x);
    x := 1
end
```

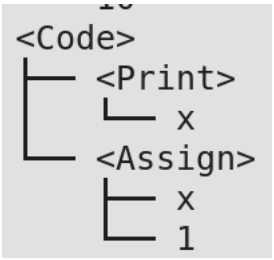

Figure 1: Code AST: the format is a node <Code> with children that are instructions

```
x := (1 + 2) * 5 / 1;
```

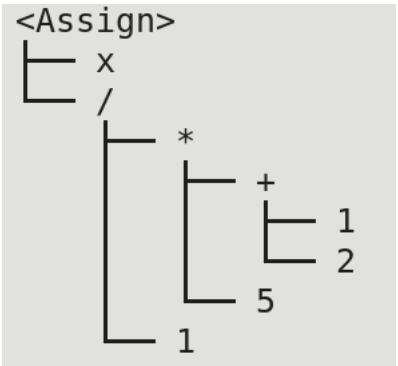

Figure 2: Assign AST: The first child is the var name, and the second child an arithmetic expression.

```
x := -(2 - 1);
```

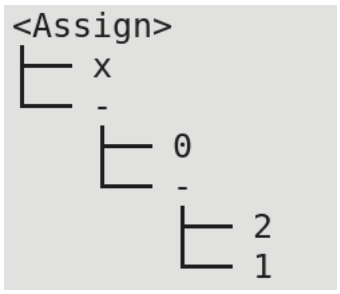

Figure 3: Negative arithmetic expressions: a minus in front of an expressions will be considered as a subtraction between 0 and expression.

```
print(x)
read(c)
```

Figure 4: Print/Read AST: the variable name is the only child of those expression.

```
for i from 0 by 1 to 10 do
    x := x + 2
done
```

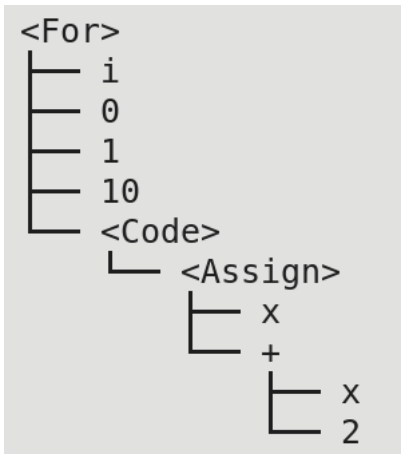

Figure 5: For AST: The first child is varname. The second child the initial value, third child the step (when not specified, it is 1), the fourth child is the maximum value. The last child is the code AST to execute if the condition are respected.

```
while i <= 10 do
    print(x);
    x := 1
done
```

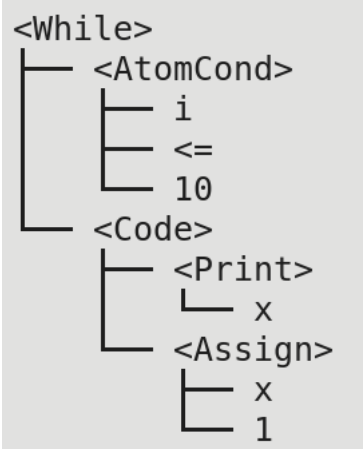

Figure 6: While AST: The first child is the condition and the second child is the code AST to execute if the condition is true.

```
if x <> 1 and x < 10
   or z = 1 then

    print(x)
else
    print(z)
endif;
```

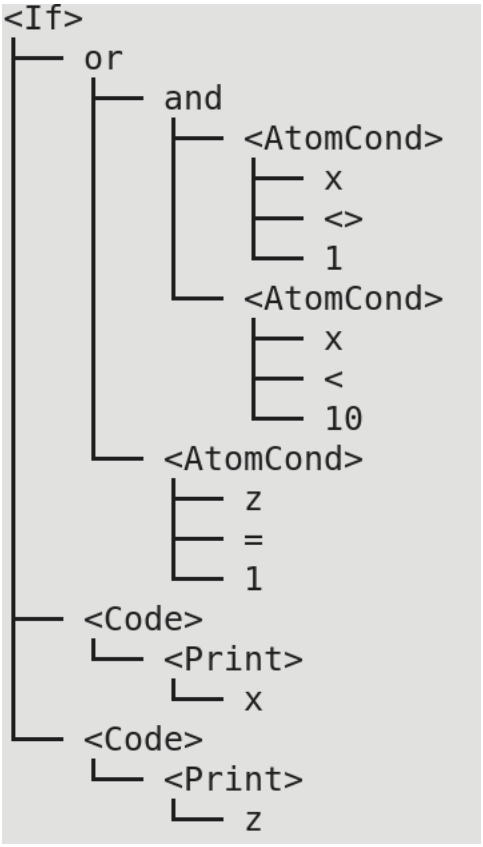

Figure 7: If AST: The first child is the condition. The condition as follow the same simplification as the arithmetic expression, so that the priority of `or` and `and` are respected. The two last child are: the code AST to execute if the condition is true, and the code to execute if the code is false (*i.e.* the else block)

.

3 Code generation

The code generation consists in transforming the resulting AST into LLVM IR code. The implementation of the code generation is in the classe `CodeGenerator`. The initial AST is a code node *i.e.* an AST with children that represent each an instruction. Each type of instruction are converted into specific instructions. Those following decisions were made when generating the LLVM IR:

- There is only one scope (since there is no function)
- An <If> with two empty blocks will not generated any LLVM IR code
- The `not` condition instructions has been implemented in the AST directly. This means that the booleon operation is switched if the `not` keyword is used. Example: not a <> b will create an instruction $a = b$.

- The memory allocation of a variable is called only once. It is allocated when the variable is first encountered in the AST. This means that the if, while, for blocks uses the same scope *i.e.* assigning a variable inside a if will be accessible outside.

Figure below shows different IMP code and the corresponding generated LLVM IR:

```
begin
    x := 1;
    print(x)
end
```

```
; T(<Assign>)
%1 = add i32 0, 1
%x = alloca i32
store i32 %1, i32* %x
call void @println(i32*
    %x)
ret void
```

Figure 8: Example of basic allocation and print

```
while i <= 10 do
    print(x);
    x := 1
done
```

```
; T(<While>)loopCount=1
%1 = load i32, i32* %i
%2 = icmp sle i32 %1, 10
br i1 %2, label
    %beginLoop1, label
    %endLoop1
beginLoop1:
call void @println(i32*
    %x)
; T(<Assign>)
%3 = add i32 0, 1
%x = alloca i32
store i32 %3, i32* %x
%4 = load i32, i32* %i
%5 = icmp sle i32 %4, 10
br i1 %5, label
    %beginLoop1, label
    %endLoop1
endLoop1:
ret void
```

Figure 9: While loop example

```
for i from 0 by 1 to 10 do
    x := x + 2
done
```

```
; T(<For>)loopCount=1
%1 = add i32 0, 0
%i = alloca i32
store i32 %1, i32* %i
%2 = add i32 0, 10
%3 = load i32, i32* %i
%4 = icmp  slt i32 %3, %2
br i1 %4, label
    %beginLoop1, label
    %endLoop1
beginLoop1:
; T(<Assign>)
%5 = load i32, i32* %x
%6 = add i32 0, %5
%7 = add i32 0, 2
%8 = add i32 %6, %7
%x = alloca i32
store i32 %8, i32* %x
%9 = add i32 0, 1
%10 = load i32, i32* %i
%11 = add i32 %10, %9
store i32 %11, i32* %i
%12 = add i32 0, 10
%13 = load i32, i32* %i
%14 = icmp  slt i32 %13,
    %12
br i1 %14, label
    %beginLoop1, label
    %endLoop1
endLoop1:
ret void
```

Figure 10: For loop example. Not that the condition must be present before the beginLoop label, and in the block.

```
x := (1 + 2) * 5 / 1
```

```
%1 = add i32 0, 1
%2 = add i32 0, 2
%3 = add i32 %1, %2
%4 = add i32 0, 5
%5 = mul i32 %3, %4
%6 = add i32 0, 1
%7 = sdiv i32 %5, %6
%x = alloca i32
store i32 %7, i32* %x
ret void
```

Figure 11: Example of an arithmetic assignment

```
if x <> 1 and x < 10
   or z = 1 then
    print(x)
else
    print(z)
endif;
```

```
; T(<If>)ifCount=1
%1 = load i32, i32* %x
%2 = icmp ne i32 %1, 1
%3 = load i32, i32* %x
%4 = icmp slt i32 %3, 10
%5 = add i1 %2, %4
%6 = icmp  eq i1 %5, 2
%7 = load i32, i32* %z
%8 = icmp eq i32 %7, 1
%9 = add i1 %6, %8
%10 = icmp  uge i1 %9, 1
br i1 %10, label
   %iftrue1, label
   %iffalse1
iftrue1:
call void @println(i32*
   %x)
br label %ifcontinue1
iffalse1:
call void @println(i32*
   %z)
br label %ifcontinue1
ifcontinue1:
ret void
```

Figure 12: If example. Note that the jump to the ifcontinue label must also be done in the else scope *i.e.* in the iffalse label.