Entwurfsdokument

Simon Bischof, Jan Haag, Adrian Herrmann, Lin Jin, Tobias Schlumberger, Matthias Schnetz

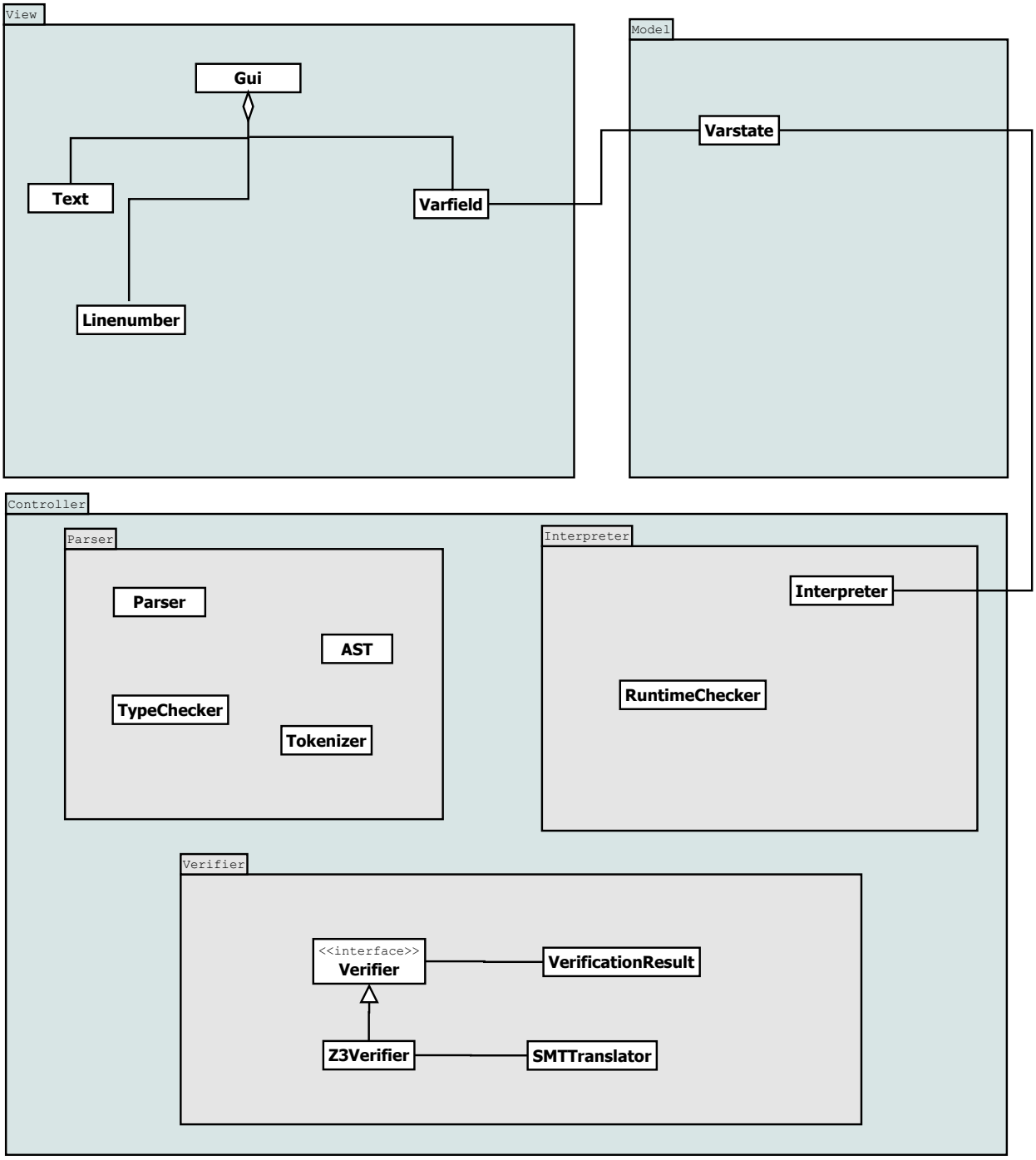
Praxis der Softwareentwicklung
Projekt 3:
Automatisches Prüfen der Korrektheit von Programmen
Gruppe 1

WS 2011/2012

Inhaltsverzeichnis

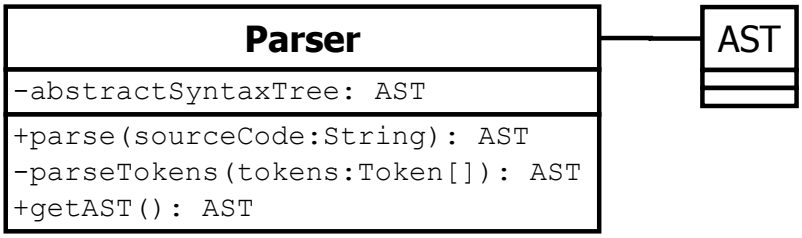
1 Klassendiagramme

1.1 Übersicht

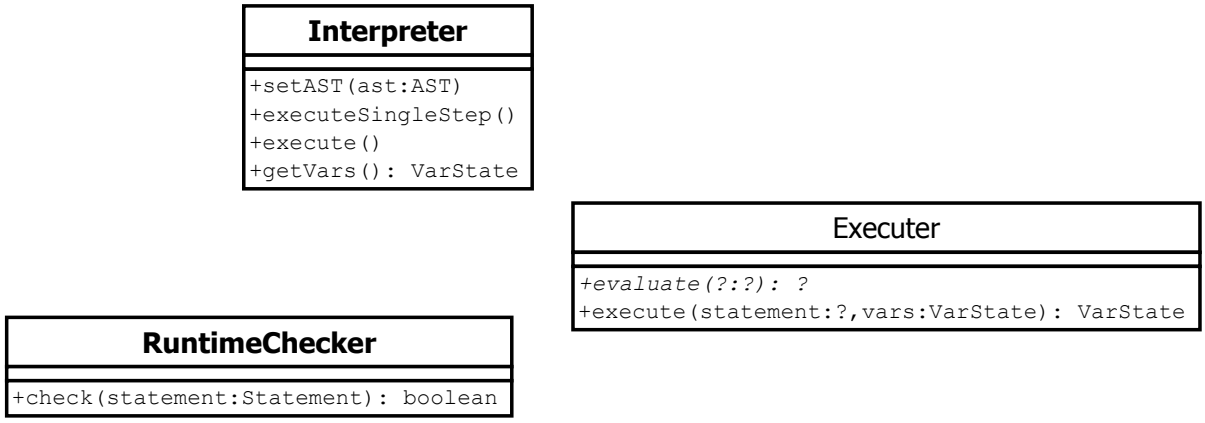


1.2 Feinstruktur der Komponenten

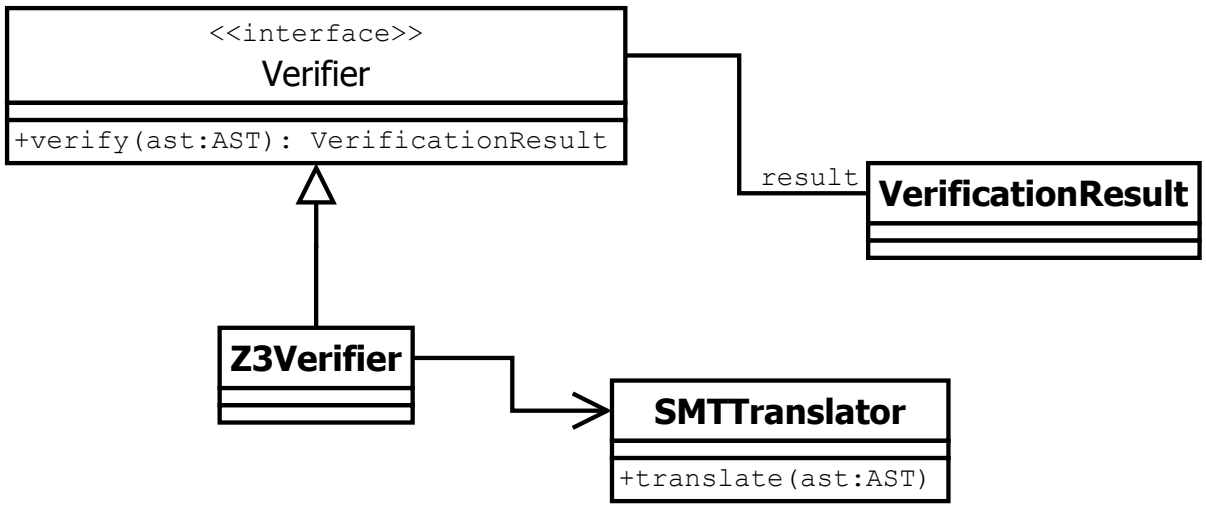
1.2.1 Parser



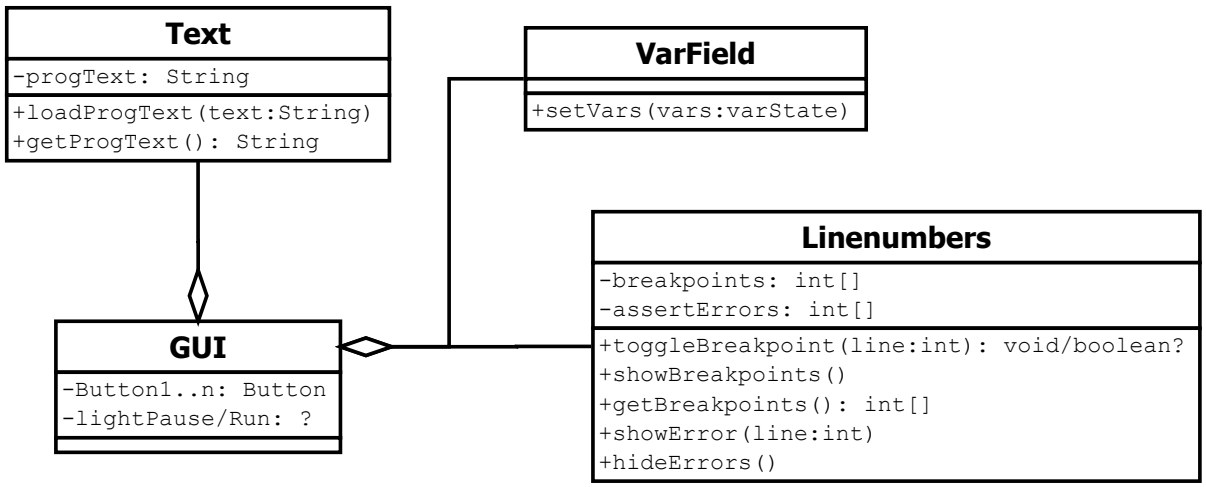
1.2.2 Interpreter



1.2.3 Beweiser



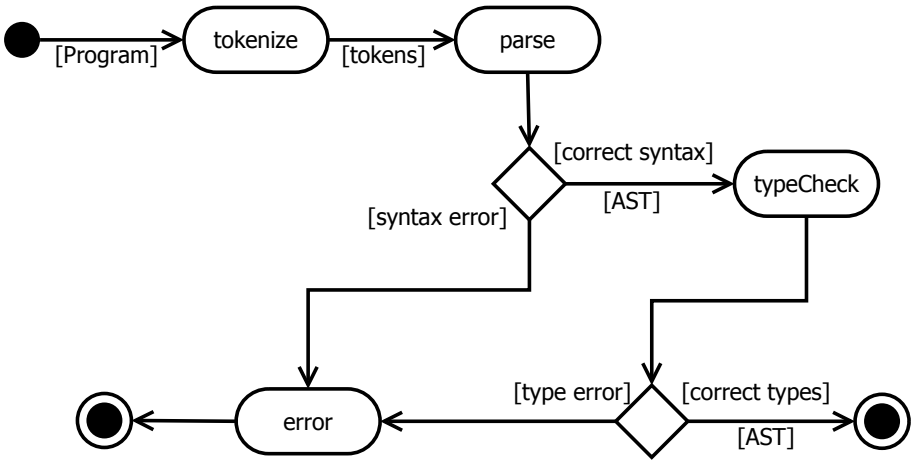
1.2.4 GUI


Text
-progText: String
+loadProgText(text:String)
+getProgText(): String
VarField
+setVars(vars:varState)
Linenumbers
-breakpoints: int[]
-assertErrors: int[]
+toggleBreakpoint(line:int): void/boolean?
+showBreakpoints()
+getBreakpoints(): int[]
+showError(line:int)
+hideErrors()
GUI
-Button1..n: Button
-lightPause/Run: ?

2 Verhaltensdiagramme

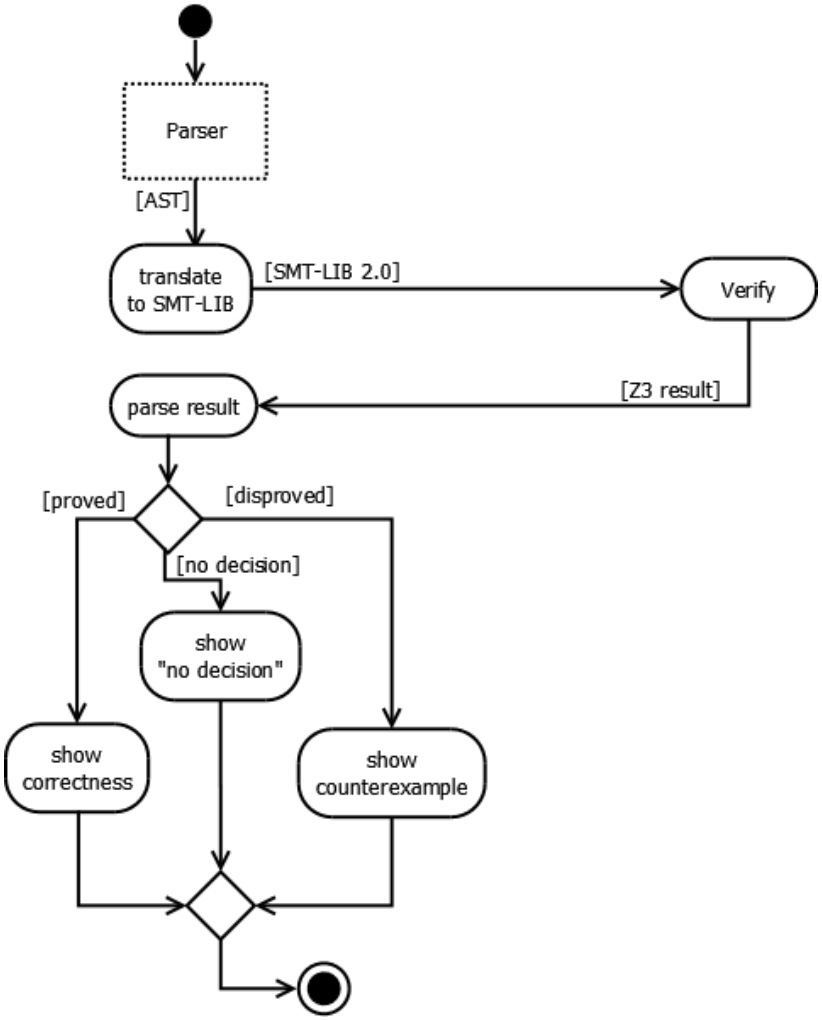
2.1 Aktivitätsdiagramme

2.1.1 Parser/Type-Checker



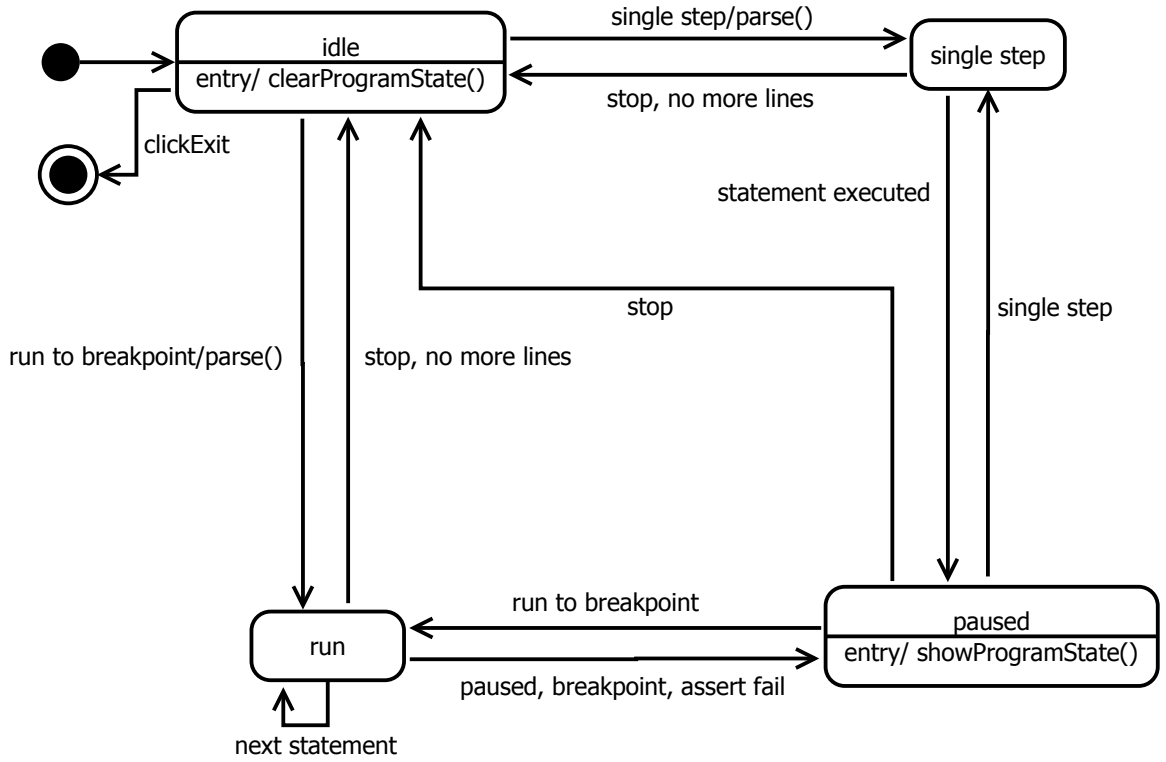
Beim Aufruf des Interpreters wird das Programm in mehreren Schritten geparst.
Der Programmtext wird als einzelner String dem Tokenizer übergeben, welcher den String an den wichtigen Stellen trennt und ein Array von Tokens zurückgibt. Der Parser generiert bei syntaktisch korrekte n Programmen daraus einen abstrakten Syntaxbaum (AST). Bei Syntaxfehlern bricht der Parser mit einem Fehler ab.
Im Erfolgsfall überprüft der Typechecker die Korrektheit der Typen; sind die Typen korrekt, gibt dieser den vom Parser generierten AST zurück, sonst beendet er sich mit einem Fehler.

2.1.2 Z3-Anbindung



Zur Überprüfung der Korrektheit des Programms wird Z3 benutzt.
Zuerst wird das Programm geparst (siehe Aktivitätsdiagramm Parser/Type-Checker). Im Fehlerfall ist keine Überprüfung durch Z3 möglich. Im Erfolgsfall wird der durch den Parser generierte AST an den SMTLib-Translator gegeben. Dieser übersetzt das Programm inklusive Spezifikation ins SMTLib-2.0-Format. Dieses bildet die Eingabe für Z3.
Die von Z3 zurückgegebene Antwort wird vom Result-Parser analysiert. Meldet der Beweiser die Korrektheit des Programms oder konnte er keine Entscheidung treffen, wird dieses Ergebnis dem Benutzer bekannt gegeben. Falls der Beweiser das Programm falsifizieren konnte, wird dem Benutzer das Ergebnis zusammen mit einem möglichen Gegenbeispiel angezeigt.

2.2 Zustandsdiagramm



Beim Starten des Programms geht dieses in den "idle"-Zustand. Hier läuft der Interpreter nicht, und es ist kein Programmzustand gespeichert. Falls einer vorhanden ist, wird dieser beim Eintritt in den "idle"-Zustand gelöscht.
Beim Auswählen von "single step" wird das Userprogramm geparst und ein Statement wird ausgeführt, nachdem der Zustand "single step" betreten worden ist. Ist kein Statement mehr vorhanden, so beendet sich der Interpreter, das Programm geht zurück in den Zustand "idle". Sonst wird nach Ausführen des Statements das Programm pausiert, der Zustand "paused" wird eingenommen. Beim Eintritt in diesen Zustand wird der Zustand des Userprogramms ausgegeben. Während der Pausierung läuft der Interpreter nicht. In diesem Zustand stehen die gleichen Möglichkeiten wie im "idle"-Zustand zu Verfügung, das Parsen bei Verlassen des Zustands entfällt aber.
Wenn im "idle"-Zustand "run to breakpoint" aufgerufen wird, wird das Userprogramm geparst und das Programm geht in den Zustand "run". Das Userprogramm wird solange ausgeführt, bis es zu Ende ist (neuer Zustand: "idle") oder der Interpreter pausiert, ein Breakpoint getroffen oder eine Assertion falsifiziert wird. In diesen Fällen ist der neue Zustand "paused".
In jedem Zustand außer "idle" ist es zusätzlich möglich, das Userprogramm abzubrechen, wobei der Interpreter beendet wird und alle vorhandenen Variablen-Informationen gelöscht werden. Das Programm geht danach in den Zustand "idle".
In jedem Zustand kann das Programm durck einen Klick auf den Exit-Button beendet werden.

3 Syntax der While-Sprache

3.1 Übersicht der Schlüsselwörter und Sonderzeichen

boolean	→ type_specifier
else	→ if_statement
false	→ logical_literal
if	→ if_statement
int	→ type_specifier
return	→ statement
true	→ logical_literal
while	→ while_statement
0..9	→ integer_literal
a..z,A..Z,_	→ identifier
&	→ mul_expression
|	→ add_expression
!	→ unary_expression
!=	→ expression
==	→ expression
<	→ rel_expression
<=	→ rel_expression
>	→ rel_expression
>=	→ rel_expression
+	→ add_expression → unary_expression
-	→ add_expression → unary_expression
*	→ mul_expression
/	→ mul_expression
%	→ mul_expression
,	→ arglist → parameter_list → variable_declaration → variable_initializer
;	→ statement
=	→ variable_declarator → statement
(	→ bracket_expression → if_statement → while_statement → methode_call → methode_declaration
)	→ bracket_expression → if_statement → while_statement → methode_call → methode_declaration
[	→ array_access → type
]	→ array_access → type
{	→ statement_block → variable_initializer
}	→ statement_block → variable_initializer
#	→ comment

3.2 Startsymbol

```
compilation_unit
```

3.3 Produktionsregeln

```
add_expression ::= mul_expression { ( "|" | "+" | "-" ) mul_expression }

arglist ::= expression { "," expression }

array_access ::= identifier "[" expression "]" { "[" expression "]" }

bracket_expression ::= "(" expression ")"
                       | method_call
                       | array_access
                       | identifier
                       | literal_expression

comment ::= "#" .* ( "\n" | "\r" )

compilation_unit ::= { field_declaration }

expression ::= rel_expression { ( "==" | "!=" ) rel_expression }

field_declaration ::= ( [ comment ] ( method_declaration
                                    | statement ) )

identifier ::= "a..z,A..Z,_" { "a..z,A..Z,_,0..9" }

if_statement ::= "if" "(" expression ")" statement_block [ "else" statement_block ]

integer_literal ::= ( "0..9" { "0..9" } )

literal_expression ::= integer_literal
                       | logical_literal

logical_literal ::= "true" | "false"

method_call ::= identifier ( "(" [ arglist ] ")" )

method_declaration ::= type identifier "(" [ parameter_list ] ")" ( statement_block )

mul_expression ::= unary_expression { ( "&" | "*" | "/" | "%" ) unary_expression }

parameter ::= type identifier

parameter_list ::= parameter { "," parameter }

rel_expression ::= add_expression [ ( "<" | "<=" | ">" | ">=" ) add_expression ]

statement ::= variable_declaration ";"
              | identifier "=" variable_initializer ";"
              | ( expression ";" )
              | ( if_statement )
              | ( while_statement )
              | ( "return" [ expression ] ";" )
```

```
statement_block ::= "{" { statement } "}"

type ::= type_specifier { "[" "]" }

type_specifier ::= "boolean" | "int"

unary_expression ::= [ ( "!" | "+" | "-" ) ] bracket_expression

variable_declaration ::= type identifier { "," identifier } [ "=" variable_initializer ]

variable_initializer ::= expression
                       | ( "{" [ variable_initializer { "," variable_initializer } ] "}" )

while_statement ::= "while" "(" expression ")" statement_block
```