Implementation report

Anthony Sébert1 , 31/11/2018

Prerequisites

Microsoft .NET Core 2.x, available at https://dotnet.microsoft.com/download

Getting started

Decompress the folder, then open a terminal.

```
cd path/to/folder/Compiler
dotnet build
```

Note: if the terminal starts in another drive, just type the name of the drive where the project folder is located, i.e. D:

Compiling a file

Once you are in the Compiler folder, run the program (a sample source code file is already provided).

```
dotnet run path/to/myfile.tri
```

And voilà!

Code generation

The *checker* and the *encoder* parts work from the results of the lexing & parsing steps.

Checker

The **checker** consumes the *abstract syntax tree*. Its purpose is to check whether the structure of the tree is semantically valid. More specifically, these components check the variable types, the variables declarations, and the scope rules.

Regarding this last aspect, a particular data structure is used to represent the *symbol table*: a stack, making possible the support of nested scopes. It contains all the identifiers that have been processed so far and their attributes.

The *abstract syntax tree* is browsed in a simple manner by the inclusion of a convenient design pattern: the visitor pattern. A global interface ensures the compatibility between all visitors, through the implementation of a unique function for each element of the language: `Visit<TArg, TResult>()`.

Identifiers defined in the *standard environment* (types, constants, procedures, functions and operator definitions) are included from the start in the global scope, the first level of the symbol table, so they are recognized during the compilation phases.

Encoder

The **encoder** uses the *abstract syntax tree* that has been checked during the previous step. It is responsible for producing the *processor instructions* corresponding to the source code. In the present case, the compiler will produce an output file in *Triangle Assembly Language*, mean to work on the *Triangle Abstract Machine*.

The variables are stored in registers during execution time, and the memory is structured in stack frames, corresponding to scopes, that contains all the necessary data. The values of the items (addresses, procedures, values) whose address or value is not known are retrieved by relative addressing by the use of dedicated operations.

Concerning *dynamic runtime items*, they are allocated on the *heap* (while static entities are found on the *stack*).

Appendix

Type checking

Samples extracted from the file Checker - Expressions.cs

```
public TypeDenoter VisitCallExpression(CallExpression ast, Void arg) {
    Declaration binding = ast.Identifier.Visit(this, null);
    if(binding is FuncDeclaration function) {
        ast.Parameters.Visit(this, function.Formals);
        ast.Type = function.Type;
    }
    else {
        ReportUndeclaredOrError(binding, ast.Identifier, "\"%\" is not a function identifier");
        ast.Type = StandardEnvironment.ErrorType;
    }
    return ast.Type;
}
```

```
public TypeDenoter VisitBinaryExpression(BinaryExpression ast, Void arg) {
    TypeDenoter e1Type = ast.LeftExpression.Visit(this, null);
    TypeDenoter e2Type = ast.RightExpression.Visit(this, null);
    Declaration binding = ast.Operation.Visit(this, null);

    if(binding is BinaryOperatorDeclaration bbinding) {
        if(bbinding.FirstArgument == StandardEnvironment.AnyType)
            CheckAndReportError(e1Type.Equals(e2Type), "incompatible argument types for \"%\"", ast.Operation, ast);
        else {
            CheckAndReportError(e1Type.Equals(bbinding.FirstArgument), "wrong argument type for \"%\"", ast.Operation, ast.LeftExpression);
```

```
            CheckAndReportError(e2Type.Equals(bbinding.SecondArgument), "wrong argument type for \"%\"", ast.Operation, ast.RightExpression);
        }
        ast.Type = bbinding.Result;
    }
    else {
        ReportUndeclaredOrError(binding, ast.Operation, "\"%\" is not a binary operator");
        ast.Type = StandardEnvironment.ErrorType;
    }
    return ast.Type;
}
```

Variables declaration

Samples extracted from the file Checker - Declarations.cs

```
    public Void VisitVarDeclaration(VarDeclaration ast, Void arg) {
        ast.Type = ast.Type.Visit(this, null);
        idTable.Enter(ast.Identifier, ast);
        CheckAndReportError(!ast.Duplicated, "identifier \"%\" already declared", ast.Identifier, ast);
        return null;
    }
```

```
    public Void VisitInitDeclaration(InitDeclaration ast, Void arg) {
        ast.Type = ast.Type.Visit(this, null);
        ast.Expression.Visit(this, null);
        idTable.Enter(ast.Identifier, ast);
        CheckAndReportError(!ast.Duplicated, "identifier \"%\" already declared", ast.Identifier, ast);
        return null;
    }
```

Scope rules

Extracted from the file IdentificationTable.cs

```
    public sealed class IdentificationTable {
        private readonly Stack<Dictionary<string, Declaration>> scopes = null;
        public IdentificationTable() {
            scopes = new Stack<Dictionary<string, Declaration>>();
            scopes.Push(new Dictionary<string, Declaration>());
        }
        public void OpenScope() { scopes.Push(new Dictionary<string, Declaration>()); }
        public void CloseScope() { scopes.Pop(); }
        public void Enter(Terminal terminal, Declaration attr) {
            if(scopes.Peek().ContainsKey(terminal.Spelling))
                attr.Duplicated = true;
            else {
```

```
                scopes.Peek().Add(terminal.Spelling, attr);
                attr.Duplicated = false;
            }
        }
        public Declaration Retrieve(string id) {
            foreach(Dictionary<string, Declaration> scope in scopes)
                if(scope.TryGetValue(id, out Declaration attr))
                    return attr;
            return null;
        }
    }
```

Visitor interface

Extracted from the file ICheckerVisitor.cs

```
    internal interface ICheckerVisitor :
        ICommandVisitor<Void, Void>,
        IDeclarationVisitor<Void, Void>,
        IExpressionVisitor<Void, TypeDenoter>,
        IParameterVisitor<FormalParameter, Void>,
        IParameterSequenceVisitor<FormalParameterSequence, Void>,
        IFormalParameterSequenceVisitor<Void, Void>,
        IProgramVisitor<Void, Void>,
        ILiteralVisitor<Void, TypeDenoter>,
        ITerminalVisitor<Void, Declaration>,
        ITypeDenoterVisitor<Void, TypeDenoter> {}
```

Registers

Extracted from the file Machine.cs

```
    public enum Register {
        CB, CT, PB, PT, SB, ST, HB, HT, LB, L1, L2, L3, L4, L5, L6, CP
    }
```

Frame object

Extracted from the file Frame.cs

```
    public class Frame {
        public static readonly Frame Initial = new Frame(0, 0);
        public int Level { get; }
        public int Size { get; }
        private Frame(int level, int size) {
            Level = level;
            Size = size;
        }
        public Frame Expand(int sizeIncrement) { return new Frame(Level, Size +
sizeIncrement); }
        public Frame Replace(int size) { return new Frame(Level, size); }
```

```
        public Frame Push(int size) { return new Frame((byte)(Level + 1), size); }
        public Register DisplayRegister(ObjectAddress address) {
            if(address.Level == 0)
                return Register.SB;

               if(Level - address.Level <= 6)
                return (Register)(Level - address.Level);

            return Register.L6;
        }
    }
```

Operations

Extracted from the file Machine.cs

```
public enum OpCode {
    LOAD, LOADA, LOADI, LOADL, STORE, STOREI, CALL, CALLI, RETURN, PUSH, POP, JUMP,
JUMPI, JUMPIF, HALT
}
```

Note: irrelevant parts of the fragments have been omitted so as to lead the eye of the reader to the essential.

[1]: Computer Science student at The Robert Gordon University (Garthdee House, Garthdee Road, Aberdeen, AB10 7QB, Scotland, United Kingdom)