PROGRAMMING LANGUAGE SPECIFICATION BY A GRAMMAR WITH CONTEXTS

Mikhail Barash

Turku Centre for Computer Science TUCS, Turku FI-20520, Finland, *and*
Department of Mathematics and Statistics, University of Turku,
Turku FI-20014, Finland, `mikhail.barash@utu.fi`

Abstract
In a recent paper (M. Barash, A. Okhotin, "Defining contexts in context-free grammars", LATA 2012) a mathematical definition of contexts within grammar rules has been proposed. The model, called grammars with contexts*, extends context-free grammars with operators for referring to the context in which a substring being defined occurs. The expressive power of such grammars is far beyond context-free languages, while a general parsing algorithm for such grammars has cubic time complexity, as in standard context-free case. The present paper investigates an application of the model and makes an attempt to specify a tiny typified programming language solely by means of a grammar. Such inherently non-context-free "context conditions" as declaration of every variable before use and checking the types of formal and actual arguments in function calls are specified by* contexts*, native to the proposed model.*

1. Introduction

Since the era of *Algol 60*, the specification of syntax of programming languages has been based on different grammatical models, mainly on *context-free grammars* (rediscovered by Chomsky in 1950s [8]) and so called *Backus-Naur Form* [4], shown to have exactly the same expressive power. For educational purposes Wirth, the author of *Pascal* programming language, has used the third equivalent model—*syntactic diagrams* [12]—in descriptions of languages.

These means allow defining *syntactic structure* of a language, including, for example, correctness of commas separating identifiers in declarations of variables, of parenthesis in nested expressions, of every `if` statement having at most one `else` branch, and so on. However, such seemingly natural restrictions as *declaration of every variable before use* or *agreement of the number* of formal parameters and actual arguments in a function call, not to mention the *agreement of their types*, can not be described by the context-free grammars.

In 1962, Floyd has shown [10] that *Algol* is not a context-free language by considering the following valid program:

$$\texttt{begin real } \underbrace{\texttt{x}\ldots\texttt{x}}_{n} \texttt{ ; } \underbrace{\texttt{x}\ldots\texttt{x}}_{n} \texttt{ := } \underbrace{\texttt{x}\ldots\texttt{x}}_{n} \texttt{ end}$$

Under a certain encoding, one can represent such programs as strings of the form $a^n b^n c^n$, with $n \geqslant 1$, which comprise a well-known non-context-free language.

As a matter of fact, the requirement that every identifier has to be declared before use, can be abstracted by the language $\{\, wcw \mid w \in \Sigma^*,\ c \notin \Sigma \,\}$, called sometimes the *copy language with central marker.* In its turn, the language $\{\, a^n b^m c^n d^m \mid n, m \geqslant 1 \,\}$ abstracts the agreement of numbers of formal parameters and actual arguments in function calls (but not the agreement of their types, though). Both of these languages are known to be non-context-free.

Context-sensitive grammars, initially proposed by Chomsky to describe the syntax of natural languages, turned out to be powerful enough to define the mentioned abstract languages; however, they are equivalent to nondeterministic linear-bounded Turing machines, which essentially means that *no efficient parsing algorithm exists* for such "grammars".

These issues have inspired a subsequent research to develop a *reasonable* (that is, efficiently parsable) grammatical model, which would be able to specify the mentioned abstract languages. Of many such attempts, one could emphasize *indexed grammars* by Aho [1] and *two-level grammars* by van Wijngaarden [28]. Unfortunately, one has never found a *polynomial* parsing algorithm for the former model, while the latter has been shown computationally universal, making it practically irrelevant. Nonetheless, two-level grammars have been used in specification of *Algol 68* programming language [28], with all "context conditions" being defined formally.

One can notice that the language $\{\, a^n b^m c^n d^m \mid n, m \geqslant 1 \,\}$ can be represented as intersection of two context-free languages, which means, in addition, that it can be parsed in polynomial time by invoking the appropriate context-free grammar parsing algorithm. The generalization of this idea pursued to *Boolean closure of context-free languages*, for which polynomial-time parsing algorithms have been developed (see, for example, the paper by Heilbrunner and Schmitz [11]). However, as it has been later shown by Wotschke [29], the above-mentioned copy language with central marker is not representable as a finite intersection of context-free languages, and thus some other apt formalisms had to be discovered.

Meanwhile, the development of fast polynomial parsing algorithms for subclasses of context-free grammars [14, 15], as well as the need to specify the syntax of programming languages by models as natural as context-free grammars, led to research on *attribute grammars* [16] and *syntax-directed translation* [2]. Every rule is associated with *semantic actions*, which, for example, may use *symbol tables* to look-up whether a certain variable has been previously declared. Such essentially *ad hoc* techniques are used even until now (see, for example, the *Bison* parser generator).

The consideration of context-free grammars as rewriting systems, imposed by Chomsky [8], concealed the essence of them as a *logic for defining syntax* [26]. In this logic there is only one set-theoretic operation available—*disjunction*—which expresses *union of languages.* Having two rules $A \to B$ and $A \to C$ in a grammar means the a string w has the property A if it has property B or property C. The missing desired *intersection of languages* is allowed in the bodies of the rules of *conjunctive grammars*, introduced in 2001 by Okhotin [17]. In such a grammar, a rule, say, $A \to B \,\&\, C$ states that a string w has property A if it has *both* the properties B *and* C. Further augmentation of the model led to *Boolean grammars* [19], where negation is also allowed in the rules ($A \to B \,\&\, \neg C$). It turned out that already conjunctive grammars are able to specify the copy language with central marker, as well as many other nontrivial languages [13, 17, 20, 25]. The main parsing algorithms for context-free grammars have been generalized to the case of conjunctive and Boolean grammars [17, 18, 19, 22, 23, 24] and the time complexity has been preserved.

Apparently the first specification of a programming language entirely by a formal grammar has been made by Okhotin [21], who constructed a Boolean grammar for a certain simple programming language. That language, however, had only one data type and hence no approach of how to implement type checking has been suggested.

This paper specifies in Section 3. a very similar language named *Kieli* (Finnish for "*language*"), augments it with two data types (*integer* and *Boolean*), and uses another formalism—*grammars with contexts*—to specify the language in a formal way.

Grammars with contexts [7, 6], defined in Section 2., extend conjunctive grammars with operators for referring to the *context* (either left or right), in which a substring being defined occurs. Thus, for example, a rule $A \to a \,\&\, \lhd B$ defines a substring a if it is preceded by a string of the form B. In relation to a grammar for a programming language, it becomes very natural to check that some variable has been previously declared by just expressing the fact that "*there has been* an identifier with the same name". The names of the two identifiers in question can be still checked by a kind of the copy language with central marker [21], representable by a conjunctive grammar.

Recent research has established some basic properties of grammars with contexts, including a cubic-time general parsing algorithm [7]. Therefore, having specified a programming language by a grammar with contexts, which is done in Section 4., one immediately gets a polynomial-time parsing algorithm for the set of valid programs in this language. Other properties of grammars with contexts include square-time unambiguous parsing algorithm [6], linear-space recognition algorithm [6], as well as linear time recursive descent algorithm [5].

2. Grammars with contexts

Definition 2.1 ([7, 6]) *A grammar with left contexts is a quadruple $G = (\Sigma, N, R, S)$, where*

- *Σ is the alphabet of the language being defined;*
- *N is a finite set of auxiliary symbols, disjoint with Σ, which denote properties of strings defined in the grammar;*
- *R is a finite set of rules, each of the form*

$$A \to \alpha_1 \,\&\, \ldots \,\&\, \alpha_k \,\&\, \lhd\beta_1 \,\&\, \ldots \,\&\, \lhd\beta_m \,\&\, \trianglelefteq\gamma_1 \,\&\, \ldots \,\&\, \trianglelefteq\gamma_n, \qquad (1)$$

with $A \in N$, $\alpha_i, \beta_i, \gamma_i \in (\Sigma \cup N)^*$;
- *$S \in N$ is a symbol representing correct sentences ("start symbol").*

For every grammar rule (1), a term α_i, $\lhd\beta_i$ or $\trianglelefteqslant\gamma_i$ is called a *conjunct.* Each conjunct α_i gives a representation of the string being defined. A conjunct $\lhd\beta_i$ describes a form of the *left context* or *the past* of the substring. Conjuncts $\trianglelefteqslant\gamma_i$ refer the form of the current substring and its left context, concatenated into a single string, so called *extended left contexts.*

Consider a substring $v \in \Sigma^*$ preceded by $u \in \Sigma^*$ and denote it as $u\langle v\rangle$. Intuitively, such a substring is generated by a rule (1), if

- v can be represented as a concatenation of shorter substrings $X_1, \ldots, X_\ell$, for every conjunct $\alpha_i = X_1 \ldots X_\ell$;
- u can be similarly described by every conjunct $\lhd\beta_i$ and uv by every conjunct $\trianglelefteqslant\gamma_j$.

The semantics of grammars with left contexts can be defined in two equivalent ways [7, 6]. One of them is by *logical deduction of elementary propositions* (items) of the form *"a string $v \in \Sigma^*$ written in the left context $u \in \Sigma^*$ has the property $\alpha \in (\Sigma \cup N)^*$"*, denoted as $[\alpha,\ u\langle v\rangle]$. An alternative definition of grammar with contexts uses a generalization of *language equations*, where unknowns are *languages of pairs* of a string and its left context.

Definition 2.2 ([7, 6]) *Let $G = (\Sigma, N, R, S)$ be a grammar with left contexts. Define the following system of items of the form $[X,\ u\langle v\rangle]$, with $X \in \Sigma \cup N$ and $u, v \in \Sigma^*$ as follows. There is a single axiom scheme $\vdash_G [a,\ x\langle a\rangle]$, for all $a \in \Sigma$ and $x \in \Sigma^*$. Each rule (1) in the grammar defines a scheme $I \vdash_G [A,\ u\langle v\rangle]$ for deduction rules, for all $u, v \in \Sigma^*$ and for every set of items I satisfying the below properties:*

- *for every conjunct $\alpha_i = X_1 \ldots X_\ell$, $\ell \geqslant 0$ and $X_j \in \Sigma \cup N$, there should exist a partition $v = v_1 \ldots v_\ell$ with $[X_j,\ uv_1 \ldots v_{j-1}\langle v_j\rangle] \in I$ for all $j \in \{1, \ldots, \ell\}$;*
- *for every conjunct $\lhd\beta_i = \lhd X_1 \ldots X_\ell$ there should be such a partition $u = u_1 \ldots u_\ell$, that $[X_j,\ u_1 \ldots u_{j-1}\langle u_j\rangle] \in I$ for all $j \in \{1, \ldots, \ell\}$;*
- *every conjunct $\trianglelefteqslant\gamma_i = \trianglelefteqslant X_1 \ldots X_\ell$ should have a corresponding partition $uv = x_1 \ldots x_\ell$ with $[X_j,\ x_1 \ldots x_{j-1}\langle x_j\rangle] \in I$ for all $j \in \{1, \ldots, \ell\}$.*

The language generated by a nonterminal symbol A is defined as $L_G(A) = \{\, u\langle v\rangle \mid u, v \in \Sigma^*, \vdash_G [A,\ w\langle v\rangle] \,\}$. The language generated by the grammar G is the set of all strings with an empty left context generated by S: $L_G(S) = \{\, w \mid w \in \Sigma^*, \vdash_G [S,\ \varepsilon\langle w\rangle] \,\}$.

This definition can be illustrated by the following trivial example.

Example 2.3 *The following grammar generates a singleton language $\{ac\}$:*

$$\begin{aligned} S &\to aC \mid bC \\ C &\to c \,\&\, \lhd A \\ A &\to a \end{aligned}$$

Consider the following logical derivation of the fact that the string ac with left context ε is defined by S:

$$\begin{array}{rl}
\vdash [a,\ \varepsilon\langle a\rangle] & (axiom) \\
\vdash [c,\ a\langle c\rangle] & (axiom) \\
[a,\ \varepsilon\langle a\rangle] \vdash [A,\ \varepsilon\langle a\rangle] & (A \to a) \\
[c,\ a\langle c\rangle], [A,\ \varepsilon\langle a\rangle] \vdash [C,\ a\langle c\rangle] & (C \to c \,\&\, \lhd A) \\
[a,\ \varepsilon\langle a\rangle], [C,\ a\langle c\rangle] \vdash [S,\ \varepsilon\langle ac\rangle] & (S \to aC)
\end{array}$$

Example 2.4 ([6]) *The following grammar defines the language*

$$\{\, u_1 \dots u_\ell \mid \text{for every } i,\ u_i \in a^*c \text{ or there exist } j,k \text{ with } j < i,\ u_i = b^k c \text{ and } u_j = a^k c \,\} :$$

$$\begin{array}{ll}
S \to AS \mid CS \mid \varepsilon & C \to B \,\&\, \trianglelefteq EFc \\
A \to aA \mid c & E \to AE \mid BE \mid \varepsilon \\
B \to bB \mid c & F \to aFb \mid cE
\end{array}$$

This language abstracts declaration of identifiers (represented in unary notation) before their use. Substrings of the form $a^k c$ stand for declarations, while every substring $b^k c$ is a reference to a declaration of the form $a^k c$. The condition that S generates a string $u_1 \dots u_\ell \langle u_{\ell+1} \dots u_n \rangle$ (with $u_i \in a^*c \cup b^*c$) only if every reference in the suffix $u_{\ell+1} \dots u_n$ has a corresponding declaration in the prefix $u_1 \dots u_\ell$ is checked inductively on ℓ. The base case is given by the rule $S \to \varepsilon$: the string $u_1 \dots u_n \langle \varepsilon \rangle$ has the desired property. The rule $S \to AS$ appends a declaration (a^*c), while a reference $\Big(b^*c \,\&\, \trianglelefteq \Big((\{a,b\}c)^* \underbrace{a^n}_{} c (\{a,b\}c)^* \underbrace{b^n}_{} c \Big)\Big)$ can be added by rules $S \to CS$ and $C \to B \,\&\, \trianglelefteq EFc$. The extended left context in the latter rule checks that b^*c has a corresponding earlier declaration: E "skips" the symbols in the prefix up to some declaration, and then F matches the symbols a in the declaration to the symbols b in the reference.

To generalize this example to identifiers over a many-letter alphabet, the technique for matching the symbols in a declaration and a reference (cf. nonterminal F) shall be much more involved than just checking the number of the appropriate a's and b's.

Example 2.5 ([17]) *The following conjunctive grammar generates the language* $\{\, wcw \mid w \in \{a,b\}^* \,\}$*:*

$$\begin{array}{ll}
S \to C \,\&\, D & \\
A \to XAX \mid cEa & C \to XCX \mid c \\
B \to XBX \mid cEb & D \to aA \,\&\, aD \mid bB \,\&\, bD \mid cE \\
X \to a \mid b & E \to XE \mid \varepsilon
\end{array}$$

The nonterminal C defines the strings consisting of two parts of equal length separated by a central marker, that is, $L(C) = \{\, w_1 c w_2 \mid w_1, w_2 \in \{a,b\}^*, |w_1| = |w_2| \,\}$. The actual comparison of symbols in corresponding positions is made by nonterminal D, which defines the language $L(D) = \{\, uczu \mid u, z \in \{a,b\}^* \,\}$. The conjunction of C and D ensures that the strings generated by the grammar are of the desired form.

The language $L(D)$ is defined inductively as follows. The base case gives strings of the form $c\{a, b\}^*$, that is, with $u = \varepsilon$. The induction ensures that a string is in $L(D)$ if and only if its first symbol is the same as the corresponding symbol on the other side, and the string without this first symbol is also in $L(D)$.

A similar language shall be used later in Section 4., when a grammar for a programming language is defined.

3. Definition of the language

The complete definition of the programming language *Kieli* is presented below.

[I] Lexical conventions.

- **[I.1]** *Alphabet of the language.* A program is a finite string over the following alphabet of: *letters* (`a`, …, `z`), *digits* (`0`, …, `9`), *punctuators* (`(`, `)`, `{`, `}`, `+`, `-`, `*`, `/`, `&`, `|`, `!`, `=`, `<`, `>`, `,`, `;`), and a *whitespace character* (␣).
- **[I.2]** *Lexemes*, the set of the following finite strings over the alphabet:
  - **[I.2.1]** *Keywords*: `int`, `bool`, `if`, `else`, `while`, `return`, `void`, `true`, `false`.
  - **[I.2.2]** *Identifiers*:
    - **[I.2.2.1]** Identifier is a finite nonempty sequence of letters and digits, starting from a letter.
    - **[I.2.2.2]** Identifier should not coincide with any of the keywords.
  - **[I.2.3]** *Number* is a finite nonempty sequence of digits.

[II] Syntactic structure.

- **[II.1]** A *program* is a finite sequence of function declarations, containing exactly one main function.
- **[II.2]** *Function:*
  - **[II.2.1]** For every identifier x, not declared before as a name of a parameter of the current function, `int` x and `bool` x are *declarations of a function parameter.*
  - **[II.2.2]** For every identifier f, not used as a name of a function before, for every keyword $t \in \{\texttt{int}, \texttt{bool}, \texttt{void}\}$, and for every set a_1, …, a_n (with $n \geqslant 0$) of declarations of a function parameter, t f `(` a_1 `,` … `,` a_n `)` is a *function header.* The identifier f is the *name* of the function, while a_1, …, a_n are *parameters* of the function.
  - **[II.2.3]** For every identifier x, `int main (` `int` x `)` is a *main function header.*
  - **[II.2.4]** For every function header F and for every properly returning statement s, $F\{s\}$ is a *function description.*
- **[II.3]** *Expression:*
  - **[II.3.1]** *Integer expression:*
    - **[II.3.1.1]** A number is an integer expression.
    - **[II.3.1.2]** An identifier x, declared either in integer variable declaration or in integer parameter declaration, is an integer expression.

[II.3.1.3] For every integer expression e, `(` e `)` is an integer expression.

[II.3.1.4] For two integer expressions e_1 and e_2 and for binary operator $op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}$, e_1 op e_2 is an integer expression.

[II.3.1.5] For every integer expression e and for unary operator $op \in \{\texttt{-}\}$, op e is an integer expression.

[II.3.1.6] For every identifier f, declared as a name of a function returning an integer value, and for every set of expressions $e_1, \ldots, e_n$ (with $n \geqslant 0$), the *integer function call* f `(` e_1 `,` $\ldots$ `,` e_n `)` is an integer expression.

- The number of *actual arguments* $\{e_1, \ldots, e_n\}$ must coincide with the number of *formal parameters* in the declaration of the function f.
- The types of expressions $e_1, \ldots, e_n$ must be the same as the types of the corresponding formal parameters of the function f.

[II.3.2] *Boolean expression:*

[II.3.2.1] A Boolean value is a Boolean expression.

[II.3.2.2] An identifier x, declared either in Boolean variable declaration or in Boolean parameter declaration, is a Boolean expression.

[II.3.2.3] For two integer expressions e_1 and e_2 and for binary operator $op \in \{\texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}, \texttt{<>}, \texttt{==}\}$, e_1 op e_2 is a Boolean expression.

[II.3.2.4] For two Boolean expressions e_1 and e_2 and for binary operator $op \in \{\texttt{<>}, \texttt{==}\}$, e_1 op e_2 is a Boolean expression.

[II.3.2.5] For every Boolean expression e and for unary operator $op \in \{\texttt{!}\}$, op e is a Boolean expression.

[II.3.2.6] *Boolean function call* is defined analogously to *integer function call.*

[II.4] *Statements:*

[II.4.1] *Declaration statement:* For every set of identifiers $x_1, \ldots, x_n$ (with $n \geqslant 1$), not declared before as names of variables or parameters within the current scope, and for every keyword $t \in \{\texttt{int}, \texttt{bool}\}$, the *variables declaration statement* t x_1 `,` $\ldots$ `,` x_n `;` is a statement.

[II.4.2] For every expression e, the *expression statement* e`;` is a statement.

[II.4.3] *Void function call* is a statement and is defined similarly to *integer function call.*

[II.4.4] For every identifier x, declared either in integer (Boolean) variable declaration or in integer (Boolean, respectively) parameter declaration, and for every integer (Boolean, respectively) expression e, the *assignment statement* x `=` e `;` is a statement.

[II.4.5] *Conditional statement:* For every Boolean expression e, and for every properly returning statements s_1 and s_2, the *if-statement* `if (` e `)` $\{s_1\}$ and the *if-then-else statement* `if (` e `)` $\{s_1\}$ `else` $\{s_2\}$ are statements.

[II.4.6] *Iteration statement:* For every Boolean expression e and for every properly returning statement s, the *while-statement* `while (` e `)` $\{s\}$ is a statement.

[II.4.7] For every set of statements $s_1, \ldots, s_n$ (with $n \geqslant 0$), the *compound statement* $\{s_1 s_2 \ldots s_n\}$ is a statement.

[II.5] *Properly returning statements:*

[**II.5.1**] For every expression e, the *return statement* `return` e `;` is a *properly returning statement.* The type of the expression e must be the same as the type of the function, within which the return statement is used.

[**II.5.2**] Every statement within a void function is a *properly returning statement.*

[**II.6**] *Scopes of visibility*:

[**II.6.1**] The scope of a declaration of a formal parameter of a function is the body of that function.

[**II.6.2**] The scope of a variable declaration located within a compound statement covers all statements in this compound statement, starting from the declaration statement itself.

[**II.6.3**] The scopes of visibility for any two identifiers of the same kind sharing the same name must be disjoint.

4. Grammar with contexts for *Kieli*

Verbatim to the specification, a program written in the language *Kieli* is a sequence of function declarations, among which exactly one main function has to be present:

S → **Functions MainFunction Functions**
Functions → **Function Functions** | ε

A function declaration consists of a header of a function followed by a set of statements:

Function → **FunctionHeaderDeclaration** tOpenBrace **Statements** tCloseBrace
MainFunction →
    MainFunctionHeaderDeclaration tOpenBrace **Statements** tCloseBrace
FunctionHeaderDeclaration →
    IntegerFunctionHeaderDeclaration |
    BooleanFunctionHeaderDeclaration |
    VoidFunctionHeaderDeclaration

Depending on the type of a value the function returns, each of the nonterminals **IntegerFunctionHeaderDeclaration**, **BooleanFunctionHeaderDeclaration**, and **VoidFunctionHeaderDeclaration** define a function header as a corresponding type keyword (or `void`) followed by a signature of a function, that is, its name and list of its formal parameters:

IntegerFunctionHeaderDeclaration → KeywordInt ␣ **FunctionSignature**
BooleanFunctionHeaderDeclaration → KeywordBool ␣ **FunctionSignature**
VoidFunctionHeaderDeclaration → KeywordVoid ␣ **FunctionSignature**
FunctionSignature → Identifier
    tOpenParenthesis **ParametersOrNoParameters** tCloseParenthesis

A function can, in general, have no parameters at all:

ParametersOrNoParameters → **Parameters** | ε

In a list of parameters, every parameter is separated by a comma:

Parameters → **Parameter** tComma **Parameters** | **Parameter**

A declaration of a function parameter should specify the type of the parameter and its name:

Parameter → TypeKeyword ␣ Identifier WS

Main function has to return an integer value and have exactly one integer parameter:

MainFunctionHeaderDeclaration →
KeywordInt ␣ `m a i n` WS tOpenParenthesis
KeywordInt ␣ Identifier tCloseParenthesis

A body of every function contains (a possibly empty) set of statements:

StatementsOrNoStatements → **Statements** | ε
Statements → **Statement Statements** | **ProperlyReturningStatement**

A statement is defined as follows:

Statement → **VariableDeclaration** | **ExpressionStatement** |
VoidFunctionCall | **AssignmentStatement** |
IfStatement | **WhileStatement** |
tOpenBrace **Statements** tCloseBrace

The keywords and the white space are defined as follows:

KeywordInt → `i n t` WS
$\vdots$
KeywordFalse → `f a l s e` WS
TypeKeyword → KeywordInt | KeywordBool
WS → ␣ WS | ε

An identifier is defined exactly to its specification:

Identifier → Letter LettersOrDigits WS & NotKeyword WS
NotKeyword → NotKeywordInt & ... & NotKeywordFalse

Every nonterminal NotKeywordInt, ..., NotKeywordFalse defines a corresponding regular language $\Sigma^* \setminus \{\texttt{int}\}, \dots, \Sigma^* \setminus \{\texttt{false}\}$.

A declaration of variables starts with the keyword specifying their type, followed by a list of “invalid” identifiers:

VariableDeclaration →
TypeKeyword ␣ **InvalidIdentifiers** ***InvalidIdentifier*** tSemicolon
InvalidIdentifiers → ***InvalidIdentifier*** tComma **InvalidIdentifiers** | ε

The rules for the nonterminal ***InvalidIdentifier*** shall be explained later.

Expressions can be either integer or Boolean:

Expression $\to$ **IntegerExpression** | **BooleanExpression**

Integer and Boolean expressions are, as in a standard context-free grammar, defined inductively on their structure as follows:

IntegerExpression $\to$
  IntegerConstant |
  ValidIntegerIdentifier |
  tOpenParenthesis **IntegerExpression** tCloseParenthesis |
  IntegerExpression IntegerBinaryOperation **IntegerExpression** |
  IntegerUnaryOperation **IntegerExpression** |
  IntegerFunctionCall
BooleanExpression $\to$
  KeywordTrue | KeywordFalse |
  ValidBooleanIdentifier |
  tOpenParenthesis **BooleanExpression** tCloseParenthesis |
  BooleanExpression BooleanBinaryOperation **BooleanExpression** |
  IntegerExpression BooleanBinaryOperation **IntegerExpression** |
  BooleanUnaryOperation **BooleanExpression** |
  BooleanFunctionCall

The nonterminal ***ValidIntegerIdentifier*** (***ValidBooleanIdentifier***) defines an identifier that has been previously declared either as an integer (Boolean, respectively) parameter of a function or as an integer (Boolean, respectively) variable inside a function, with the scopes of visibility respected.

The condition of being previously declared can be naturally expressed by an extended left context:

ValidIntegerIdentifier $\to$
  Identifier & $\trianglelefteq$ **Functions** *anything-except-function-header*[1]
    KeywordInt ␣ **Identifiers** C
ValidBooleanIdentifier $\to$
  Identifier & $\trianglelefteq$ **Functions** *anything-except-function-header*
    KeywordBool ␣ **Identifiers** C

First, the nonterminal **Functions** "skips" the prefix of the program until a declaration of some function starts. Then, the keyword `int` (or `bool`) and a list of identifiers is matched. Finally, the nonterminal C [21] is used to compare the identifier in applicative use to the identifier in declarative use:

[1]The rules for nonterminals written in *slanted font* define simple regular languages that speak for themselves and are omitted due to the space constraints.

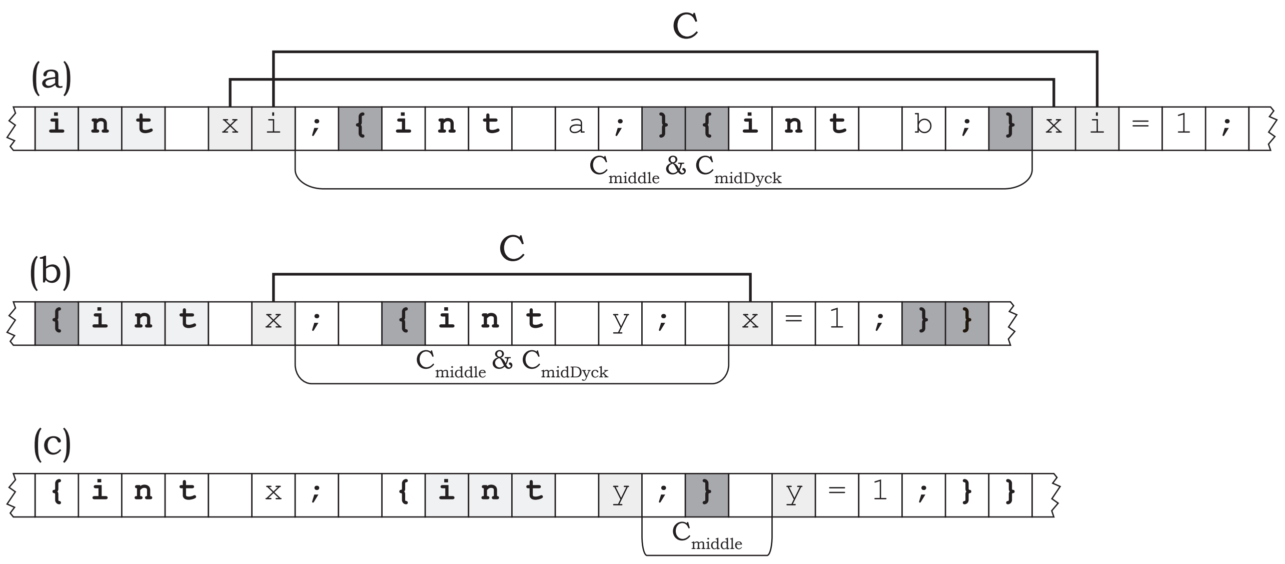
C
(a)
int xi; {int a; } {int b; } xi = 1;
C_{middle} & $C_{midDyck}$
C
(b)
{int x; {int y; x = 1; } }
C_{middle} & $C_{midDyck}$
(c)
{int x; {int y; } y = 1; } }
C_{middle}

Figure 1:: How nonterminal C compares names of identifiers and scopes of visibility.

C $\to$ CLEN & CITERATE | C WS
CLEN $\to$ LetterOrDigit CLEN LetterOrDigit | LetterOrDigit CMID LetterOrDigit
CA $\to$ LetterOrDigit CA LetterOrDigit | `a` LettersOrDigits CMID
$\vdots$
C9 $\to$ LetterOrDigit C9 LetterOrDigit | `9` LettersOrDigits CMID
CITERATE $\to$
    CA `a` & CITERATE `a` | $\cdots$ | CZ `z` & CITERATE `z` |
    C0 `0` & CITERATE `0` | $\cdots$ | C9 `9` & CITERATE `9` |
    LettersOrDigits CMID
CMID $\to$ CMIDDLE & CMIDDYCK
CMIDDLE $\to$ *any-punctuator any-string any-punctuator* |
    ␣ *any-string any-punctuator* | *any-punctuator any-string* ␣ |
    ␣ *any-string* ␣ | *any-punctuator* | ␣

The scopes of visibility are checked by the nonterminal CMIDDYCK, which defines the language

$$\{\, w \in \Sigma^* \mid \text{braces in } w \text{ are balanced, with possibly some open braces left unmatched} \,\} :$$

CMIDDYCK $\to$
    anything-except-braces | CMIDDYCK *anything-except-braces* CMIDDYCK |
    tOpenBrace *anything-except-braces* tCloseBrace |
    tOpenBrace *anything-except-braces* CMIDDYCK *anything-except-braces* tClose-
Brace | tOpenBrace

This allows one to correctly process scopes of visibility, as shown in Figure 1.

Now when the nonterminals ***ValidIntegerIdentifier*** and ***ValidBooleanIdentifier*** have been defined, the rule for the nonterminal ***InvalidIdentifier*** can be given:

> ***InvalidIdentifier*** → Identifier & ¬ ***ValidIdentifier***
> ***ValidIdentifier*** → ***ValidIntegerIdentifier*** | ***ValidBooleanIdentifier***

Though the negation is not defined within the formalism of grammars with contexts, here its use is particularly handy for shortness of notation. It is worth noticing, that negation here *is not necessary at all*, and one can get rid of it by using the following rule for ***InvalidIdentifier***:

> ***InvalidIdentifier*** → Identifier & $\trianglelefteqslant$ **Functions**
> *anything-except-function-header* KeywordInt ␣ **Identifiers** D,

where the nonterminal D would define the conjunctive language

$\{w_1 x_1 w_2 x_2 \ldots w_\ell x_\ell w_{\ell+1} \mid x_i$ start and end with anything but letters or digits and either all w_i are different *or* some of them are equal but at least one x_i contains more "}"-braces than "{"-braces$\}$.

Calls to functions require checking the name of the function in question, as well as agreement of formal parameters and actual arguments types:

> ***ValidFunctionNameAndArguments*** →
> C tOpenParenthesis **Expressions** tCloseParenthesis &
> Identifier tOpenParenthesis Q tCloseParenthesis
> **Expressions** → **Expressions** tComma **Expression** | **Expression**
> ***IntegerFunctionCall*** →
> Identifier tOpenParenthesis **Expressions** tCloseParenthesis & $\trianglelefteqslant$**Functions**
> KeywordInt ␣ ***ValidFunctionNameAndArguments***
> ***BooleanFunctionCall*** →
> Identifier tOpenParenthesis **Expressions** tCloseParenthesis & $\trianglelefteqslant$**Functions**
> KeywordBool ␣ ***ValidFunctionNameAndArguments***

The first conjunct in the rule for nonterminal ***ValidFunctionNameAndArguments*** uses nonterminal C to check that the function name has been previously declared. The comparison of formal parameters and actual arguments of a function is done by the second conjunct of that rule, which uses the nonterminal Q, as shown in Figure 2.

> Q → QLEN & QITERATE | Q WS
> QLEN → TypeKeyword ␣ Identifier QLEN **Expression** |
> tComma QLEN tComma | QMID
> INTVARORЕXPR → KeywordInt ␣ Identifier | **IntegerExpression**
> BOOLVARORЕXPR → KeywordBool ␣ Identifier | **BooleanExpression**
> QX → INTVARORЕXPR | BOOLVARORЕXPR | tComma
> QY → QX QY | ε

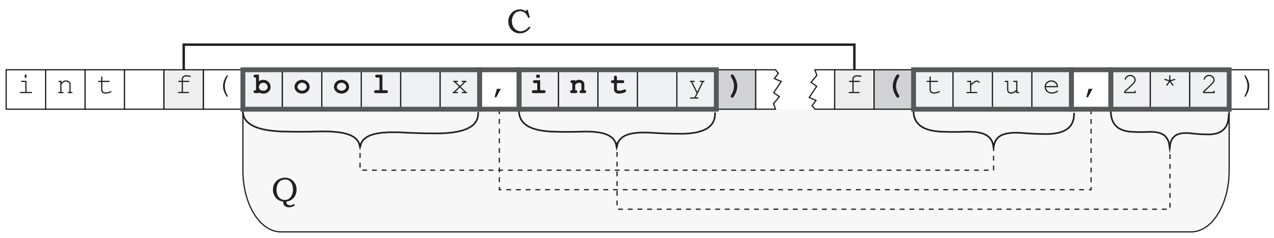

Figure 2:: How the agreement of formal parameters and actual arguments in a function call is checked by nonterminal ***ValidFunctionNameAndArguments***.

QINT → QX QINT QX | INTVARORE XPR QY QMID
QBOOL → QX QBOOL QX | BOOLVARORE XPR QY QMID
QCOMMA → QX QCOMMA QX | tComma QY QMID
QITERATE →
  QINT INTVARORE XPR & QITERATE INTVARORE XPR |
  QBOOL BOOLVARORE XPR & QITERATE BOOLVARORE XPR |
  QCOMMA tComma & QITERATE tComma | QY QMID
QMID → tCloseParenthesis *anything* tOpenParenthesis

An expression statement is defined as an expression followed by a semicolon:

ExpressionStatement → **Expression** WS tSemicolon

Call to a void function is defined in a way similar to integer and Boolean function calls:

VoidFunctionCall →
  Identifier tOpenParenthesis **Expressions** tCloseParenthesis & ⊲**Functions** KeywordVoid ␣ ***ValidFunctionNameAndArguments***

An assignment statement respects the types of the variable being assigned and of the expression in the right-hand side part of the statement:

AssignmentStatement →
  ValidIntegerIdentifier tEqual **IntegerExpression** tSemicolon |
  ValidBooleanIdentifier tEqual **BooleanExpression** tSemicolon

Now conditional and iteration statements can be defined in a natural way, explicitly stating that the conditions should be of Boolean type:

IfStatement → KeywordIf tOpenParenthesis **BooleanExpression** WS tCloseParenthesis tOpenBrace **Statements** tCloseBrace **OptionalElse** WS
OptionalElse → KeywordElse tOpenBrace **Statements** tCloseBrace WS | ε
WhileStatement → KeywordWhile tOpenParenthesis **BooleanExpression** WS tCloseParenthesis tOpenBrace **Statements** tCloseBrace WS

Every compound statement in a program must be properly returning. If the function has been declared as of integer or Boolean type, its body must explicitly return a value in a return statement:

$$\begin{aligned}
&\textbf{\textit{ProperlyReturningStatement}} \to \\
&\qquad \text{KeywordReturn } \textbf{IntegerExpression} \text{ WS tSemicolon } \& \ \lhd\textbf{Functions} \\
&\qquad\qquad \text{KeywordInt } \textit{anything-except-function-header} \mid \\
&\qquad \text{KeywordReturn } \textbf{BooleanExpression} \text{ WS tSemicolon } \& \ \lhd\textbf{Functions} \\
&\qquad\qquad \text{KeywordBool } \textit{anything-except-function-header}
\end{aligned}$$

If the function has been declared as void, no explicit return statement is needed, and every statement can be deemed as properly returning:

$$\begin{aligned}
&\textbf{\textit{ProperlyReturningStatement}} \to \textbf{Statement} \ \& \ \lhd\textbf{Functions} \\
&\qquad\qquad \text{KeywordVoid } \textit{anything-except-function-header}
\end{aligned}$$

5. Conclusions and future work

An application of grammars with contexts to formal definition of syntax of programming languages has been thus studied. Type checking is carried out apparently for the first time by a formal grammar constructed within the lines of the present paper. It is worth noticing that if all context operators *were* removed from the grammar, it would still define the "syntactic structure" of programs, with no "semantic checks" done.

The existing general parsing algorithms, applicable to grammars with contexts, require a grammar to be transformed to a certain normal form [7, 6, 3], and such a transformation leads, in the worst case, to a triple exponential blow-up of size of a grammar. Moreover, the time complexity of the algorithms, which is quadratic even in the case of a restricted subclass of *unambiguous grammars* [6], hardly makes them likely for practical applications. Hence, generalization of such parsing algorithms as, for example, Knuth's LR algorithm [14] or Earley's algorithm [9] might be of a practical value.

One more direction of further research is to study constructions of programming languages [27] which can be formally expressed by grammars with contexts.

Thus, for example, already in the grammar constructed in this paper, the nonterminal Q could consider two methods of passing parameters – by reference and by value. In the former case, the argument, corresponding to a parameter passed by reference, should be a variable. Another requirement could be that every variable should be initialized. Such a constraint can be expressed using this time a symmetrical model — grammars with *right contexts*. Thus, when a variable is being declared, one can state that "*there will be later* an assignment to an identifier with the same name". Yet another use of the right contexts could have place when processing forward subroutine declarations (in sense of *Pascal* [12]). When a signature of a function is being declared, it could be checked that "*there will be later* a body of a function with the same name and the same list of formal parameters".

Introducing into a programming language such constructs as *arrays with user-defined constant bounds* (for example, in spirit of *Pascal*, `var a:array[5..10] of integer`) could involve checking that the upper bound is greater than the lower bound. This can be pursued to constructing a grammar for *the language of correct arithmetical expressions with constant terms*, containing such strings as `10+20*2-5==45` and `20+30>=20-30`, but not containg `11*1==1*1*1` or `1/(2-2)>0`. The ideas of this very language could be used in checking such certainly unreachable code as `if (3+7==8+4) x=0`, or checking that the *cases* in a *switch statement* are pairwise disjoint.

Acknowledgments

The author is grateful to Dr. Alexander Okhotin for valuable suggestions and numerous discussions about the paper. The author is thankful to anonymous referees for valuable comments.

References

[1] A. V. Aho. Indexed grammars — an extension of context-free grammars. *J. ACM*, 15(4):647–671, October 1968.

[2] A. V. Aho and J. D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, 1972.

[3] A.Okhotin. Improved normal form for grammars with one-sided contexts. In H. Jürgensen and R. Reis, editors, *DCFS*, volume 8031 of *Lecture Notes in Computer Science*, pages 205–216. Springer, 2013.

[4] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language ALGOL 60. *Commun. ACM*, 6(1):1–17, January 1963.

[5] M. Barash. Recursive descent parsing for grammars with contexts. In *SOFSEM student research forum*, 2013.

[6] M. Barash and A. Okhotin. An extension of context-free grammars with one-sided context specifications. Submitted.

[7] M. Barash and A. Okhotin. Defining contexts in context-free grammars. In A. H. Dediu and C. Martín-Vide, editors, *LATA*, volume 7183 of *Lecture Notes in Computer Science*, pages 106–118. Springer, 2012.

[8] N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959.

[9] J. Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, February 1970.

[10] R. W. Floyd. On the nonexistence of a phrase structure grammar for ALGOL 60. *Commun. ACM*, 5(9):483–484, September 1962.

[11] S. Heilbrunner and L. Schmitz. An efficient recognizer for the Boolean closure of context-free languages. *Theor. Comput. Sci.*, 80(1):53–75, 1991.

[12] K. Jensen and N. Wirth. *Pascal user manual and report – ISO Pascal Standard.* Springer-Verlag, 4 edition, 1991.

[13] A. Jez. Conjunctive grammars generate non-regular unary languages. *Int. J. Found. Comput. Sci.*, 19(3):597–615, 2008.

[14] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[15] D. E. Knuth. Top-down syntax analysis. *Acta Inf.*, 1:79–110, 1971.

[16] D. E. Knuth. The genesis of attribute grammars. In P. Deransart and M. Jourdan, editors, *WAGA*, volume 461 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1990.

[17] A. Okhotin. Conjunctive grammars. *Journal of Automata, Languages and Combinatorics*, 6(4):519–535, 2001.

[18] A. Okhotin. A recognition and parsing algorithm for arbitrary conjunctive grammars. *Theor. Comput. Sci.*, 302(1-3):365–399, 2003.

[19] A. Okhotin. Boolean grammars. *Inf. Comput.*, 194(1):19–48, 2004.

[20] A. Okhotin. On the equivalence of linear conjunctive grammars and trellis automata. *ITA*, 38(1):69–88, 2004.

[21] A. Okhotin. On the existence of a Boolean grammar for a simple programming language. In *Automata and Formal Languages (Proceedings of AFL 2005, 17–20 May 2005, Dobogókő, Hungary)*, 2005.

[22] A. Okhotin. Generalized LR parsing algorithm for Boolean grammars. *Int. J. Found. Comput. Sci.*, 17(3):629–664, 2006.

[23] A. Okhotin. Recursive descent parsing for Boolean grammars. *Acta Inf.*, 44(3-4):167–189, 2007.

[24] A. Okhotin. Fast parsing for Boolean grammars: A generalization of Valiant's algorithm. In Y. Gao, H. Lu, S. Seki, and Sh. Yu, editors, *Developments in Language Theory*, volume 6224 of *Lecture Notes in Computer Science*, pages 340–351. Springer, 2010.

[25] A. Okhotin. A simple P-complete problem and its language-theoretic representations. *Theor. Comput. Sci.*, 412(1-2):68–82, 2011.

[26] W. C. Rounds. LFP: A logic for linguistic descriptions and an analysis of its complexity. *Computational Linguistics*, 14(4):1–9, 1988.

[27] R. W. Sebesta. *Concepts of programming languages.* Pearson Addison-Wesley, Boston, 9 edition, 2010.

[28] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. Revised report on the algorithmic language ALGOL 68. *Acta Inf.*, 5:1–236, 1975.

[29] D. Wotschke. The Boolean closures of the deterministic and nondeterministic context-free languages. In W. Brauer, editor, *GI Jahrestagung*, volume 1 of *Lecture Notes in Computer Science*, pages 113–121. Springer, 1973.