

Compiler design - grammar and semantics

F. Sørensen

July 29, 2008

Abstract

This article contains a grammar and semantics for Mini-Java. This is used in the later articles to build a compiler that can compile a Mini-Java program to Java bytecode, and allow a Java runtime environment to execute the program.

1 Overview

This article series will cover how to construct a compiler. It contains 5 articles with the titles below. It is written to capture the knowledge taught on 4.semester in Computer Science on Aalborg University(AAU).

- Compiler design - grammar and semantics
- Compiler design - lexing and parsing
- Compiler design - abstract syntax tree and contextual analysis
- Compiler design - code generation
- Compiler design - overview of the implementation

A compiler is used for language translation. It can convert or compile from one language to another. Often languages are compiled from highlevel to lowlevel. This can be C++ to x86 assembler or Java to Java bytecode. It is also possible to construct compilers that translate from one highlevel language to another. An example of this is the Toba project[Tob08] that compiles Java sourcecode to C sourcecode.

Since this article series is about compiler design and construction it will not contain much information about language design. It uses Mini-Java(MJava)[pro08b] which is a subset of Java. Mjava will be used as is and the language will not be changed. This makes it possible to check the correctness of the compiler with the example programs posted on the projects homepage. Since this article series is about the construction of the compiler many subjects are not covered in depth.

In the design of the compiler the focus will be on ease of understanding more than speed. The focus will also be to build every part of the compiler from scratch and not use tools to construct parts of the compiler. This is to make sure that all parts of the compiler will be described in detail.

The compiler will compile the MJava sourcecode to Java bytecode(JBC). JBC has been chosen primarily because it is natural for a subset of Java to be

compiled to JBC. But also because it is a binary format so it is possible to show how a highlevel language is compiled to a lowlevel language. The compiling of a MJava program is shown in figure 1, first the MJava program is compiled to JBC in (a) and afterwards it can be run on a Java virtual machine(JVM) as shown in (b). A full description of tombstone diagrams can be found in [AWFB00].

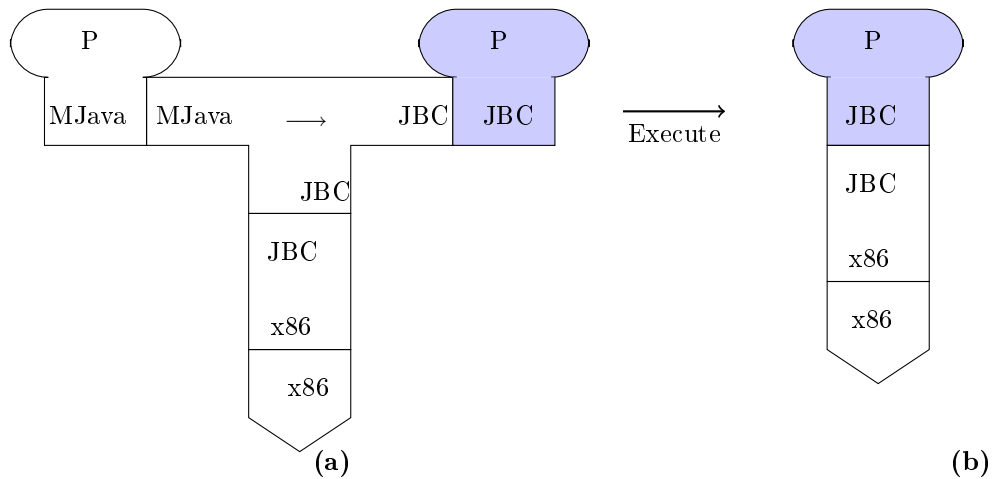


Figure 1: Tombstone diagram of a Mini-java program compiled to Java bytecode (a) (the compiler is running on a x86 machine), and then executed by a Java virtual machine on an x86 machine (b)

2 Introduction to Mini-Java

MJava is a subset of the Java language. It is very limited compared to Java but it makes for a good language to implement because it is not too large but still complete enough to allow powerful programs to be written in it. MJava only supports three different types, int, int arrays and boolean. It is possible to construct new classes but overloading is not allowed.

An example program in MJava is shown in figure 2.

The construct *System.out.println(...)*; only accepts integers as input. Also the expression *e.length()*; only applies to expressions of type *int[]*.

3 Grammar

A grammar for a language is a description of the syntax (look) of the language. It only deals with the look of the language not the meaning. As an example: the MJava grammar defines an operator $\mathcal{E}\mathcal{E}$ this is boolean and operation, but the grammar tells nothing about this it only makes sure that the operator can only be accepted in the correct places. The grammar also defines how variable names must look like but it sets no requirements that a variable name must be declared before use, this is handled by the semantics. When checking a source program against the grammar it is possible to tell if the source program is well formed not whether or not it will run. For a full explanation of grammars see [Sip06].

```

1 class Factorial{
2     public static void main(String [] a){
3         System.out.println(new Fac().ComputeFac(10));
4     }
5 }
6
7 class Fac {
8     public int ComputeFac(int num){
9         int num_aux ;
10        if (num < 1)
11            num_aux = 1 ;
12        else
13            num_aux = num * (this.ComputeFac(num-1)) ;
14        return num_aux ;
15    }
16 }

```

Figure 2: Factorial program in MJava

The grammar for MJava is given in Backus-Naur Form(BNF) in figure 3. BNF is a way to express a context-free language, it is in many ways similar to regular expressions but more powerful. For a full explanation of BNF see [Sip06].

The grammar in figure 3 is from [pro08a] it does not contain all non-terminals for the language. The terminals are described in figure 4, 5 and 6.

As specified in the grammar for identifier in figure 4. An identifier can contain any sequence of digits, letters and underscores but must start with a letter. The identifier names are case sensitive.

An integer literal is a sequence of digits with an optional - in front to signal a negative number. The grammar for this is specified in figure 5. Notice that it is legal to write an integer literal as *00* or *01*.

The last grammar for terminals are the grammar for binary operators. The only operators supported by MJava are, $\&\&$ boolean and, $<$ less than, $+$ plus, $-$ minus and $*$ times.

3.1 Lexical grammar for MJava

To construct the lexer a lexical grammar is needed. The lexical grammar differs from the language grammar, it does not capture the constructs in the language. It only focus on the different tokens in the language. That is identifiers, keywords, punctuation etc. The lexical grammar is shown in figure 7. This grammar will be used when constructing the lexer itself. The production rules will be translated directly to tokens that can be used by the parser.

4 Semantics

The semantic rules of a language defines how the different syntactical constructs should be understood and evaluated. What happens when we write *while*, which value do we reference when we reference the variable *var* if *var* has both been defined as a local variable and as a class variable.

The semantic here will be given as a big-step, operational semantic, for in-depth information about big-step semantics see [Hut04]. To allow the reader

```

Goal ::= MainClass ( ClassDeclaration )* <EOF>
MainClass ::= class Identifier { public static void main ( String
[ ] Identifier ) { Statement } }
ClassDeclaration ::= class Identifier ( extends Identifier )? { (
VarDeclaration )* ( MethodDeclaration )* }
VarDeclaration ::= Type Identifier ;
MethodDeclaration
public Type Identifier ( ( Type Identifier ( , Type Identifier )*)?
) { ( VarDeclaration )* ( Statement )* return Expression ; }
Type ::= int [ ] | boolean | int | Identifier
Statement ::= { ( Statement )* } | if ( Expression ) Statement
else Statement | while ( Expression ) Statement | Sys-
tem.out.println ( Expression ) ; | Identifier = Expression ; |
Identifier [ Expression ] = Expression ;
Expression ::= Expression <BINARY_OPERATOR> Expres-
sion | Expression [ Expression ] | Expression . length |
Expression . Identifier ( ( Expression ( , Expression )*)? ) |
<INTEGER_LITERAL> | true | false | Identifier | this |
new int [ Expression ] | new Identifier ( ) | ! Expression | (
Expression )
Identifier ::= <IDENTIFIER>

```

Figure 3: MJava grammar in BNF

```

Identifier ::= Letter SeqLetter
SeqLetter ::= Letter SeqLetter | _ SeqLetter | Digit SeqLetter
Letter ::= a-zA-Z
Digit ::= 0-9

```

Figure 4: MJava grammar for identifier

```

IntegerLiteral ::= - SeqDigit | SeqDigit
SeqDigit ::= SeqDigit Digit
Digit ::= 0-9

```

Figure 5: MJava grammar for integer literal

```

BinaryOperator ::= && | < | + | - | *

```

Figure 6: MJava grammar for binary operators

```

Token ::= Identifier | IntegerLiteral | BinaryOperator | UnaryOperator
        | Boolean | Character | ClassLevel | Statement | Main
ClassLevel ::= Class | Extends | Public
Statement ::= Return | If | Else | While | Sysout | DotLength | This
            | Type
Class ::= class
Character ::= { | } | ; | , | ( | ) | . | | [ | ] | =
Main ::= public static void main ( String [ ] Identifier )
Extends ::= extends
Public ::= public
Return ::= return
Type ::= int [ ] | boolean | int
If ::= if
Else ::= else
While ::= while
Sysout ::= System.out.println
DotLength ::= . length
Boolean ::= true | false
This ::= this
UnaryOperator ::= ! | new
Identifier ::= Letter SeqLetter
SeqLetter ::= Letter SeqLetter | _ SeqLetter | Digit SeqLetter
Letter ::= a-zA-Z
Digit ::= 0-9
IntegerLiteral ::= - SeqDigit | SeqDigit
SeqDigit ::= SeqDigit Digit
Digit ::= 0-9
BinaryOperator ::= && | < | + | - | *

```

Figure 7: MJava lexical grammar in BNF

to understand the semantics given here some of the theory is explained.

When a program executes it makes changes to memory. It is possible to view a configuration of the memory as a state. At any point of a program executing the memory will be in a specific configuration, when the next command is executed the memory will change. This change is called a transition to a new state. With this mindset it is possible to view an executing program as an oriented graph where each node is a state. This graph is called a transition system. The leaf nodes where there is no outgoing edges are called *end configurations*. When in one of those states the program terminates. The set of transitions are pairs of configurations (λ_1, λ_2) if there exist an edge between λ_1 and λ_2 in the transition system. This set defines a binary relation on Γ that is the set of configurations.

Definition 4.1 *Transition system:* A transition system is a 3-tuple, (Γ, \rightarrow, T) where Γ is the set of configurations, \rightarrow is the transition relation and $T \subseteq \Gamma$ is a set of end configurations.

It is possible to draw a transition system for a program as a graph, but for most programs the graph will be very large. Instead it is possible to define the transition system using operational semantics. The semantics for MJava will be defined using a big-step semantic. A big-step semantic defines how to evaluate a full program from start to an end configuration in one step.

If the expression $(2+3)*(1+4)$ should be evaluated. First the $(2+3)$ and $(1+4)$ needs to be calculated. Then the values they evaluate to must be multiplied. It is then possible to write $(2+3)*(1+4) \rightarrow 25$ that is the left side evaluates to the right side. It is also possible to evaluate full programs this way although it will give a very large transition system.

5 Semantics for MJava

Since the semantics course on AAU does not teach about types and object oriented language constructs much of the semantics of MJava will be informal. The semantic rules for arithmetic expressions are formal but since it do not consider types it is not fully accurate.

5.1 Environment store model

When talking about formal semantics, the environment-store model has been developed. The model is defined in [Hut04]. This model is used to describe how variables is bound to memory, being that variables is bound to a storage address which contains a value. Using the environment-store model a program state is given by the storage addresses to which variables are bound and the values stored in these addresses. The definition of the variable environment Env_v is:

$$Env_v = Var \cup \{next\} \rightarrow Loc$$

In this definition Var is the set of defined variables in the program. The set Loc is the possible addresses variables can be bound to. Finally the function $next$ is a function that returns the next free location address. This gives that the Env_v is the partial function from variables to storage addresses. The other

part of the variable binding is the *Store* part. This part is the partial function from storage addresses to the values stored within.

$$Store = Loc \rightarrow \mathbb{Q}$$

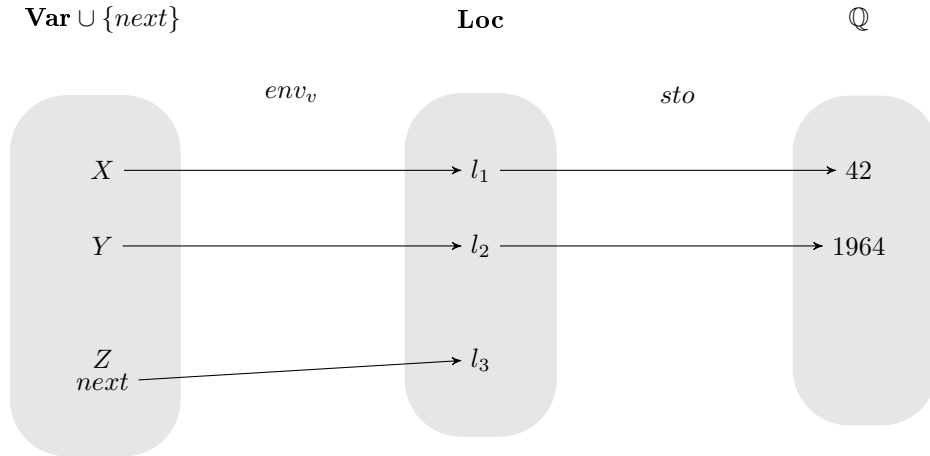


Figure 8: An example on a variable environment and a store from [Hut04] page 92

In figure 8 the variable environment, the locations and the values is shown as the three boxes. The function from the variable environment to the storage locations is called env_v . That is the function that maps from X to location l_1 is called env_v . From locations to values is another function called sto . Looking at the figure it could be the function from l_1 to 42 illustrating that the value 42 is stored in this location. Together env_v and sto describes a program state.

5.2 Arithmetic

The formal semantic for arithmetic expressions are shown in semantics 1.

MJava allows arithmetic expressions to contain method calls and a method call can change the sto . Because of this the language has sideeffects, this is why an evaluation of an arithmetic expression needs to evaluate the components to a value and a new sto . Example: $env_v, env_m \vdash \langle a_1, sto \rangle \rightarrow \langle v_1, sto' \rangle$. The evaluation of a_1 evaluates to v_1 and a new sto .

5.3 Boolean

Boolean expressions are evaluated in much the same way as the arithmetic expressions. It will follow the precedence rules of Java. The boolean operators defined in MJava are:

- $\&\&$ (logical and)
- $!$ (logical not)

First $!$ is evaluated and then $\&\&$ afterwards. Examples are $!true \&\& true \rightarrow false$ and $!false \&\& !false \rightarrow true$.

- [Plus]
$$\frac{env_v \vdash \langle a_1, sto \rangle \rightarrow \langle v_1, sto' \rangle \quad env_v \vdash \langle a_2, sto' \rangle \rightarrow \langle v_2, sto'' \rangle}{env_v \vdash \langle a_1 + a_2, sto \rangle \rightarrow \langle v, sto'' \rangle}$$
where $v = v_1 + v_2$
- [Minus]
$$\frac{env_v \vdash \langle a_1, sto \rangle \rightarrow \langle v_1, sto' \rangle \quad env_v \vdash \langle a_2, sto' \rangle \rightarrow \langle v_2, sto'' \rangle}{env_v \vdash \langle a_1 - a_2, sto \rangle \rightarrow \langle v, sto'' \rangle}$$
where $v = v_1 - v_2$
- [Times]
$$\frac{env_v \vdash \langle a_1, sto \rangle \rightarrow \langle v_1, sto' \rangle \quad env_v \vdash \langle a_2, sto' \rangle \rightarrow \langle v_2, sto'' \rangle}{env_v \vdash \langle a_1 * a_2, sto \rangle \rightarrow \langle v, sto'' \rangle}$$
where $v = v_1 \cdot v_2$
- [Parent]
$$\frac{env_v \vdash \langle a, sto \rangle \rightarrow \langle v, sto' \rangle}{env_v \vdash \langle (a), sto \rangle \rightarrow \langle v, sto' \rangle}$$
- [Num] $env_v, sto \vdash n \rightarrow v$ if $N[n] = v$
- [Var] $env_v, sto \vdash x \rightarrow v$ if $env_v(x) = l$ and $sto(l) = v$

Semantic 1: Aritmentic semantics for MJava

5.4 Commands

A command in MJava is *while*, *if* aso. The commands in MJava works the same way as the commands in Java.

5.5 Classes and extensions

It is possible to declare new classes and extend existings classes but it is not allowed to overload methods. Extending classes works the same way as in Java.

References

- [AWFB00] David A. Watt and Deryck F. Brown. *Programming Language Processors in Java*. Prentice Hall, 2000.
- [Hut04] Hans Huttel. *Pilen ved træets rod*. 2004.
- [pro08a] The MiniJava project. Bnf for minijava. <http://www.cambridge.org/us/features/052182060X/grammar.html>, 2008.
- [pro08b] The MiniJava project. The minijava project. <http://www.cambridge.org/us/features/052182060X/index.html>, 2008.
- [Sip06] Michael Sipser. *Introduction to the Theory of Computation 2.ed*. Thomson course technology, 2006.

[Tob08] Toba. A java-to-c translator. <http://www.cs.arizona.edu/projects/sumatra/toba/>, 2008.