Final project report

Authors: AUMONT Adrien, KNABENHANS Félix
Date: May 2025

Abstract

The goal of our project was to implement variables, while loops, and arrays in the Amy programming language. To support variables, we needed a way to both initialize and modify them. This required adding on top of the previous labs while ensuring nothing breaks. We also had to make syntax decisions for how these new features should appear in Amy. The same can be said of the integration of while loops and arrays. In the end, we managed to implement everything we wanted to. Variables in Amy can now be initialized and updated in a manner similar to C with the exception that variable need to be initialized with a value. While loops follow a structure comparable to if statements, and fixed-size arrays are now fully type-checked and can be modified.

We decided that arrays in Amy should always be mutable, and variables must be initialized before use. Currently, there is no syntax for filling the array with a given value at initialization. Overall, these changes provide a consistent and expressive way to use variables, while loops, and arrays in Amy.

Introduction

In the first part of our project, we implemented a simple compiler for the Amy language, a simple purely functional language. We implemented a compiler pipeline starting by lexing (splitting the code into tokens), then parsing the Amy base code. This part filters basic syntax errors using the LL1 grammar defined in the parser.

We obtain a nominal AST, that is then passed through the name analyzer to obtain a symbolic AST. It checks that variables are defined before being used, no name ambiguity, and also that ADTs are correctly defined.

We then check that the code follows typing rules. Finally, the code can be either interpreted using Scala or compiled to WASM.

Our extension needs to review each layer by adding the necessary keywords, grammar rules, naming rules, type rules, and a new set of instructions. Our extension will only support compilation.

Our final goal is to be able to simplify some verbose code, necessary by the functional paradigm, using imperatives features.

Example

```
object Fib
  def fibonacci(n:Int(32)): Int(32)[] = {
    if(n <= 1){error("")}else{()};

    val fib:Int(32)[] = new Int(32)[n];
    fib[0] = 0;
    fib[1] = 1;
```

```
    var i:Int(32) = 2;
    while (i < n) {
      fib[i] = fib[i - 1] + fib[i - 2];
      i = i + 1
    };
    fib
  }
  val res:Int(32)[] = fibonacci(10);
  var i:Int(32) = 0;
  while (i < res[]) {
    Std.printString("index : " ++ Std.intToString(i) ++
                    " res : " ++ Std.intToString(res[i]));
    i = i + 1
  }

end Fib
```

What is possible

```
// Simple variable definition
var <name> : <type> = <initialValue> ?<match ...>; <nextExpression>
// Modify a variable
<varName> = <newValue> ?<match ...> ?<; nextExpression>
// Arrays definition
var/val <name> : <type> [] = new <type>[<size>] ?<match ...>; <nextEpression>
// Get array size
<arrayName> []
// Get array value
<arrayName> [<index>]
// Set array value
<arrayName>[<index>] = <newValue> ?<match ...> ?<; nextExpression>
// While loop
while(<condition>){
    <Expression(s)>
} ?<match ...>
```

What is not possible

```
// Mutate a val
<valName> = <newValue> ?<match ...>
// Set array size
<arrayName>[] = <newValue> ?<match ...>
```

Implementation

Variable Initialization

1. **Lexer:** Add keyword `var`.

2. **Parser:** Add grammar rule `var <identifier> : <type> = <expr_without_val/var_reassign> ; Expr`, same as a `val`, represented as `Assign()` case class.
3. **Name Analyzer:** Track mutable variables in a set.
4. **Type Checker:** Type check similar to `Let()`.
5. **Code Generation:** Local variables are mutable in WASM, no extra code needed.

Variable Reassignment

1. **Lexer:** Nothing to add.
2. **Parser:** Most difficult part of this project, adding this reassignment to the grammar without using a distinct prefixed keyword. Since a reassignment is `<identifier> = <expr_without_val/var_reassign> ?<match ...> ?<; expr>` in the grammar it causes LL1 to break as an operator can also start with `identifier`.

   The behavior of reassignment is comparable to a val/var definition without the obligation to follow it with a next expression. The problem is that the other element that also start with `identifier` are simple expressions. Those elements can be chained using operators but not the reassignment.

   If reassignment is a simple expression, we have a conflict in the following case `id = 1 * id = 2` can be parsed as `(id = 1)*(id = 2)` or `id = (1 * (id = 2))`. There is probably a way to solve it in the operator but none of the option we thought was practical.

   The solution we found was to consider reassignment as it's own categorie, embracing the first first conflict and solving it with a tedious left factoring. This left factoring is complicated because it needs to look at the next token after id. If the next token is = we know it's a reassignment, hence easy to parse.

   If it's an operator we already parsed too far, we can not simply call operator on it as we are already 2 tokens ahead. This means we have an operation detached from the operator tree. We need to insert it properly in the tree (correct precedence and associativity). We need to do all this while maintaining proper match structure etc.

   If the next token is different than =or any operator we can continue to parse normaly (foreshadowing: if it's a `[` not that easy). [IMAGE: Figure]

   We get from the above parsing to a single asignement with its new value having the expected operator tree.
3. **Name Analyzer:** Check that the name exists and is in the mutable var set
4. **Type Checker:** Add a constraint that new value is the same type as the variable, variable reassignment is of `Unit` type
5. **Code Generation:** Set the local variable to the new value

---

Arrays

1. Arrays are always mutable, it is not possible to get a view of an array.
2. **Lexer:** Add keyword `new`.

3. **Parser:**

- Add `type[]` referring to a new Type Tree containing the type of the element inside the array.
- Add `new <type>[<expr>]` to the grammar for array initialization
- Add `<identifier>[]` and `<identifier>[<expr>]` as simple expressions. Size getter and value getter are represented by different case classes for simplicity.
- Add a different type of reassignment for arrays as `[] = ?<match ...>“.

4. **Name Analyzer:** Only transform nominal expression to symbolic expression.
5. **Code Generation:**

- Arrays are stored in memory and defined by their address (same as objects).
- We store the size at the memory pointed by the array address.
- Array creation must be with a positive size (> 0)
- Every accesses to an array are bound check (both load and stores), if it goes out of bound the program crashes (unreachable instruction executed).

While Loops

1. **Lexer:** Add keyword `while`.
2. **Parser:** add `while(<expr>){<expr>}` to the grammar as you would for an `if` statement.
3. **Name Analyzer:** Only transform nominal expression to symbolic expression.
4. **Type Checker:** Add a constraints for the condition to be a boolean, and just fully type check each expression fully. While loops are of Unit type.
5. **Code Generation:** Code schema: loop -> condition -> jump end block if not condition -> while body -> jump to loop -> block

---

Extensions

- Modifiable variables could be enhanced further by allowing pass by reference in function calls, but since WASM does not support it we didn't implement it.
- Arrays could be further extended by adding a way to fill them with values at creation, match could also be supported, could allow multidimentional arrays etc.
