A compiler for TinyJava - Language Processors

Ignacio Iker Prado Rujas

13th of September 2016

Abstract

In this document we cover a basic overview about the implemented compiler for a language that looks like `Java`, but that in no case achieves all of its potential. We name it `TinyJava`.

A compiler combines different phases in which several analysis are performed, each of them relying on the previous ones. It begins by analyzing the words that form the given code, and it associates to each of them different *lexical units*. Afterwards, it ensures that the sentences are well formed by means of a *context-free grammar*. The next analysis verifies *identifier restrictions* of the variables and performs a *type analysis*. Lastly, it generates *portable code* that can be executed on a *p*-machine.

1. Lexical analyzer

All regarding this analysis is inside the `lexical_analyzer` package. The class named `LexicalAnalyzerTiny` is generated by JFlex. The input for JFlex is the `grammar.flex` file.

In this first analysis words are explored in order to classify them into different lexical classes, which are symbols with certain associated attributes. These ones are mainly the *lexeme* (meaning the actual word: a *String*), its location inside the code file (line and column) and a certain class represented by an integer.

For instance, it can detect comments that are ignored, but if the word *double* is found, it returns a new lexical unit that will represent this word and its properties. This matching is perfomed by means of regular expressions, defined in `grammar.flex`.

2. Syntactic analyzer

The code of this part of the compiler is in the `syntactic_analyzer` package. The classes `LexicalClass` and `SyntacticAnalyzerTiny` are generated with CUP, thanks to the `LALR_Parser.cup` file.

Working alongside the lexical analyzer discussed above, it checks that the code file is fine in terms of its syntax, which is described by a context-free grammar and summarized as follows:

$$
\begin{aligned}
P \rightarrow\ & CLASSLIST\ \texttt{main}\ \{\ STMLIST\ \} \\
\mid\ & \texttt{main}\ \{\ STMLIST\ \}
\end{aligned}
$$

$$CLASSLIST \rightarrow CLASSLIST\ CLASSDEF \mid CLASSDEF$$

$$
\begin{aligned}
CLASSDEF \rightarrow\ & \texttt{class}\ CLASSNAME\ \{\ ATTRIBLIST\ METHLIST\ \} \\
\mid\ & \texttt{class}\ CLASSNAME\ \{\ METHLIST\ ATTRIBLIST\ \}
\end{aligned}
$$

$$
\begin{aligned}
ATTRIBLIST \rightarrow\ & ATTRIBLIST\ ATTRIBDECL\ ; \\
\mid\ & ATTRIBDECL\ ;
\end{aligned}
$$

$$
\begin{aligned}
ATTRIBDECL \rightarrow\ & \texttt{public}\ BASICTYPE\ ID \\
\mid\ & \texttt{private}\ BASICTYPE\ ID
\end{aligned}
$$

$$ATTRIBOBJ \rightarrow ID\ .\ ID$$

$$
\begin{aligned}
METHLIST \rightarrow\ & METHLIST\ METHOD \\
\mid\ & CONSTRUCTOR
\end{aligned}
$$

$$CONSTRUCTOR \rightarrow \texttt{public}\ CLASSNAME\ MPARAMS\ \{\ STMLIST\ \}$$

$$
\begin{aligned}
METHOD \rightarrow\ & \texttt{public}\ BASICTYPE\ ID\ MPARAMS\ \{\ STMLIST\ \texttt{return}\ EXPR\ ;\ \} \\
\mid\ & \texttt{public}\ BASICTYPE\ ID\ MPARAMS\ \{\ \texttt{return}\ EXPR\ ;\ \} \\
\mid\ & \texttt{private}\ BASICTYPE\ ID\ MPARAMS\ \{\ STMLIST\ \texttt{return}\ EXPR\ ;\ \} \\
\mid\ & \texttt{private}\ BASICTYPE\ ID\ MPARAMS\ \{\ \texttt{return}\ EXPR\ ;\ \} \\
\mid\ & \texttt{public void}\ ID\ MPARAMS\ \{\ STMLIST\ \} \\
\mid\ & \texttt{private void}\ ID\ MPARAMS\ \{\ STMLIST\ \}
\end{aligned}
$$

$$
\begin{aligned}
MPARAMS \rightarrow\ & (\ MPARAMSLIST\ ) \\
\mid\ & (\ )
\end{aligned}
$$

$$
\begin{aligned}
MPARAMSLIST \rightarrow\ & MPARAMSLIST\ ,\ MPARAM \\
\mid\ & MPARAM
\end{aligned}
$$

$$MPARAM \rightarrow BASICTYPE\ ID$$

$$STMLIST \rightarrow STMLIST\ STM \\ \mid STM$$

$$STM \rightarrow NEWDECL\ ; \mid DECL\ ; \mid ASSIGN\ ; \mid ASSIGNATTRIB\ ; \\ \mid ASSIGNARRAY\ ; \mid WHILE \mid FOR \mid SWITCH \mid IF\_ELSE \\ \mid VOIDMETHOBJ \mid ARRAYDECL\ ;$$

$$NEWDECL \rightarrow CLASSNAME\ ID\ =\ \texttt{new}\ CLASSNAME\ (\ EXPRLIST\ )$$

$$DECL \rightarrow BASICTYPE\ ID \mid BASICTYPE\ ID\ =\ EXPR$$

$$ASSIGN \rightarrow ID\ =\ EXPR$$

$$ASSIGNATTRIB \rightarrow ATTRIBOBJ\ =\ EXPR$$

$$ASSIGNARRAY \rightarrow ARRAYELEM\ =\ EXPR$$

$$ARRAYELEM \rightarrow ID\ [\ EXPR\ ]$$

$$ARRAYDECL \rightarrow BASICTYPE\ [\,]\ ID\ =\ \texttt{new}\ BASICTYPE\ [\ INT\ ]$$

$$BASICTYPE \rightarrow \texttt{int} \mid \texttt{bool} \mid \texttt{double} \mid \texttt{char} \mid \texttt{String}$$

$$WHILE \rightarrow \texttt{while}\ EXPR\ \{\ STMLIST\ \}$$

$$IF\_ELSE \rightarrow \texttt{if}\ EXPR\ \{\ STMLIST\ \}\ \texttt{else}\ \{\ STMLIST\ \} \\ \mid \texttt{if}\ EXPR\ \{\ STMLIST\ \}$$

$$FOR \rightarrow \texttt{for}\ (\ ASSIGN\ ;\ EXPR\ ;\ ASSIGN\ )\ \{\ STMLIST\ \}$$

$$SWITCH \rightarrow \texttt{switch}\ EXPR\ \{\ SWITCHBODY\ \}$$

$$SWITCHBODY \rightarrow SWITCHBODY\ SWITCHSTM \\ \mid SWITCHSTM$$

$$SWITCHSTM \rightarrow \texttt{case}\ INT\ \{\ STMLIST\ \}$$

$$VOIDMETHOBJ \;\rightarrow\; ID\ .\ ID\ (\ EXPRLIST\ )\ |\ ID\ .\ ID\ (\ )$$

$$EXPRLIST \;\rightarrow\; EXPRLIST\ ,\ EXPR\ |\ EXPR$$

$$METHOBJ \;\rightarrow\; ID\ .\ ID\ (\ EXPRLIST\ )\ |\ ID\ .\ ID\ (\ )$$

$$EXPR \;\rightarrow\; EXPR\ \texttt{||}\ EXPRAND\ |\ EXPRAND$$

$$EXPRAND \;\rightarrow\; EXPRAND\ \texttt{\&\&}\ EXPREQ\ |\ EXPREQ$$

$$EXPREQ \;\rightarrow\; EXPREQ\ \texttt{==}\ EXPRREL\ |\ EXPREQ\ !=\ EXPRREL\ |\ EXPRREL$$

$$EXPRREL \;\rightarrow\; EXPRREL\ <\ EXPRADD\ |\ EXPRREL\ >\ EXPRADD$$
$$|\ EXPRREL\ <=\ EXPRADD\ |\ EXPRREL\ >=\ EXPRADD\ |\ EXPRADD$$

$$EXPRADD \;\rightarrow\; EXPRADD\ \texttt{+}\ EXPRMULT\ |\ EXPRADD\ \texttt{-}\ EXPRMULT$$
$$|\ EXPRMULT$$

$$EXPRMULT \;\rightarrow\; EXPRMULT\ *\ EXPRUNARY\ |\ EXPRMULT\ \texttt{/}\ EXPRUNARY$$
$$|\ EXPRMULT\ \%\ EXPRUNARY\ |\ EXPRUNARY$$

$$EXPRUNARY \;\rightarrow\; \texttt{+}\ EXPRUNARY\ |\ \texttt{-}\ EXPRUNARY$$
$$|\ !\ EXPRUNARY\ |\ EXPRLAST$$

$$EXPRLAST \;\rightarrow\; (\ EXPR\ )\ \ ID\ |\ INT\ |\ DOUBLE\ |\ BOOLVAL\ |\ CHAR\ |\ STRING$$
$$|\ ATTRIBOBJ\ |\ METHOBJ\ |\ ARRAYELEM$$

Previously on the JFlex file we had defined that, for instance, an identifier of a variable consists of characters starting by a letter, or that a $BOOLVAL$ can only be `true` or `false`. Also, as the analysis is performed, it generates an abstract syntax tree. The nodes represent the different analyzed units, such as a node−program whose attributes are the class definitions and the list with the statements of the main program. All of this elements, the statements and the expressions are managed by the subpackage `syntax`.

If the analysis detects any syntactic errors, it will show a message on the console. If not, it will proceed to the next phase.

3. Semantic analyzer

The package named `semantic_analyzer` (alongside `error_handle`) collects this part of the code.

In this part the generated syntax tree is explored in order to detect indentifier or type errors, such as a variable that is used but not initialized or trying to add an integer expression with a boolean expression. The class `IdAndTypeChecker` is in charged of this, using `IdManager` which basically contains a stack of tables that gather information about each identifier on a certain block. Every time the code enters on a new block, a new table is pushed onto the stack, and when the block ends it is poped. Depending on the type of identifier, different kind of information is saved on the table, and this information is enclosed on a `IdInfo`.

The main tests regarding types are in the `getType()` method, on the `IdAndTypeChecker` class. See the `Errors` class to get an idea of the type of errors that you can expect it to detect.

4. Code generator

The classes needed for generating the *p*-code are in the `code_generator` package.

On this last phase of the compilation, the given code is translated into code that can be executed on a *p*-machine. The `CodeManager` is in charged of organizing this, and saving the right information. Each statement and expression has its own `code()` method (or `code_L()` or `code_R()`) that tells the `CodeManager` what instructions to add, but it can be tricky because of the jumps on the code.

In order to solve this, we use a queue of queues named `pendingCode` that saves the generated code until the required label is obtained. Other helpful information that is kept is a map that associates to each object the name of its class, a second map that associates to each method the line number where it begins and a third map that associates to each class a map that associates to each attribute an integer value that represents the offset of the attribute viewed as an element of a register:

$$\begin{aligned} \texttt{objClass}:&\ \text{Object} \longrightarrow \text{Class}, \\ \texttt{methLine}:&\ \text{Method} \longrightarrow \text{Initial line number}, \\ \texttt{attrOffset}:&\ \text{Class} \longrightarrow \text{Attribute} \longrightarrow \text{Attribute offset}. \end{aligned}$$

In this way, composing ($\texttt{objClass} \circ \texttt{attrOffset}$) we obtain for each object the offset of its attributes, which is needed for the code generation.

The generated code is forwarded to a file named `output.txt` inside the `examples` folder, from where the input is taken.