Tard

Jaap Suter

August 3, 2005

Contents

Chapter 1

Introduction

Tard is a simple language developed as a hobby project. The three main goals were:

- Learn more about compiler development
- Evaluate ANTLR
- Find out more about targeting the .NET virtual machine.

The documentation will draw some conclusions about these goals near the end, but will focus of the actual document is on the language and its implementation instead. The purpose of this document to give the user an overview of the project so he or she can understand the language, implementation and tests. Using this, the reader might be able to also draw conclusions on the power of ANTLR and the .NET virtual machine.

Tard stands for Totally Awesome Rapid Deployment. This is a base jumping maneuver which just happened to be what was on my mind when this project was started. It has no relevance to language features or implementation details.

Chapter 2

The Language

Tard is a simple language. It borrows many syntax elements from C but there are also a few important differences. It has currently no support for functions. The parser is capable of parsing multiple function definitions and does constraint checks on the application entry point. However, there is no support for calling functions, parameter lists or early returns. Time restrictions limited the amount of effort in this direction.

What is left is an expression based language in which a program has one entry point:

```
void main() { ...}
```

A program can have only one entry point (called `main`). The entry point takes no parameters, and returns void. The language supports the following basic types: `void`, `bool`, `char`, `int`, `string`.

A variable is introduced by declaring the type followed by the identifier. For example:

```
int a;
bool b;
char c;
string s;
```

Variables need to be declared before they are used. Using variables is done in expressions. Every statement that is not variable declaration is an expression with resulting type (potentially `void`). Tard supports the following expressions:

- `write(x0, x1, .., xn);` - Prints the values of the expressions to the screen
- `read(x0, x1, .., xn);` - Acceps user input and stores it into the given variables
- `x0 = x1 = ... = xn = expression` - Assignment

- `+, -, *, /, &&, ==, !=, <=, <, >=, >, etc.` - All the usual mathematical and relational operators.
- {`list of expressions`} - Compound statement, where the last expression is the value of the entire compound block
- `if (boolean expression) expression [else expression]` - If statement with optional else clause.
- `while (boolean expression) expression` - While statement

Since blocks are also expressions, blocks can be arbitrarily nested. Variables can be declared inside of a block and can have the same name as a variable in an enclosing block. Evaluating variables will always refer the most nearby declaration.

Chapter 3

Problems Encountered

This section outlines some of the problems encountered in the development of Tard and explains their solutions.

3.1 ANTLR Tree Walkers

The ANTLR grammar file specifies a lexer and and a parser. The parser is setup so it builds an abstract syntax tree (AST). ANTLR allows the user to also specify a tree-walker in which action and rewrite rules can be embedded.

There are two problems with these tree-walkers. The first is that ANTLR seems to have a restriction that a grammar file can only contain a single tree walker. No matter what was tried, ANTLR would not allow the definition of two separate tree walkers in a single file. Having two tree walkers is an obvious requirement because the contextual analysis is separated from the code generation fase.

The second problem is that embedding actions inside of the tree-walkers becomes very confusing. Defining too much code inside of the grammar specification not only makes things harder to debug, it also complicates the distinction between the grammar and the actual compiler.

Fortunately, ANTLR allows for so-called heterogeneous abstract syntax trees. In an heterogeneous AST every node can be it's own type. All node types are derived from a common base class, but each node can implement its own behaviour depending on its context and type. This allows for a very clean grammar file containing only a lexer and a parser. The parser definition contains rules on how to produce specific node type instances for different rules. When built, the AST is a gigantic polymorphic tree that allows for total customizability. The grammer contains a minimal amount of C++ code.

Each node type in the tree implements at least two functions. The first one visit each node in the tree so they can all perform a full contextual analysis, thereby checking types and scope constraints. In this phase, the AST also generates information on which variables belong to what scope and what occurences refer to what declaration.

The second function performs the code-generation. Again the entire polymorphic AST is visited and each node generates pieces of assembly meanwhile delegating code-generation tasks to child-nodes too.

The resulting design is very clear and elegant. The grammar is solely responsible for lexing, parsing and AST construction. For each lanuguage construct there exists a distinct node-type implemented in C++ and responsible for contextual analysis and code-generation, split up in two or more functions.

3.2 Ambiguity and LL(K)

One goal of the grammar specification was that it would allow for an LL(1) parser. In ANTLR, lexers are parsers as well. For Tard, the lexer is LL(2) to easily make a distinction between `<=` and `<`. There is a single ambiguity left in the lexer related to newline handling. Depending on the platform, text files use `\n`, `\r` or `\r\n`. In practice, this ambiguity is never a problem.

The actual parser for Tard is LL(1), which was the goal. There were two main problems with making the parser LL(1). The first one is related to assignments. Because assignments are also expressions, the rule `identifier = expression` is ambigious since the right-hand-side expression could contain another assignment, or just a lone identifier. The `if-then-else` construct poses another ambiguity due to the optional `else` part.

Fortunately, ANTLR allows for special-case peeks into the token buffer. By decorating the production rule with a prediction set `identifier ASSIGN => assignment | identifier` the ambiguity is resolved and the parser can remain LL(1).

3.3 Left Hand Side of Assignments

Assignments are also expressions, which allows for chaining of assignments. For example: `x = y = z = 3 + 4;`. However, this causes an ambiguity that can not easily be resolved by updating the parser grammer. The straight forward grammar allows for constructs like `4 - x = y + 3 = 4 * y = 4`; which is obviously not correct. The solution is to perform extra checks on the left-hand-side of the assignment construct during the contextual analysis phase. Then, it can easily be detected and produce a diagnostic.

3.4 Nesting Of Blocks

Tard allows for nested blocks in which variable names can be reused for different actual variables. However, the underlying target machine (the .NET virtual machine) has no concept of nested blocks. Local variables are always declared at function scope. Therefore, all variables from all blocks in a function need to be collected and declared up front at the beginning of the function.

To avoid name-collisions, we append the node-depth of each declaration to the actual variable name. In other words, the virtual machine won't refer to variable `x`, but to `x_2` and `x_3`. The context analysis phase uses the generic visitation pattern to visit all block expressions within each function to collect all the different variable declarations.

Chapter 4

Syntax, Constraints and Semantics

This section defines some production rules in which * stands for zero-or-more, + for one-or-more, — for alternative and [] for optional. Literals are defined within quotes.

4.1 Program

A program is a collection of functions. All functions are fully checked for correctness and generate full code. However, without the ability to call other functions, the only function that is actually ever executed is the `main` function, e.g. the entry point. The production rule for a program is:

```
program -> fun_def+
```

There should be at least one function definition called `main` that returns `void`. Functions can't take parameters anyway, so the constraint that the entry point taks no parameters is implicit in the grammar.

Upon execution of the program, the function called `main` is executed.

4.2 Functions

A function is a combination of a name, a return type and a block expression. A function can't take any parameters. The production rule is:

```
fun_def -> type identifier "()" block
```

The return type of the function needs to be in correspondence with the type of the block expression. There can be only one function definition for each name.

When a function is called, its block expression is evaluated and the resulting value, if not `void` is left on the stack for the caller to use.

4.3 Blocks

A block is a list of staments enclosed in curly braces. Its production rule is:

```
block -> "{" statement* "}"
```

The result type of a block is equal to the last statement in the block. If the block is empty or it contains only variable declarations or the last expression has type `void`, the type of the block itself is `void`.

When a block is evaluated, each statement is evaluated from top to botom. When a statement leaves a value on the stack it is popped off, unless it is the last stament in the block.

4.4 Statement

A statement is a variable declaration or an expression followed by a semi colon. The production rules is as follows:

```
statement -> var_decl | expression ";"
```

4.5 Declarations

A variable declaration consists of a type and an identifier:

```
var_decl -> type identifier
```

Variable initialization can be done in a separate statement following the declaration. Within each block, there can be only one variable with a given name. Variables shadow variables with the same name in enclosing blocks. Void variables are not allowed.

A variable declaration results in the compiler making room for it on the stack.

4.6 Types

A type can be one of the following:

```
type -> "string"
      | "char"
      | "void"
      | "bool"
      | "int"
```

4.7 Expressions

An expression is an application and combination of operators and values. The production rules that define all the operators are:

```
expression   -> and_expr ["||" and_expr]
and_expr     -> compare_expr ["&&" compare_expr]
compare_expr -> add_expr [("<" | ">" | "<=" | ">=" | "!=" | "==") add_expr]
add_expr     -> mul_expr ("+" | "-" mul_expr)*
mul_expr     -> sign_expr ("*" | "/" sign_expr)*
sign_expr    -> [ "+" | "-" ] not_expr
not_expr     -> "!"* value
```

This set of production rules implicitly defines the precedence of operators. The precedence is defined similar to the ones found in most other languages

The type constraints for binary operators are defined as in the following table. Lhs stands for left hand side and rhs for right hand side. Comparable means `int`, `bool` or `char`. Type of lhs means that it needs to have the same time as the left hand side of the operator.

Operator	Lhs	Rhs	Result
+	int	d	d
-	int	int	int
	int	int	int
/	int	int	int
¡	int	int	bool
¿	int	int	bool
¿=	int	int	bool
¡=	int	int	bool
==	comparable	type of lhs	bool
!=	comparable	type of lhs	bool
&&	bool	bool	bool
——	bool	bool	bool

The type constraints for unary operators are defined in the following table.

Operator	Operand
+	int
-	int
!	bool

The type of a complete expression is the type of the last `or_expression`.

The evaluation of an expression is done according to their intuitive meaning. Nothing is said about shortcircuiting of the boolean predicates. A compiler is free to do this or not, and no program should rely on behaviour either way.

4.8 Values

A value is something that can be used by operators to create more complicated expressions. The production rule is:

```
value -> literal
       | assignment
       | read
       | write
       | "(" expression ")"
       | block
       | if_statement
       | while_statement

literal -> "true"
         | "false"
         | integer
         | character
         | string
```

The production rules for `integer`, `character`, `string` are defined in the lexer and have their intuitive C-style meaning.

Values can have type `void` which means they result in no value.

4.9 Assignment

An assignment is an identifier followed by an optional assigment:

```
assignment -> identifier ["=" expression]
```

Because identifiers are assignments are expressions, this allows for chaining of assignments, as in `x = y = 3`. Because the actual trailing assignment is optional, this rule also works for just references identifiers.

Identifiers are defined in the lexer and have their intuitive C-style definition. Identifiers are introduced by a variable declaration. The variable declaratin needs to preceed the use of the identifier.

The type of the right hand side expression needs to be the same as the type of the left hand side identifier. Ambigious chaining of ssignments as in `x = y + z = 3` is explicitly forbidden.

Evaluating an assignment statemen proceeds as follows. If there is an assignment on the right hand side, the expression is evaluated. the resulting value is then stored into the memory location associated with the left hand side identifier.

Then, regardless of whether or not there is a right hand side assignment, the value associated with the left hand side identifier is loaded, resulting in the value of the entire assignment expresssion.

4.10 Read and Write

Read and write statements are what allows the program to perform input and output. The production rules are:

```
write -> "write" "(" expression ("," expression)* ")"
read  -> "read" "(" identifier ("," identifier)* ")"
```

The `write` statement takes a variable length list of expressions. It evaluates every single one and prints it to the screen followed by a newline. The number of parameters must be at least one. If it is one, the resulting type and value are the type and value of the expression. If there are more than one parameters, the resulting type of a `write` expression is void.

The `read` statement takes a variable length list of identifiers of at least length one. The identifiers must refer to a variable declared earlier in the program. The value read must have the same type as the type of the variable declaration, otherwise a runtime exception is thrown. The value is then stored in the memory location associated with the variable.

The return type and value of a read is equal to the first parameter if only one parameters is given, `void` otherwise.

4.11 If Statement

An if statement allows for conditional execution of code. It consists of a condition, a then-expression and an optional else-expression.

```
if_statement -> "if" "(" expression ")" expression ["else" expression]
```

The scope of the variables declared in the condition extends to all three expressions. Variables declared in the other two expressions are limited to their own scope.

The type of the condition must be boolean. If the condition is true, the then-expression is evaluated and the optional else-expression is ignored. If the condition is false, the then-expression is ignored, and the optional else-expression is evaluated.

The type of the if statement is void if the type of the then-expression and else-expression are different of if there is no else-expression. Otherwise the type is equal to the type of the then- and else-expressions and the resulting value is the value of whatever expression is executed.

4.12 While Statement

A while statement allows for loop constructions. It consists of a condition-expression that must be of boolean type, and a body-expression that can be of any type.

```
while_statement -> "while" "(" expression ")" expression
```

Regardless of the body, the type of the entire while statement is always `void`.

The scope of the variables declared in the condition extends to the body. Variables declared in the body are limited to their own scope.

Upon evaluation of the while statement, the condition and body are evaluated one after another in a loop, until the condition evaluates to false.

Chapter 5

Code Generation

This section provides details on how code is generated for each of the different language constructs. This section is heavily .NET CLR dependent, although the syntax of the .NET Microsoft Intermediate Language is quite straightforward.

5.1 Code Templates

The following list gives a pseudo-code overview of all the code templates used to generate different constructs. Everything in between quotes is literal .NET intermediate language code. The ## operator means a literal concatenation of strings. For exact details and precise code templates, refer to the `generate_code` function of the actual implementation of ech language construct.

5.1.1 Program

```
program -> define[fun_def rest_of_functions] =
    define[fun_def]
    if not empty rest_of_functions:
        define[rest_of_functions]
```

5.1.2 Function Definition

```
fun_def -> define[type identifier block] =
    ".method static public" type identifier "il managed"

    if is entry point function
        ".entrypoint"
    ".maxstack 128"
    ".locals init"
    "("
        extract_variable_declarations[block]
    ")"
    evaluate[block]

    if type of block not is void
        "pop"

    "ret"
    "}"
```

5.1.3 Block

```
block -> extract_variable_declarations[statement rest_of_statements]

    if statement is var_decl
        define[var_decl]

    if statement is block
        extract_variable_declarations[block]

    if not empty rest_of_statements
        extract_variable_declarations[rest_of_statements]

block -> evaluate[statement rest_of_statements] =

    if statement not is var_decl
        evaluate[statement]

    if not empty rest_of_statements

        if type of statement is not void
            "pop"

        evaluate[rest_of_statements]
```

5.1.4 Variable Declaration

```
var_decl -> define[type identifier] =

    type identifier ## depth_of_var_decl_block ","
```

5.1.5 Expression

```
binary_expression -> evaluate[lhs op rhs] =

    evaluate[lhs]
    evaluate[rhs]
    op

unary_expression -> evaluate[operand operator] =
    evaluate[operand]
    operator
```

5.1.6 Assignment

```
assignment -> evaluate[lhs optional_assignment] =

    if optional_assignment
        evaluate optional_assignment
        "strloc" internal_name[lhs]

    "ldloc" internal_name[lhs]

identifier -> internal_name[name] =

    name ## depth of look_up_closest_var_decl_with[name]
```

5.1.7 Read

```
read -> evaluate[identifier rest_of_identifiers] =

    "call string [mscorlib]System.Console::ReadLine()"

    if type of identifier not is string
        "call int32 [mscorlib]System.Int32::Parse(string)"

    if total number of parameters is one
        "dup"

        "stloc" internal_nam[identifier]

    if not empty rest_of_identifiers
        evaluate[rest_of_identifiers]
```

5.1.8 Write

```
write -> evaluate[expression rest_of_expression] =

    evaluate[expression]

    if total number of parameters is one
        "dup"
    "call void [mscorlib]System.Console::WriteLine("
    type of expression
    ")"

    if not empty rest_of_identifiers
        evaluate[rest_of_identifiers]
```

5.1.9 If Statement

```
if -> evaluate[condition then else] =

    label_id = unique_label_for_this_if_statements

    evaluate[condition]

    "brzero else_" ## label_id

    evaluate[then]

    if type of if is void and type of then is void
        "pop"

    if else
        "br skip_else_" ## label_id
        "else_" ## label_id ":"
        evaluate[else];
        if type of if is void and type of else is void
            "pop"
        "skip_else_" ## label_id ":"
    else
        "else_" ## label_id ":"
```

5.1.10 While Statement

```
while -> evaluate[condition body] =

    label_id = unique_label_for_this_if_statements

    "while_begin_" ## label_id ":"
    evaluate[condition]
    "brzero while_end_" ## label_id
    evaluate[body]
    if type of body is void
        "pop"
    "br while_begin_" ## label_id
```

```
"while_end_" ## label_id
```

Chapter 6

The Implementation

The implementation is quite straightforward. The public interface is defined in two header files, one is the compiler and the other one defines the exceptions that the compiler can generate.

The core of the compiler is defined inside of `detail` headers. There is a single base class called `node` that is derived from the `ANTLR::CommonAST` type. This `node` type defines the common interface that all language constructs share, as well as some utility funtions.

Every language construct derives from `node` and implements its own specialization for both context analysis as well as code generation. The actual node class is never instantiated as only derived classes are used.

Inside the compiler, an ANTLR lexer and a parser are instantiated. These are used to parse the input stream and create the abstract syntax tree.

Then, the AST does an entire context analysis pass making sure that the program is semantically correct. If this succeeds, another pass is done over the now-decorated AST in which each node generates code and sends it to the given output stream.

The resulting output is a well-formed .NET intermediate language program that can be assembled by `ilasm.exe` into an executable. More information about this step can be found in the `readme.txt` found in the Tard root directory.

If an error occurs during the compilation fase, either a parse error or a contextual error, an exception is thrown from the `compile` function. The exception will be of type `tard_exception` and have an enumeration indiciating the type of error. No other information is stored in the exception but it would be trivial to decorate it with error specific information, e.g. a string explaining the problem. For testing purposes, the enumeration suffices.

Chapter 7

Testing

The compiler comes with an extensive suite of tests. The tests can be categorized in one of three categories, `parse_error`, `context_error`, `well_formed`. There is one special well formed program called `large_program` that aims to pack many different Tard features in to a single program. All the tests are located in the `tard/test/scripts` directory.

Since the compiled programs execute on the .NET virtual machine, no effort is spend testing runtime errors. A division by zero on the virtual machine will simply generate the appropiate .NET exception, which clearly lies outside of the Tard domain.

7.1 Parse Error Tests

Since ANTLR takes care of the lexing and parsing itself, there is not that much to test as far as parse errors go. The two simple tests for parse errors test a garbage program, as well as a program in which the number of opening and closing parenthesis are mismatched.

test_parse_error_garbage

test_parse_error_paren_mismatch

It is assumed that if these two programs manage to throw the proper `parse_error` exception, the ANTLR parser is working correctly.

7.2 Context Error Tests

Context errors are much more common and obviously not caught by ANTLR. Therefore, the test-suite comes with a large number of context error specific tests.

test_context_error_multiple_entry_points

This tests makes sure the compiler detects it if a program implements more than one entry point.

test_context_error_entry_point_signature

This test makes sure the compiler detects a signature mismatch for the application entry point.

test_context_error_define_before_use

Makes sure the compiler can detect when a variable is used before it is defined.

test_context_error_multiply_defined_symbol

Detects when two variables of the same name are declared in the same scope.

test_context_error_lhs_of_assignment

Detects when the left hand side of an assignment statement is more than just a simple identifier.

test_context_error_assign_bool_to_int

Detects a type mismatch, trying to convert a boolean to an integer.

test_context_error_assign_int_to_bool

Detects a type mismatch, trying to an integer to a boolean.

test_context_error_mix_int_and_bool

Detects a type mismatch, trying to us an integer and a boolean inside of a single binary expression.

test_context_error_mix_bool_and_int

Detects a type mismatch, trying to us an integer and a boolean inside of a single binary expression.

test_context_error_multi_write_in_expression

Makes sure that a write of multiple values has type void and can't be used in expressions.

test_context_error_undefined_identifier

Makes sure that undeclared variables can't be used.

test_context_error_if_scope_error

Detects a scoping problem in the if-statement. Trying to use a value declared in the then-expression inside of the else-expression.

test_context_error_if_condition_type_error

Makes sure the compiler can detect if the condition of the if statement is not boolea.

test_context_error_ambigious_if_is_void

Makes sure that the return type of an ambigious if is void.

7.3 Well Formed Programs

The following program (and the `large_program`) test that certain features of the language actually do work.

test_empty_program

Tests that an empty program with just an entry point that doens't do anything can compile and run. This is the simplest possible program.

test_multiple_fun_defs

Tests that multiple function definitions are allowed.

test_string

Tests reading and writing of string types. This test is interactive since the `read` command is used.

test_var_decl

This test does some complicated variable declarations in different scopes of different nesting, also testing variable shadowing.

test_assignment

This tests assignments of different types and chaining of assignments.

test_expression

This tests some complicated mathematical and logical expressions. This test is inter-active since the read command is used.

test_write_expression

This test makes sure we can write a complicated expression.

test_write_as_expression

This test makes sure we can use a single parameter write statement as an expression of non-void type.

test_if

This test does some miscallenous testing of the if-statement. It should print the numbers one to six.

test_while

This test exercises the while statement.

7.4 Large Program

The large program test is an attempt at creating a little application that pretends to do something useful, along the way combining several different elements of Tard and demonstrating different language constructs.

7.4.1 test_large_program

This large program is a simple guess-the-number game. The user is asked to enter the maximum number of tries allowed and a random number. Then another user has to guess numbers repeatedly until he guesses it, which makes him the winner, or until he runs out of tries in which he becomes the loser. After each guess, the program will say whether the number is greater or less than the actual number.

Admittedly, it's not a very impressive program, but it demonstrates some features very neatly. What follows is the definition of this program:

```
void main()
{
    write("------------------------------------------------");
    write("             THE GUESSING GAME                  ");
    write("------------------------------------------------");

    int num_tries;
    write("please enter the maximum number of guesses allowed");
    read(num_tries);

    int rand;
    write("please enter a random number");
    read(rand);

    bool guessed_it;
    guessed_it = false;

    while(!guessed_it && num_tries > 0)
    {
        write("please guess what the number is");
        int guess;
        read(guess);

        if (guess > rand)
        {
            write("the number is smaller");
        }
        else
        if (guess < rand)
        {
            write("the number is bigger");
        }
        else
        {
            guessed_it = true;
        };

        num_tries = num_tries - 1;

        if (!guessed_it)
        {
            write("number of tries you have left:");
            write(num_tries);
        };
    };

    if (guessed_it)
        write("you are the winner")
    else
        write("you lost");
}
```

7.4.2 Example Test Output

The following is the screen output of a sample run of the large test program.

```
-------------------------------------------------
THE GUESSING GAME
-------------------------------------------------

please enter the maximum number of guesses allowed

5

please enter a random number

7

please guess what the number is

3

the number is bigger

number of tries you have left:

4

please guess what the number is

9

the number is smaller

number of tries you have left:

3

please guess what the number is

6

the number is bigger

number of tries you have left:

2

please guess what the number is
```

```
7

number of tries you have left:

1

you are the winner
```

Chapter 8

Conclusion

Tard is a simple language and compiler. It is developed as an experiment to gain more experience in compiler writing as well as gain insight in usign ANTLR and targeting the .NET virtual machine.

Tard as a language itself is far from complicated although it has some interesting scope-rules. The fact that all statements are also expressions also introduces some intresting complications. It would be relatively easy to extend Tard with some language features like function calls, parameterized functions and const declarations. Other features like arrays, records or pointers would be significantly harder.

ANTLR is a great code-generator for building parsers and lexers. It certainly outperforms Yacc and Bison in terms of user friendliness and power. The Boost Spirit library is another parser framework that I have experience with, and it remains my preferred choice for writing parsers. Embedding the code generator inside the actual compiler has many benefits.

The .NET virtual machine, or execution engine as Microsoft calls it since it does just-in-time compiling, is a very easy target to develop for. The documentation is quite excellent and the `ilasm.exe` compiler is a great tool to turn intermediate code into a final executable. The .NET execution engine will remain my platform of choice for targeting languages. It's only drawback is its lack of portability.

Previously I had written a parser, interpreter, compiler and specific virtual machine for another language called Lean. This was an S-expression based function language like Lisp and Scheme. Such a language is much easier to develop since the grammar leaves very little room for ambiguity. I am considering to port Lean to the .NET runtime and generate code in Microsoft Intermediate Language.

All in all, Tard was a good learning experience to see that a lot of work has to be done in between specifying a grammar and actually generating an executable. Especially so for imperative languages, compared to functional languages, due to their heavy context sensitivity and ambiguities.