Amy Compiler Extension - Imperative features

CS320 - Final Report

Stephane Selim Diana Petrescu
EPFL
{firtname.lastname}@epfl.ch

1. Introduction

Initially, Amy was a simple functional Scala-like language. In this paper, we present a compiler extension for the Amy project. This extension aims to extend the Amy current language with imperative features. More precisely, we implemented mutable local variables and while loops.

However, before extending the Amy language, we first had to implement a basic compiler that would take (functional) Amy source files as input and produce WebAssembly bytecode files as output. In order to achieve this, there were several steps that we followed. The first one was the Lexical Analysis: we needed to convert the input text into a list of tokens. Now that we had our tokens, we had to parse them into an Abstract Syntax Tree. We paid attention to write a grammar for the parser that was LL(1) as it could be parsed in linear time (whereas the CYK algorithm for example is in O(n^3)). Once we had translated our source programs into Abstract Syntax Tree, the last step was the code generation to transform the ASTs into WebAssembly. However, before generating code for the Amy programs (or interpreting them), we also added some checking steps to verify that the programs were correct or to throw an error if they weren't. Therefore, we implemented a name analyzer and a type checker.

2. Examples

In this section, we present some examples of our extension. We added useful imperative functionality to Amy and here is how they can be applied.

In imperative programming, the use of "real" variables (whose value may change during execution) allows the program to characterize the state of a system at a given moment but also its evolution over time. We present here a simple example showing how to declare and initialize variables and how to use them. Note that in our implementation we don't allow uninitialized variables, for the purpose of bug-free and safe programming.

```
var j : Int = 0; // declare and initialize a variable j
var i : Int = 3;
i = j;
j = i + 1; //reassign
j += 1; // reassignment operators
Std.printInt(i); // prints 0
Std.printInt(j) // prints 2
```

Loops are a powerful concept widely used in imperative programming. It is a control flow statement that allows the same code to be executed as many times as needed, until a certain condition is reached. In this project, we decided to implement the while loop as it is quite intuitive and can later be used to implement other loops if desired. Furthermore, this while loop itself is a direct illustration of the use of the variables presented above. In the following example, we can see the syntax for the while loop.

```
def fact(n: Int): Int = {
  var res: Int = 1;
  var j: Int = n;
  while(1 < j) {
    res = res * j;
    j = j - 1
  };
  res
}
```

3. Implementation

Here we outline the thought process we've taken to implement these imperative features.

3.1 Theoretical Background

There's not much to say in the matter of theory behind the imperative features, save for a few things to note. As already mentioned in the introduction, choosing to extend our current language with imperative features such as mutable variables and while loops, renders our language no longer purely functional. That means that our extended language introduces side effects, hence our choice of leaving the type of such expressions (variables and while loops) as Unit type in our implementation.

Second, since Amy is a simplistic language and doesn't support nesting functions nor higher order functions, we just need to talk about the fact that the variables are lexically scoped. Furthermore, seeing how Amy is syntactically designed to have all the function definitions come first (as we cannot declare variables or any other statements before the function definitions), the lifetime of any variable declared inside a function block extends only to this block, and then the scoping of the variables declared in the after expression is pretty straightforward.

Moreover, uninitialized variables are not allowed in our language. Every variable must be initialized with a value upon declaration, not just for the purpose of the ease of the implementation, but to prevent further headaches regarding potential errors in the program and to allow safe and bug-free programming, since errors related to uninitialized variables are common and could pass unseen.

Finally, the syntax and the meaning of the while loop are the ones that we can find in most imperative programming languages. More precisely, this control flow structure takes as parameter a boolean condition and repeat the executions of the body of the loop as long as the condition holds. The theoretical background used (in addition to the ones describing a while loop) was the lecture given by Professor Viktor Kuncak on the code generation for loops [Kuncak 2017]. We will give more details in the following section.

3.2 Implementation Details

We will start with the implementation of variables and then proceed to the while loops.

3.2.1 Variables

Integrating a new feature like variables requires changes through the all the phases of the compiler mentioned in the introduction.

First the lexing and parsing, which were by far the major hiccup of this implementation. In our program, we want to be able to parse variable declarations and variable reassignments:

```
var j : Int = 0; // declare and initialize a variable j
j = 1; // or j += 1
```

These are the new expressions that we should feed to our compiler parser.

The problem was to add our new features to the grammar while keeping it an (LL1) grammar and maintaining the language syntax. Normally, the grammar should be modified as follows:

```
Expr ::= Val Param = ExprMatchDef; Expr |
Var Param OptExprMatchDef; Expr |
Id = ExprMatchDef; Expr
ExprMatchDef ExprHelper
```

Since an *ExprMatchDef* can be derived as a simple id (e.g. when returning a value) or a function call (e.g. x()) which also starts with an id, we have $First(ExprMatchDef ExprHelper) \cap First(Id = ExprMatchDef; Expr) = id$. Thus the parser doesn't know which of those two to parse when it comes upon an id. So an idea was to factorize out id, but this changes several things: first the precedence of id (variables) in our grammar just moved from highest to the lowest precedence, second when constructing the AST of a function call, we would have to go through all the tree to fetch the arguments of the function from the leaves of the tree, which makes it really tedious when constructing the AST.

Different solutions were tried out to be able to parse these expressions, that ended up changing the language syntax or introducing side effects and other language features, until we settled on the current one, which is to tackle this problem in the lexer phase. Until now, we only had to introduce one new token for the lexer which is *VAR()*. We also introduce another new token *VARIDREASSIGN()* which is "id" and "=" flattened together into a single token (the same with *VARPLUSEQUALS()* and other reassignment operators). This allows us to correctly parse variable reassignments while preventing a first conflict in the grammar, since technically now a variable reassignment doesn't start with

an Id anymore, but with an *ID Equalsign* token. This is not the cleanest solution as ideally, id and equals sign are essentially different tokens. The lexer should have no knowledge of the language grammar since it is the phase before the parser, and here the lexer is doing more than what it should do.

Further problems arose with this particular implementation of the lexer, as now some tokens may be mistaken for others. In the following example:

```
def doSomething(a: Int) : List = {
        //do Something
}
var s : List = doSomething(0);
```

The return type of the function $List =$ will be mistaken as the start of a variable reassignment, since it follows the definition of the token *VARIDREASSIGN()*. The same for the type of the variable s. To help solve this, we now take advantage of the recursive implementation of the lexer and we add an implicit parameter *REASSIGNPOSSIBLE* that acts as flag signaling when a variable reassignment is not expected. It doesn't mean that if this value is true then a variable reassignment is expected, it merely indicates that it is possible. That way in our example, after done lexing the colon and producing a token for it, we set the flag to false, indicating that a variable reassignment is impossible to occur at this point, and hence no *VARIDREASSIGN()* will be produced, instead a simple id token indicating the return type of the function will be produced. Again, this shows that the lexer requires a prior knowledge of the language grammar, which is not ideal.

After being done with the lexing, the parser should now be straightforward. The variable reassignment expressions, as well as the reassignment operators now share the same precedence as the variable declarations and the grammar is LL1. One thing to note, is that the new reassignment tokens are implemented as sentinel tokens since they hold a value, in this case the name of the variable that's being reassigned.

Now that we're done with the lexing and the parsing, comes the Name analysis. This is an important phase of the compilation of the variables since it will help us prevent reassigning values to immutable variables. For this, we modified in the AST the tree node *Let(df, value, body, variable)* which is a variale declaration, so that it contains an extra parameter, the boolean *variable* that indicates whether it's a mutable variable or not, and which is set when parsing a *val* or a *var* accordingly. So now the map that contains the local variables of the program in the name analyzer, maps them not only to their unique identifiers, but to a boolean flag that indicates whether this variable is mutable or not. That way, when we come upon a variable reassignment, we first check in the locals map if it's mutable or not, and send an error if it's not.

For Type checking, the main goal is to type check variable reassignments. We have to check that the value that's being reassigned to the variable typechecks to this variable's type. Gladly, we have a map that maps variables to their respective types, so we can use it to retrieve the type of the variable that's being reassigned and use it for the typecheck. And as mentioned before, variable reassignments are given *UnitType*.

Finally comes the code generation, and there's nothing much to say here. Variable are saved in memory and when reassigned, we only have to look up their indices and change their values using *SetLocal(index)*.

3.2.2 While loops

As for the variables, integrating while loops requires changes through all the phases of the compiler mentioned earlier.

For the lexer, we first had to add a new Token *While()* to represent the keyword while.

For the parsing, we had to do two things. The first one was to introduce the while expression *WhileExpr* as base expression in our precedence model, meaning that it has the same priority as for example literals or error expression. The second one was to change how the expression *Expr* mentioned in the previous subsection was defined, in order to allow while loops (for example) to finish with a reassignment, as most of the time they finish by $i+ = 1$ or a similar incrementation.

For the type checking part, it was quite obvious that the parameter should be of boolean type. However, for the types of the body and the while expression itself we hesitated some time before making a decision. Our first choice was to make them of type *Any*. This would have meant that the while could return something and could be part of some other expression (like for example an addition). In order to make this possible, we would have had to change the usual structure and meaning of the while to make it more meaningful and comprehensible. An example is given here where we changed the usual structure of the while by appending at the end a parenthesis containing the return expression.

```
var x = 0;
val h: Int = 1 + while(x < 10){ // called for example
                 val j: Int = 1; // for a side effect; type
                 x = x + j // checks
  }(4 + 3);
Std.printInt(h) // prints 8
```

Nevertheless, we finally chose to make them of type *UnitType* because of its simplicity and comprehensibility and to avoid any possible confusion. Therefore, we decided that such an example would throw an error:

```
var x = 0;
val h: Int = 1 + while(x < 10){ // type check error
                 val j: Int = 1;
                 x = x + j
  }
```

Finally, for the code generation part, we found different ways to implement the while loop in WebAssembly. The first one was one that we implemented without having any other references:

```
val label = getFreshLabel()


Loop(label) <:> cgExpr(cond) <:> If_void <:>
cgExpr(body)<:> Br(label) <:> Else <:>
End <:> End
```

We then implemented another version inspired from the code generation for loop lecture mentioned earlier. However, it was a quite long solution:

```
val nextLabel = getFreshLabel()
val startLabel = getFreshLabel()
val bodyLabel = getFreshLabel()


Block(nextLabel) <:> Loop(startLabel) <:>
Block(bodyLabel) <:> cgExpr(cond) <:>
If_void <:> Br(bodyLabel) <:> Else <:>
Br(nextLabel) <:> End <:> End <:>
cgExpr(body) <:> Br(startLabel) <:>
End <:> End
```

So, the third version we thought of implementing was the following one:

```
val whileLabel = getFreshLabel()
val blockLabel = getFreshLabel()


Loop(whileLabel) <:> Block(blockLabel) <:>
cgExpr(cond) <:> Eqz <:> If_void <:>
Br(blockLabel) <:> Else <:> cgExpr(body) <:>
Br(whileLabel) <:> End <:>
End <:> End
```

We wanted to compare these solutions with an existing one. So, we used *Emscripten* and compiled with it a while loop written in C to obtain a WebAssembly code. We find out that the third solution was equivalent to the one obtained with *Emscripten*. However, we decided to keep the solution inspired from the lecture even if it the longest.

A final observation that need to be done is that we added at the end of the code generation for the while loop (as for the variables) a *Const(1)* in order to keep the program correct and consistent with the way it was already implemented. Indeed, as our while loop and our variables were of type UnitType and didn't return anything, they didn't leave anything on the stack.

4. Possible Extensions

We managed to do everything that was planned. However, we could extend the actual compiler in many more ways.

The lexer as it is now, is particularly rigid and very specific to our language. However by careful manipulation of the *REASSIGNPOSSIBLE* flag provided with the lexer, extensions to the language are not a far cry.

For example we could extend our language with type Inference:

```
var i = 0;
```

We might face the problem of incorrectly producing a *VARIDREASSIGN()* token for *i = 0*, but if we set the *REASSIGNPOSSIBLE* to false after lexing the *var*, we might be able to parse this correctly and make way for type inference in our language

We could also for example extend the concept of variables and add global variables.

Special care has to be taken when handling variables and scopes if we choose to extend our language with nesting blocks and higher order functions since it requires a correct handling of the different environments and closures.

Moreover, we could also implement other imperative features. For instance, other type of control structure or loops that are usually used in imperative languages could be added. An example of another loop usually used is the "for loop". Furthermore, "continue" or "break" statements could also be added.

References

V. Kuncak. *Compiling Loops*. EPFL, 2017.