

Implementation of the Finito Programming Language

Robbert Haarman
s0070122
Studentassistent: Ivo Pooters

June 27, 2005

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 3 |
| 2 | Language Overview | 4 |
| 3 | Implementation Problems | 5 |
| 4 | The Finito Language | 6 |
| 4.1 | Syntax | 6 |
| 4.2 | Contextual Constraints | 7 |
| 4.3 | Semantics | 7 |
| 5 | Code Templates | 9 |
| 6 | The Finito Compiler | 10 |
| 6.1 | The Lexical Scanner and Parser | 10 |
| 6.2 | The Checker | 10 |
| 6.3 | The Code Generator | 10 |

| | | |
|----------|-------------------------------|-----------|
| 6.4 | The Compiler Driver | 10 |
| 6.5 | State | 11 |
| 6.6 | Symbol | 11 |
| 6.7 | Tree | 11 |
| 7 | Tests | 12 |
| 8 | Conclusions | 13 |

1 Introduction

This document describes the implementation of the programming language Finito, as performed for the course Vertalerbouw at Twente University, academic year 2004/2005.

The assignment is to design a programming language and implement a compiler for it. The programming language must have certain features, which are illustrated in the assignment by means of an example language. The example language is specified as a basic expression language, as well as several optional extensions. Finito is a basic expression language with `if` and `while`.

2 Language Overview

Finito closely resembles the example language from the assignment in constraints and semantics. In particular, it mimicks the example language in the following properties:

- Strong typing
- Boolean, character and integer types
- Need to declare variables before use
- Variables are declared without initialization
- Programs consist of declarations and expressions
- There is an assignment expression, which returns the value being assigned
- Expressions can have a void return type, meaning the return value cannot be used
- Read and print are built-in commands that take one or more arguments
- Read command modifies its arguments, and returns a value only if it is called with one argument
- Print command returns a value only if called with one argument
- If expression returns a value only if it has an else clause
- Only boolean expressions can be used as conditions
- Infix arithmetic, with the canonical priority rules

Despite the similarities above, Finito has a very different syntax from the example language in the assignment. Punctuation is reduced to a minimum; arguments are separated by whitespace, declarations/expressions are separated by newlines (although semicolons are also allowed).

3 Implementation Problems

This section describes problems that were encountered during the implementation of the Finito compiler, and how they were dealt with.

The first problem was ANTLR. ANTLR is meant to speed up the development of parsers and code transformers by generating Java code from a grammar file. However, it hides certain details that a developer might require access to, and requires the developer to know the ANTLR language, in addition to the implementation language. ANTLR has extensive documentation and a large FAQ, but these are not always as helpful as could be wished for. Also, ANTLR is slow, which leads to great loss of time, especially while debugging small changes.

One particular issue that consumed a lot of time was the modification of the checker to add type information to AST nodes. The default ANTLR AST type does not contain fields for type information. Fortunately, it is possible to tell ANTLR to use a different type, but the specifics of this were not immediately obvious. This resulted in lots of modify, recompile, sigh cycles.

Another problem relates to the TokenTypes files that ANTLR generates. These files are shared between lexer, parser, and tree walkers, and are rewritten far more often than necessary. This results in the classes that depend on them being recompiled much more often than necessary, which in turn slows down the recompile cycle. No solution was found for this problem, although the compiler works fine in the current situation.

4 The Finito Language

This section describes the syntax, contextual constraints, and semantics of the Finito language.

4.1 Syntax

```
 $\langle program \rangle ::= \langle body \rangle$   
 $\langle body \rangle ::= \langle decl \rangle \langle body \rangle$   
           $| \langle expr \rangle \langle compound\text{-}expr \rangle?$   
 $\langle decl \rangle ::= \langle typename \rangle \text{ identifier}$   
 $\langle typename \rangle ::= \text{ bool } | \text{ char } | \text{ int}$   
 $\langle expr \rangle ::= \text{ if } \langle expr \rangle \text{ then } \langle expr \rangle \text{ (else } \langle expr \rangle \text{ )?}$   
           $| \text{ while } \langle expr \rangle \langle body \rangle \text{ end}$   
           $| \langle comparison \rangle$   
           $| \langle funcall \rangle$   
 $\langle funcall \rangle ::= \text{ identifier } \langle expr \rangle^*$   
 $\langle comparison \rangle ::= \langle sum \rangle \text{ compop } \langle sum \rangle$   
           $| \langle sum \rangle$   
 $\langle sum \rangle ::= \langle term \rangle + \langle term \rangle$   
           $| \langle term \rangle - \langle term \rangle$   
           $| \langle term \rangle$   
 $\langle term \rangle ::= \langle operand \rangle * \langle operand \rangle$   
           $| \langle operand \rangle / \langle operand \rangle$   
           $| \langle operand \rangle$   
 $\langle operand \rangle ::= \text{ character}$   
           $| \text{ number}$   
           $| \text{ identifier}$   
           $| \langle subexpr \rangle$   
 $\langle subexpr \rangle ::= ( \langle body \rangle )$ 
```

An **identifier** consists of one or more letters and/or underscores. A **number** consists of one or more digits. **character** is `\%x`, where *x* is any character. **compop** is one of `==` (equals), `!=` (does not equal), `<` (lesser than), `>` (greater than), `<=` (lesser than or equal), or `>=` (greater than or equal).

Declarations and expressions are separated by line breaks and/or semicolons. Comments start in `#` and run to the end of the line.

4.2 Contextual Constraints

Variables must be declared before use. They must also be in scope, which means that they must have been declared in the current compound expression or any enclosing compound expression. A variable declared in a different compound expression on the same level, or on a deeper level, is not visible and hence cannot be used.

Inner declarations shadow outer declarations of the same name, meaning that, as long as the inner declaration is in scope, the variable declared in the outer declaration is not accessible.

4.3 Semantics

The semantics of each Finito construct are as follows.

Literals, be they boolean, character, or integer literals, are processed at compile time, so that the run-time equivalent of the literal is returned when the program reaches the point at which the literal occurred. The type of the return value is the type of literal; boolean, character, or integer.

Declarations are processed at compile time. The compiler chooses a location for the variable, and associates this location and the type of the variable with the symbol name. Declarations do not exist at run time, and thus have no return type.

A reference to a single symbol causes the current value stored in the variable with that name to be loaded and returned.

Arithmetic is performed according to the common rules: multiplication and division before addition and subtraction. The value of the computation is returned.

Comparisons are executed as follows: first, the two values to be compared are computed (comparisons have lower priority than arithmetic). Then the values are compared and discarded. The comparison returns **true** or **false** depending on whether its condition is satisfied or not (e.g. `==` returns **true** if its two operands are equal, **false** otherwise).

Function calls are processed as follows: first, the values of the arguments are computed and stored in the positions where functions expect them. (Arguments are evaluated from left to right.) Then, control is transferred to the function by jumping to its address, saving the return address. The function computes a return value and stores it in the expected place, then returns by jumping to the return address.

Procedure calls are analogous to function calls, except that procedures take

reference parameters and do not return values. For reference parameters, the address of the parameter, rather than its value, is passed to the procedure.

If-then-else expressions are executed as follows: first the test is evaluated. Then, execution depends on whether the expression has an else clause or not. For if expressions that have an else clause, it is evaluated and its value returned if the test evaluated to **false**, or, if the test evaluated to **true**, the consequent is evaluated and its value returned. For if expressions without an else clause, the consequent is evaluated and its value discarded if the test evaluated to **true**, else no action is taken. The value that the test evaluated to is discarded in any case.

While loops have the following semantics: first, the test is executed. If the test returned **false**, the loop is finished and execution continues after it. Otherwise, the loop is executed once, after which processing starts anew with evaluating the test. While loops do not return a value; any value produced in the loop body is discarded, as is the result of the test.

5 Code Templates

TBD

6 The Finito Compiler

The compiler discussed in this document compiles Finito programs into Jasmin assembly code, which can be used to generate Java class files. The modular design of the compiler makes it straightforward to add additional target languages, although no additional target languages have been developed.

The compiler is implemented as a number of Java classes in the package `finito`. It consists of a lexical scanner, a parser, a checker, a code generator, and a compiler driver that calls the other components in the right order and with appropriate parameters. These components are described briefly in the following subsections. All of them are part of the `finito` package, and described in more detail in the Javadoc documentation that accompanies this document.

6.1 The Lexical Scanner and Parser

The lexical scanner and parser are both contained in the file `src/finito.g`. They process files conforming to the grammar from section 4.1 and build an AST with nodes of type `Tree`.

6.2 The Checker

The checker is defined in `src/Checker.g`. It is an ANTLR `TreeParser` that operates on `Tree` nodes. The checker enforces the context constraints detailed in section 4.2, and also decorates the AST with type information.

6.3 The Code Generator

The code generator provided with this compiler is defined in `src/JVMCodeGenerator.g`. It operates on a decorated AST (as produced by the checker), and generates Jasmin assembly code that can be assembled into JVM class files.

6.4 The Compiler Driver

The compiler driver ties together the lexer, parser, checker and code generator into a single command-line program that reads Finito code and produces either Jasmin assembly or an error message. The driver lives in `src/Compiler.java`.

Called with no arguments, the compiler driver reads Finito source from standard input, and writes assembly code to standard output, which can be

assembled into a class called **Main**. Command-line arguments can be used to modify this behavior:

-n *name* Set the name of the class to be produced to *name*, instead of **Main**.

-o *file* Send output to *file*, instead of standard output.

file Read input from *file*, instead of standard input.

6.5 State

The code generator uses a helper class **State** to bundle information about visible symbols and generated code. A new **State** is instantiated for every compound expression, so that each compound has its own symbol table. Nested expressions result in nested **States**.

Code is generated inside a **State** object, so that some information can be prepended to it before the eventual emission. For example, in the JVM, a method needs to declare how many stack words and local variables it uses. This has to be declared at the top of the method in Jasmin assembly, but the information is only available after the whole method has been processed. Using **State** objects, the code generator can gather information and generate code, then emit everything in the right order.

6.6 Symbol

The **Symbol** class defines a storage format for symbols. For each symbol, its name, its location, and its type are stored. The name is the textual representation of the symbol. The type is an integer; integer constants for the types **void**, **char** and **int** are defined. The location is an integer. In the JVM code generator, it is the number of the local variable in which the value of the symbol is stored; in another implementation, it might be an index in a table describing the location, a memory address, etc.

6.7 Tree

Because the **AST** class provided by ANTLR does not store type information, but the Finito compiler needs it, the Finito compiler uses its own class **Tree** for **AST** nodes. **Tree** extends ANTLR's **CommonAST**, adding methods for getting and setting the return type of a node. Type information is added by the Checker, and is used to verify that the program is well-typed, as well as to add correct JVM type tags in the output Jasmin code.

7 Tests

The directory `tests` contains several test programs for the compiler. There are simple test programs that test the correct behavior of individual language constructs, simple test programs that test the correct behavior of the compiler in the face of errors, and one extensive test program that tests every aspect of the language. This last program is discussed in the appendices to this document.

8 Conclusions

On the whole, this assignment has been an enjoyable conclusion of an interesting course.