Describing the syntax of programming languages using conjunctive and
Boolean grammars

Alexander Okhotin

Department of Mathematics and Statistics, University of Turku, Turku FI-20014, Finland

Abstract

A classical result by Floyd ("On the non-existence of a phrase structure grammar for ALGOL 60", 1962) states that the complete syntax of any sensible programming language cannot be described by the ordinary kind of formal grammars (Chomsky’s “context-free”). This paper uses slightly more general formal grammar models, namely, conjunctive grammars and Boolean grammars, to describe the set of well-formed programs in a simple typeless procedural programming language. A complete Boolean grammar, which defines such concepts as declaration of variables and functions before use and scopes of visibility, is constructed and explained. Using general parsing algorithms for Boolean grammars, one can then parse a program in time subcubic in its length. Next, it is shown how to transform this grammar to an unambiguous conjunctive grammar, with square-time parsing. This becomes apparently the first specification of the syntax of a programming language entirely by a computationally feasible formal grammar.

Key words: Programming languages, parsing, scannerless parsing, conjunctive grammars, Boolean grammars, context-free grammars

1. Introduction

Formal grammars emerged in the early days of computer science, when the development of the first programming languages required mathematical specification of syntax. The first and the most basic model, independently introduced by Chomsky [8] (as “phrase structure grammars”, later “context-free grammars”), and in the Algol 60 report [20] (as “metalinguistic formulae”, later “Backus–Naur form”), is universally recognized as the standard model of syntax, the ordinary kind of formal grammars.

Already in the Algol 60 report, a grammar was used to describe the complete lexical contents of the language and the basic elements of its syntax. More complicated syntactical requirements, such as the rules requiring declaration of variables before use, were described in plain words together with the semantics of the language. The hope of inventing a more sophisticated grammar that would describe the entire syntax of Algol 60 was buried by Floyd [11], who proved that no (ordinary) grammar could ensure declaration before use. Floyd considered strings of the following form, here translated from Algol 60 to C.

```
main()
{
  int x;  
  x = i;  
  j = k;  
}
```

Such a string is a valid program if and only if \( i = j = k \) (otherwise the assignment statement would refer to an undeclared variable). This very simple fragment of the programming language is an instance of an abstract formal language \( L_1 = \{ a^n b^n c^n \mid n \geq 0 \} \), and since the latter is known to be non-representable by any grammar, one can infer that no grammar can describe the whole programming language.
Many other simple syntactic conditions common to all programming languages are also beyond the scope of ordinary formal grammars. For instance, strings of the following form are valid programs if and only if $i = k$ and $j = \ell$, for otherwise the number of arguments in one of the calls would not match that in the function prototype.

```c
void f(int, ..., int); void g(int, ..., int); main() { f(0, ..., 0); g(0, ..., 0); }
```

This construct is modelled by the formal language $L_2 = \{ a^m b^n a^m c^n | m, n \geq 0 \}$, which is another standard example of a language not described by any grammar. The next example corresponds to checking a pair of identifiers over a two-symbol alphabet; these strings, defined for any two identifiers $w, w' \in \{a, b\}^+$, are valid programs if and only if $w = w'$.

```c
main() { int w; w' = 0; }
```

This is an instance of the formal language $L_3 = \{ wcw | w \in \{a, b\}^+ \}$, investigated, in particular, by Sokolowski [38] and by Wotschke [43]. Not only it is not described by any (ordinary) grammar, it is not representable as an intersection of finitely many languages described by grammars [43].

Floyd’s [11] result, besides pointing out important limitations of grammars, can also be regarded as a formulation of a problem: that of defining an extended formal grammar model powerful enough to describe those constructs of programming languages that are beyond the power of ordinary grammars. Viewed in this way, the result of Floyd has prompted formal language theorists to search for such a model. Many early attempts ended up with seemingly promising new grammar families, which, at a closer examination, turned out to be too powerful to the point of being useless: this can be said of Chomsky’s [8] “context-sensitive grammars” and of van Wijngaarden’s [42] “two-level grammars”, which could simulate Turing machines. Accordingly, these models define computations, rather than any syntactic structures, which makes them irrelevant to language specification.

Besides these unsuccessful attempts, a few generalized grammar models capable of giving meaningful descriptions of syntax were discovered. The first such models were Aho’s indexed grammars [1], Fischer’s macro grammars [10] and tree- adjoining grammars by Joshi et al. [14]. Later, the ideas behind these models led to the more practical multi-component grammars [37] [41], which became a standard model in computational linguistics and receive continued attention.

However, even though these models are powerful enough to define the above three abstract languages $L_1$, $L_2$ and $L_3$, the mere existence of grammars for these languages does not at all imply that a grammar for any programming language can be constructed. On the contrary, a simple extension of Floyd’s example shows that, again, no multi-component grammar can describe the set of well-formed programs. Consider the following strings, in which the same variable is referenced an unbounded number of times.

```c
main() { int x...x; x...x = 0; ... x...x = 0; }
```

Such a string is a well-formed program if and only if all numbers $j_1, \ldots, j_k$ are equal to $i$. This is an instance of an abstract language $L_4 = \{(a^n b)^k | n, k \geq 0\}$, which is known to have no multi-component grammar (because its commutative image is not a semilinear set).

Out of curiosity, one can still construct an indexed grammar for $L_4$. However, it is not difficult to present yet another simple case of programming language syntax that is beyond their expressive power. This time, consider strings of the following form.

```c
int f(int, ..., int); main() { f(f(0, ..., 0), ..., f(0, ..., 0)); }
```

Such strings are valid programs if and only if all numbers $j_1, \ldots, j_k$ are equal to $i$. This is an instance of an abstract language $L_5 = \{(a^n b)^k | n, k \geq 0\}$, which is known to have no multi-component grammar (because its commutative image is not a semilinear set).
Such a string is a well-formed program if and only if \( i = k = j_1 = \ldots = j_k \). This is an instance of an abstract language \( L_5 = \{(a^nb^n)^n \mid n \geq 0\} \), which cannot be described by an indexed grammar.

These examples suggest that no collection of abstract languages can be representative of all syntactic constructs of programming languages. In order to show that some kind of formal grammars are powerful enough to define those syntactic constructs, the only convincing demonstration would be a complete grammar for some programming language. This paper provides such a demonstration for the families of conjunctive grammars and Boolean grammars, constructing a complete grammar for the set of well-formed programs in a simple model procedural language featuring a single data type, a standard set of flow control statements and nested compound statements with rules for variable scope.

Conjunctive grammars \([21, 30]\) extend ordinary grammars by allowing a conjunction of any syntactic conditions to be expressed in any rule. Consider that a rule \( A \rightarrow BC \) in an ordinary grammar states that if a string \( w \) is representable as \( BC \) — that is, \( w = uv \), where \( u \) has the property \( B \) and \( v \) has the property \( C \) — then \( w \) has the property \( A \). In a conjunctive grammar, one can define a rule of the form \( A \rightarrow BC \& DE \), which asserts that every string \( w \) representable both as \( BC \) (with \( w = uv \)) and at the same time as \( DE \) (with \( w = xy \)) therefore has the property \( A \). The more general family of Boolean grammars \([20, 30]\) further allows negation: a rule \( A \rightarrow BC \& \neg DE \), states that if a string is representable as \( BC \) (with \( w = uv \)), but is not representable as \( DE \), then it has the property \( A \).

Although conjunctive grammars have a Chomsky-like definition by term rewriting, whereas Boolean grammars are defined by language equations generalizing those by Ginsburg and Rice \([12]\), their true meaning lies in logic. The understanding of ordinary grammars in terms of logical inference can be found, for instance, in Kowalski’s \([17\) Ch. 3\] textbook. In one of the first papers exploring more powerful logics inspired by grammars, Schuler \([33]\) argues that the set of well-formed programs of Algol 60 can be described in his formalism \([36]\). For the modern logical understanding of grammars, the reader is referred to the fundamental work of Rounds \([33]\), who explained different kinds of formal grammars as fragments of the important FO(LFP) logic \([13, 40]\). Conjunctive grammars are another such fragment.

The importance of conjunctive and Boolean grammars is justified by two facts. On the one hand, they enrich standard inductive definitions of syntax with important logical operations, which extend the expressive power of such definitions in a practically useful way; this shall be further supported in the present paper. On the other hand, these grammars have generally the same parsing algorithms as ordinary grammars \([2, 26, 27, 28, 29]\), and share the same subcubic upper bound on the time complexity of parsing \([31]\), which makes them suitable for implementation.

The most practical parsing algorithm for Boolean grammars is a variant of the Generalized LR (GLR) \([27]\), which runs in worst-case time \( O(n^4) \) and operates very similarly to the GLR for ordinary grammars \([29]\). Two implementations of Boolean GLR are known \([18, 23]\). In the literature, GLR parsers have sometimes been applied to analyzing programming languages symbol by symbol, without an intermediate layer of lexical analysis \([7, 9, 15]\): this is known as scannerless parsing \([34]\). The Boolean grammar for a programming language constructed in this paper follows the same principle, and a Boolean GLR parser for the new grammar is not much different from the GLR operating on an ordinary grammar.

The theoretical work on conjunctive and Boolean grammars is reviewed in a recent survey paper \([30]\). This paper includes all the necessary definitions, given in Section 2 and illustrates the use of conjunction and negation on two examples given in Section 4.

The model programming language is defined in Section 4 and every point of the definition is immediately expressed in the formalism of Boolean grammars. Since the grammar follows the principle of scannerless parsing, for this reason alone, it is bound to be somewhat involved, with definitions of nested syntactic structure occasionally interleaved with simulation of finite automata. This is known in the literature as skipping “water” in search for “islands” \([15]\). In a few places, defining separation into tokens inside the grammar results in rather awkward rules. This is a trait of scannerless parsing and not a fault of Boolean grammars. Although the grammar could have been simplified by combining it with a more or less traditional lexical analyzer, those improvements are not attempted in this paper, which is dedicated to the use of grammars for syntax specification. Finding out the optimal ways of using conjunctive and Boolean grammars is left for future research.

An improvement of a different kind is presented in Section 5, where it is shown how to eliminate negation...
and ambiguity in the earlier constructed grammar. In other words, a Boolean grammar is reformulated
as an unambiguous conjunctive grammar [29], which is a conceptually easier model with a better worst-
case parsing complexity—namely, square time in the length of the input. In the literature on scannerless
parsing, grammars are typically ambiguous, with attached external disambiguation rules [6, 7, 34]. From this
perspective, this section demonstrates a new, entirely grammatical approach to disambiguating scannerless
parsers for programming languages.

The paper is concluded with two kinds of research directions, suggested in Section 6. First, what kind of
parsers could handle this or similar grammars in less than square time? Second, what kind of new grammar
models could describe the syntax of programming languages more conveniently?

2. Conjunctive and Boolean grammars

2.1. Conjunctive grammars

In ordinary formal grammars, rules specify how substrings are concatenated to each other, and one can
define disjunction of syntactic conditions by writing multiple rules for a nonterminal symbol. In conjunctive
grammars, this logic is extended to allow conjunction within the same kind of definitions.

Definition 1 ([21, 30]). A conjunctive grammar is a quadruple \( G = (\Sigma, N, R, S) \), in which:

- \( \Sigma \) is the alphabet of the language being defined;
- \( N \) is a finite set of symbols for the syntactic categories defined in the grammar (in the common jargon, “nonterminal symbols”);
- \( R \) is a finite set of rules, each of the form

\[
A \rightarrow \alpha_1 \& \ldots \& \alpha_m, \tag{1}
\]

where \( A \in N, m \geq 1 \) and \( \alpha_1, \ldots, \alpha_m \in (\Sigma \cup N)^* \);
- \( S \in N \) is a symbol representing the property of being a syntactically well-formed sentence of the
  language (“the initial symbol”).

Each concatenation \( \alpha_i \) in a rule (1) is called a conjunct. If a grammar has a unique conjunct in every rule
\( (m = 1) \), it is an ordinary grammar (Chomsky’s “context-free”). If every conjunct contains at most one non-
terminal symbol \( (\alpha_1, \ldots, \alpha_m \in \Sigma^* \cap \Sigma^* \cup \Sigma^*) \), a grammar is called linear conjunctive. Multiple rules for the
same nonterminal symbol may be presented in the usual notation, such as

\[
A \rightarrow \alpha_1 \& \ldots \& \alpha_m \mid \beta_1 \& \ldots \& \beta_n,
\]
etc. The vertical line is essentially disjunction.

Each rule (1) means that any string representable as each concatenation \( \alpha_i \) therefore has the property \( A \).
This understanding can be equivalently formalized by term rewriting [21] and by language equations [22].
Consider the former definition, which extends Chomsky’s definition of ordinary grammars by string rewriting,
using terms instead of strings.

Definition 2 ([21]). Let \( G = (\Sigma, N, R, S) \) be a conjunctive grammar, and consider terms over concatenation
and conjunction, with symbols from \( \Sigma \cup N \) and the empty string \( \varepsilon \) as atomic terms. The relation of one-step
rewriting on such terms \( (\Rightarrow) \) is defined as follows.

- Using a rule \( A \rightarrow \alpha_1 \& \ldots \& \alpha_m \in R \), with \( A \in N \), any atomic subterm \( A \) of any term may be rewritten
  by the term on the right-hand side of the rule, enclosed in brackets.

\[
\ldots A \ldots \Rightarrow \ldots (\alpha_1 \& \ldots \& \alpha_m) \ldots
\]

- A conjunction of several identical strings may be rewritten to one such string.

\[
\ldots (w \& \ldots \& w) \ldots \Rightarrow \ldots w \ldots \quad (w \in \Sigma^*)
\]
The language defined by a term $\varphi$ is the set of all strings over $\Sigma$ obtained from it in a finite number of rewriting steps.

$$L_G(\varphi) = \{ w \mid w \in \Sigma^*, \varphi \Rightarrow^* w \}$$

The language described by the grammar is the language defined by its initial symbol.

$$L(G) = L_G(S) = \{ w \mid w \in \Sigma^*, S \Rightarrow^* w \}$$

An important property of conjunctive grammars is that every string in $L(G)$ has a corresponding parse tree, which is exactly a proof tree in this logic theory. This is, strictly speaking, an acyclic graph rather than a tree in a mathematical sense. Its leaves (sinks) correspond to the symbols in $w$. Every internal node in this tree is labelled with some rule (1), and has as many descendants as there are symbols in all conjuncts $\alpha_1, \ldots, \alpha_m$. Subtrees corresponding to different conjuncts in a rule define multiple interpretations of the same substring, and accordingly lead to the same set of leaves, as illustrated in Figure 1. For succinctness, the label of an internal node can be just a nonterminal symbol, as long as the rule can be deduced from the descendants’ labels (this is always the case for ordinary grammars, but need not be true for conjunctive grammars).

2.2. Boolean grammars

The second family of grammars used in this paper are Boolean grammars, which further extend conjunctive grammars with a negation operator.

Definition 3 (26, 30). A Boolean grammar is a quadruple $G = (\Sigma, N, R, S)$, where

- $\Sigma$ is the alphabet;
- $N$ is the set of symbols representing syntactic categories;
- $R$ is a finite set of rules of the form
  $$A \rightarrow \alpha_1 \& \ldots \& \alpha_m \& \neg \beta_1 \& \ldots \& \neg \beta_n$$
  with $A \in N$, $m, n \geq 0$, $m + n \geq 1$ and $\alpha_i, \beta_j \in (\Sigma \cup N)^*$;
- $S \in N$ is the initial symbol.
A rule \([2]\) is meant to state that every string representable as all \(\alpha_1, \ldots, \alpha_m\), but not representable as any of \(\beta_1, \ldots, \beta_n\), therefore has the property \(A\). This intuitive definition is formalized by using language equations, that is, by representing a grammar as a system of equations with formal languages as unknowns, and using a solution of this system as the language defined by the grammar. The definition of Boolean grammars exists in two variants: the simple one, given by the author [26], and the improved definition by Koutrouliotis et al. [19] based on the semantics of negation in logic programming. Even though the simple definition handles some extreme cases of grammars improperly [16], it ultimately defines the same family of languages, and is therefore sufficient in this paper.

**Definition 4** (Okhotin [26]). Let \(G = (\Sigma, N, R, S)\) be a Boolean grammar, and consider the following system of equations in which every symbol \(A \in N\) is an unknown language over \(\Sigma\).

\[
A = \bigcup_{A \rightarrow \alpha_1 \& \cdots \& \alpha_m \& \beta_1 \& \cdots \& \beta_n \in R} \left[ \bigcap_{i=1}^{m} \alpha_i \cap \bigcap_{j=1}^{n} \beta_j \right] \quad (3)
\]

Each symbol \(B \in N\) used in the right-hand side of any equation is a reference to a variable, and each symbol \(a \in \Sigma\) represents a constant language \(\{a\}\).

Assume that for every integer \(\ell \geq 0\) there exists a unique vector of languages \((\ldots, L_A, \ldots)_{A \in N}\) with \(L_A \subseteq \Sigma^*\ell\), such that a substitution of \(L_A\) for \(A\), for each \(A \in N\), turns every equation (3) into an equality modulo intersection with \(\Sigma^*\ell\). Then the system is said to have a strongly unique solution, and, for every \(A \in N\), the language \(L_G(A)\) is defined as \(L_A\) from the unique solution of this system. The language generated by the grammar is \(L(G) = L_G(S)\).

Boolean grammars also have parse trees, which, however, reflect only positive components of the parse [26]. When an internal node is labelled with a rule \([2]\), it has descendants corresponding to the symbols in the positive conjuncts \(\alpha_1, \ldots, \alpha_m\), which represent multiple parses of this substring, like in a conjunctive grammar. Negative conjuncts have no representation in the tree.

### 2.3. Ambiguity

Informally, a grammar is unambiguous if every string can be parsed in a unique way. For ordinary grammars, this is formalized by uniqueness of a parse tree. For conjunctive grammars, the same kind of definition would no longer be useful, because a grammar may define multiple parses for some substrings, only to eliminate those substrings later using intersection: then the parse tree can still be unique, but in terms of complexity of parsing, such grammars are ambiguous. For Boolean grammars, a parse tree represents only partial information on the parse, and a definition of ambiguity by parse tree uniqueness becomes completely invalid.

These observations led to the following definition

**Definition 5** ([26]). A Boolean grammar \(G = (\Sigma, N, R, S)\) is unambiguous, if

I. the choice of a rule for every single nonterminal \(A\) is unambiguous, in the sense that for every string \(w\), there exists at most one rule

\[
A \rightarrow \alpha_1 \& \cdots \& \alpha_m \& \beta_1 \& \cdots \& \beta_n,
\]

with \(w \in L_G(\alpha_1) \cap \cdots \cap L_G(\alpha_m) \cap L_G(\beta_1) \cap \cdots \cap L_G(\beta_n)\) (in other words, different rules generate disjoint languages), and

II. all concatenations are unambiguous, that is, for every conjunct \(s_1 \ldots s_\ell\) that occurs in the grammar, and for every string \(w\), there exists at most one partition \(w = u_1 \ldots u_\ell\) with \(u_i \in L_G(s_i)\) for all \(i\).

The unambiguous concatenation requirement applies to positive and negative conjuncts alike. For a positive conjunct belonging to some rule, this means that a string that is potentially generated by this rule must be uniquely split according to this conjunct. For a negative conjunct \(\neg DE\), this condition requests that a partition of \(w \in L_G(DE)\) into \(L_G(D) \cdot L_G(E)\) is unique, even though \(w\) is not defined by any rule involving this conjunct.
3. Language specification with conjunctive and Boolean grammars

Conjunctive grammars for the above languages $L_1 = \{ a^n b^n c^n \mid n \geq 0 \}$, $L_2 = \{ a^m b^n a^m c^n \mid m, n \geq 0 \}$ and $L_3 = \{ w cw \mid w \in \{a, b\}^* \}$ are known to exist \cite{Okhotin21}, and constructing grammars for $L_4$ and $L_5$ is an exercise. What is more important, is that these syntactical elements of programming languages can be described in such a way that generalizes further to a grammar for an entire programming language.

The example of a conjunctive grammar given below defines strings of the form $(a^n b)^k$, with $n \geq 1$ and $k \geq 2$. Such a string models $k - 1$ references to the same declaration $a^n$. In an ordinary grammar, one can define such strings only for $k = 2$, and a typical grammar will use a rule $C \rightarrow aCa$ to match the number of symbols $a$ in the first block (declaration) and in the second block (reference). The following grammar arranges the same matching to be done between the first block and every subsequent block.

**Example 1.** The following conjunctive grammar describes the language $\{ (a^n b)^k \mid n \geq 0, k \geq 1 \}$.

\[
S \rightarrow SA \& Cb \mid A \\
A \rightarrow aA \mid b \\
C \rightarrow aCa \mid B \\
B \rightarrow BA \mid b
\]

The rules for $A$ and for $B$ define regular languages $L(A) = a^n b$ and $L(B) = b(a^n b)^*$. Then, $C$ defines the language $L(C) = \{ a^n x a^n \mid n \geq 0, x \in b(a^n b)^* \}$ representing a single identifier check done in the standard way. The rules for $S$ arrange for all references to be checked by $C$.

All strings $(a^n b)^k$ are defined by $S$ inductively on $k$. If $k = 1$, then a string $a^n b$, representing a lone declaration without references, is given by a rule $S \rightarrow A$. For $k \geq 2$, the rule $S \rightarrow SA \& Cb$ imposes two conditions on a string. First, the conjunct $SA$ requires the string to be a concatenation of a string $(a^n b)^{k-1}$ with any string $a^n b \in L(A)$; this verifies that all earlier references are correct. The other conjunct $Cb$ compares the number of symbols $a$ in the first block $a^n b$ (the declaration) to that in the last block $a^n b$ (the last reference). This ensures that the string is actually $(a^n b)^k$, as desired.

The parse tree of the string $aababaab$ is given in Figure 2. The parts drawn in black show $C$ comparing the number of symbols $a$ in the first block to that in the second block and in the third block. The nodes in the upper part of the tree, labelled with different rules for $S$, arrange these comparisons to be made.

Alternatively, the structure of comparisons defined in this grammar is illustrated in the informal diagram in Figure 3 (left), where the upper part shows how the rule $S \rightarrow SA \& Cb$ recursively refers to $S$ for shorter substrings. The lower part of the diagram illustrates the length equality defined by $C$. All subsequent grammars in this paper shall be illustrated by similar diagrams.

Even though the language in Example 1 is just one simple abstract language, the grammar construction technique for conjunctive grammars demonstrated in this example is sufficient to arrange all identifier checks in a simple programming language. Another essential element is the ability to compare identifiers over a multiple-symbol alphabet, which is modelled in the next example.

**Example 2 (Okhotin \cite{Okhotin21}).** The following conjunctive grammar describes the language $\{ wcw \mid w \in \{a, b\}^* \}$.

\[
S \rightarrow C \& D \\
C \rightarrow aCa \mid aCb \mid bCa \mid bCb \mid c \\
D \rightarrow aA \& aD \mid bB \& bD \mid cE \\
A \rightarrow aAa \mid aAb \mid bAa \mid bAb \mid cEa \\
B \rightarrow aBa \mid aBb \mid bBa \mid bBb \mid cEb \\
E \rightarrow aE \mid bE \mid \varepsilon
\]

First, $C$ defines the language of all strings $xcy$, with $x, y \in \{a, b\}^*$ and $|x| = |y|$, and thus the conjunction with $C$ in the rule for $S$ ensures that the string consists of two parts of equal length separated by a center marker. The other conjunct $D$ checks that the symbols in corresponding positions are the same. The actual language generated by $D$ is $L(D) = \{ uzu \mid u, z \in \{a, b\}^* \}$, and it is defined inductively as follows: a string is in $L(D)$ if and only if
either it is in $c\{a, b\}^*$ (the base case: no symbols to compare),

• or its first symbol is the same as the corresponding symbol on the other side, and the string without its first symbol is in $L(D)$ (that is, the rest of the symbols in the left part correctly correspond to the symbols in the right part).

The comparison of a single symbol to the corresponding symbol on the right is done by the nonterminals $A$ and $B$, which generate the languages \( \{ xcva | x, v, y \in \{a, b\}^*, |x| = |y| \} \) and \( \{ xcvby | x, v, y \in \{a, b\}^*, |x| = |y| \} \), respectively, and the above inductive definition is directly expressed in the rules for $D$, which recursively refer to $D$ in order to apply the same rule to the rest of the string.

The last thing to demonstrate is the use of negation in Boolean grammars. Consider the following variant of Example 1 in which one has to ensure that the first block is not equal to any subsequent block. This is achieved by putting negation over the identifier comparison.

Example 3. The following Boolean grammar describes the language \( \{ a^{n_1}ba^{n_2}b \ldots a^{n_k}b | k \geq 1, n_1, \ldots, n_k \geq 0, n_2 \neq n_1, \ldots, n_k \neq n_1 \} \).

\[
\begin{align*}
S & \rightarrow SA \& -Cb \mid A \\
A & \rightarrow aA \mid b \\
C & \rightarrow aCa \mid B \\
B & \rightarrow BA \mid b
\end{align*}
\]
Figure 3: (left) How the grammar in Example 1 defines strings of the form \((a^n b)^k\); (right) How the grammar in Example 2 defines strings of the form \(uczu\).

This grammar can be rewritten without using negation by replacing \(C\) with a new nonterminal symbol that defines identifier inequality.

4. A model programming language and its grammar

For a quick introduction into the model programming language used in this paper, consider the following sample program in this language.

```
average(x, y) { return (x+y)/2; }
factorial(n) {
    var i, product;
    i=1;
    product=1;
    while(i<=n) {
        product=product*i;
        i=i+1;
    }
    return product;
}
factorial2(n) {
    if(n>=2)
        return n*factorial2(n-1);
    else
        return 1;
}
main(arg) {
    return average(factorial(arg), factorial2(arg));
}
```

This is a well-formed program. All functions and variables are defined before their use. The number of arguments in each function call matches that in its definition: for example, \(\text{average}\) is defined with two arguments and called with two arguments. Each declaration has its scope of visibility: if \(\text{product}\) were declared and initialized inside the \(\text{while}\) statement, then the statement \(\text{return product;}\) would refer to an undeclared variable. There are no duplicate declarations.

The rest of this section gives a semi-formal definition of the syntax of this language, with every point immediately expressed in the formalism of Boolean grammars.

4.1. Alphabet

A program is a finite string over an alphabet \(\Sigma\) that consists of the following 54 characters: 26 letters \((a, \ldots, z)\), 10 digits \((0, \ldots, 9)\), the space \((\ )\), and 17 punctuators \((", ", "\), ", ", ", ", ", +, -, *, /, %, &\, |, !, =, <, >\). All whitespace characters, such as the newline and the tab, are treated as space.
The Boolean grammar to be constructed defines strings over this alphabet Σ. The grammar has the following 7 nonterminals defining basic subsets of the alphabet.

\[
\begin{align*}
\text{anyletter} & \rightarrow a | \ldots | z \\
\text{anydigit} & \rightarrow 0 | \ldots | 9 \\
\text{anyletterdigit} & \rightarrow \text{anyletter} | \text{anydigit} \\
\text{anypunctuator} & \rightarrow \text{"} | \text{\(\{\)} | \text{\(\)}} | \text{\(\},\)} | \text{\(\;\)|\(\),\)} | \text{\(\;\)+|\(\;\)-|\(\;\)*|\(\;\)/} | \text{\(\;\);&|\(\;\)|\(\;\)!|\(\;\)=|\(\;\)<|\(\;\)>|\%} \\
\text{anypunctuatorexceptrightpar} & \rightarrow \text{\(\;\)} | \text{\(\;\),}\)} | \text{\(\;\);}\)} | \text{\(\;\)+|\(\;\)-|\(\;\)*|\(\;\)/} | \text{\(\;\);&|\(\;\)|\(\;\)!|\(\;\)=|\(\;\)<|\(\;\)>|\%} \\
\text{anypunctuatorexceptbraces} & \rightarrow \text{\(\;\)} | \text{\(\;\),}\)} | \text{\(\;\);}\)} | \text{\(\;\)+|\(\;\)-|\(\;\)*|\(\;\)/} | \text{\(\;\);&|\(\;\)|\(\;\)!|\(\;\)=|\(\;\)<|\(\;\)>|\%}
\end{align*}
\]

There are also the following 4 symbols for simple regular sets of strings over Σ.

\[
\begin{align*}
\text{anystring} & \rightarrow \text{anystring} \text{anyletter} | \text{\(\varepsilon\)} \\
\text{anyletterdigits} & \rightarrow \text{anyletter} | \text{anyletterdigit} | \text{\(\varepsilon\)} \\
\text{anystringnotbeginningwithspace} & \rightarrow \text{anyletter} | \text{anyletterdigit} | \text{anystringnotbeginningwithspace} | \text{\(\varepsilon\)} \\
\text{safeendingstring} & \rightarrow \text{anystring} \text{anypunctuator} | \text{anystring} | \text{\(\varepsilon\)}
\end{align*}
\]

The last nonterminal, safeendingstring, denotes a string that can be directly followed by an identifier or a keyword. It shall be used to ensure that names are never erroneously split in two, so that, for instance, an expression \text{varnish;} is never mistaken for a declaration \text{var nish;}.

\subsection*{4.2. Lexical conventions}

The character stream is separated into tokens from left to right, with possible whitespace characters between them. Each time the longest possible sequence of characters that forms a token is consumed, or a whitespace character is discarded. The handling of whitespace is facilitated by a nonterminal \text{ws} representing a possibly empty sequence of whitespace characters.

\[
\begin{align*}
\text{ws} & \rightarrow \text{ws} \varepsilon | \varepsilon
\end{align*}
\]

The programming language has 28 token types. In the grammar, each of them is represented by a nonterminal symbol enclosed in frame (\text{\(\\vphantom{1}\)\(\vphantom{1}\)=\(\vphantom{1}\),\(\vphantom{1}\)}) etc.), which defines the set of all valid tokens of this type, possibly followed by whitespace characters.

First, there are 5 keywords: \text{var}, \text{if}, \text{else}, \text{while}, \text{return}.

\[
\begin{align*}
\text{Keyword} & \rightarrow \text{var} | \text{if} | \text{else} | \text{while} | \text{return} \\
\text{var} & \rightarrow \text{var} \text{ws} \\
\text{if} & \rightarrow \text{if} \text{ws} \\
\text{else} & \rightarrow \text{else} \text{ws} \\
\text{while} & \rightarrow \text{while} \text{ws} \\
\text{return} & \rightarrow \text{return} \text{ws}
\end{align*}
\]

An identifier is a finite nonempty sequence of letters and digits that begins from a letter and is not a keyword. Using the extended descriptive power of Boolean grammars generously, the set of identifiers can be defined precisely according to its definition.

\[
\begin{align*}
\text{id} & \rightarrow \text{tId} \text{ws} & & \text{~Keyword} \\
\text{tId} & \rightarrow \text{anyletter} | \text{tId} \text{anyletter} | \text{tId} \text{anydigit}
\end{align*}
\]
A number is a finite nonempty sequence of digits.

\[
\text{NUM} \rightarrow \text{tNum}_1 \ \text{ws} \\
\text{tNum}_1 \rightarrow \text{tNum}_1 \ \text{anydigit} \mid \text{anydigit}
\]

There are 21 tokens built from punctuator symbols, namely, 13 binary infix operators (+, −, ∗, %, &, |, <, >, <=, ==, !=), 2 unary prefix operators (+, !), the assignment operator (=), the comma (”, “), the semicolon (“;”), parentheses (“(“,”)”), and figure brackets ({"} ).

\[
\begin{align*}
\rightarrow & \rightarrow + \ \text{ws} \\
\rightarrow & \rightarrow / \ \text{ws} \\
\rightarrow & \rightarrow \% \ \text{ws} \\
\rightarrow & \rightarrow \text{mod} \ \text{ws} \\
\rightarrow & \rightarrow ! \ \text{ws} \\
\rightarrow & \rightarrow \text{len} \ \text{ws}
\end{align*}
\]

4.3. Identifier matching

The grammar for the programming language often has to specify that two identifiers are identical. The general possibility of doing that was demonstrated in Example 2 above. A version of that example adapted to handle identifiers and token separation rules in the model programming language shall now be constructed.

The nonterminal symbol \( C \) defines the set of all strings \( wxwy \) where \( w \) is an identifier, \( x \) is an arbitrarily long middle part of the program between these two identifiers, and \( y \) is a possibly empty sequence of whitespace characters.

\[
\begin{align*}
C & \rightarrow C_{\text{len}} \ \& \ C_{\text{iterate}} \mid C \ \sim \\
C_{\text{len}} & \rightarrow \text{anyletterdigit} C_{\text{len}} \ \text{anyletterdigit} \mid \text{anyletterdigit} C_{\text{mid}} \ \text{anyletterdigit}
\end{align*}
\]

The form of the middle part \( x \) in \( wxwy \) is ensured by \( C_{\text{mid}} \), which verifies that \( w \) is the longest prefix and the longest suffix of \( wxw \) that is an identifier.

\[
\begin{align*}
C_{\text{mid}} & \rightarrow \sim \mid \text{anypunctuator anystring anypunctuator} \mid \sim \ \text{anypunctuator} \ \sim \\
C_{\text{mid}} & \rightarrow \sim \ \text{anypunctuator} \ \text{anystring} \ \text{anypunctuator} \mid \sim \ \text{anypunctuator} \ \sim \mid \sim \ \text{anypunctuator}
\end{align*}
\]

In the definition of \( C_{\text{iterate}} \), each symbol \( C_{\sigma} \), with \( \sigma \in \{a,\ldots,z,0,\ldots,9\} \), specializes in comparing one particular symbol of the alphabet (cf. \( A \) and \( B \) in Example 2).

\[
\begin{align*}
C_{\text{iterate}} & \rightarrow C_{\sigma} \ \sigma \ \& \ C_{\text{iterate}} \ \sigma \\
C_{\text{iterate}} & \rightarrow \text{anyletterdigits} C_{\text{mid}} \\
C_{\sigma} & \rightarrow \text{anyletterdigit} C_{\sigma} \ \text{anyletterdigit} \\
C_{\sigma} & \rightarrow \sigma \ \text{anyletterdigits} C_{\text{mid}}
\end{align*}
\]

(for all \( \sigma \in \{a,\ldots,z,0,\ldots,9\} \))

As the first application of identifier comparison, consider generation of lists of unique identifiers. In the sample program given in the beginning of this section, there is a function header \( \text{average}(x, y) \) and a variable declaration statement \( \text{var} \ i, \ \text{prod}; \) Each of them contains a list of identifiers being declared, and no identifier may appear on the list twice.

Such lists are generated by a nonterminal symbol \( Z_{\text{distinct id}} \) in a way that reminds of Example 1 although this time, instead of comparing all elements to the first element, the grammar arranges the comparison of every pair of identifiers on the list. This is done by a two-level iteration: first, \( Z_{\text{distinct id}} \) sets up a comparison of every element to all previous elements, to be carried out by \( \text{no-multiple-declaration} \).

\[
Z_{\text{distinct id}} \rightarrow Z_{\text{distinct id}} \ \text{ID} \ \& \ \text{no-multiple-declarations} \mid \text{ID}
\]

On the second level of iteration, \( \text{no-multiple-declaration} \), applied to a prefix of the list, ensures that no previous element coincides with the last element of the prefix.

\[
\begin{align*}
\text{no-multiple-declaration} & \rightarrow \text{ID} \ \& \ \text{no-multiple-declaration} \ \& \ \neg C \mid \text{ID}
\end{align*}
\]

The actual test for inequality is done by appropriately negating \( C \).
4.4. Expressions

Arithmetical expressions are formed of identifiers and constant numbers using binary operators, unary operators, brackets, function calls and assignment. The definition of an expression \((E)\) is formalized in the grammar in the standard way.

**Basic expressions:** any identifier is an expression and any number is an expression.

\[
E \rightarrow \text{ID} \mid \text{NUM}
\]

**Expression enclosed in brackets:** \((e)\) is an expression for every expression \(e\).

\[
E \rightarrow (E)
\]

**Binary operation:** \(e_1 \ op \ e_2\) is an expression for every binary operator \(op\) and for all expressions \(e_1, e_2\).

\[
E \rightarrow E \ op \ E \quad (op \in \{+,-,\times,/)\}
\]

**Unary operation:** \(op\ e\) is an expression for every unary operator and for all expressions \(e\).

\[
E \rightarrow -E \mid \neg E
\]

**Assignment:** \(x = e\) is an expression for every identifier \(x\) and expression \(e\).

\[
E \rightarrow \text{ID} \leftarrow E
\]

**Function call:** \(f (e_1, \ldots, e_k)\) is an expression for all identifiers \(f\) and expressions \(e_1, \ldots, e_k\), with \(k \geq 0\). This is a call to the function \(f\) with the arguments \(e_1, \ldots, e_k\). For later reference, it is also denoted by a separate nonterminal symbol \(E^{\text{call}}\).

\[
E \rightarrow E^{\text{call}}
\]

\[
E^{\text{call}} \rightarrow \text{ID} (Z_{\text{expr}})
\]

The nonterminal \(Z_{\text{expr}}\) defines (possibly empty) lists of expressions separated by commas.

\[
Z_{\text{expr}} \rightarrow Z_{\text{expr}}^1 \mid \epsilon
\]

\[
Z_{\text{expr}}^1 \rightarrow Z_{\text{expr}}^1 | E \mid E
\]

4.5. Statements

The model programming language has the following six types of statements \((S)\). The rules of the grammar defining their form are standard.

**Expression-statement:** \(e\ ;\) is a statement for every expression \(e\).

\[
S \rightarrow E | E
\]

**Compound statement:** \(\{s_1 \ s_2 \ldots s_k\}\) is a statement for all \(k \geq 0\) and for all statements \(s_1, \ldots, s_k\).

\[
S \rightarrow \{S^*\}
\]

\[
S^* \rightarrow S^* S \mid \epsilon
\]

**Conditional statement:** \(\text{if } (e) \ s\) and \(\text{if } (e) \ s\ \text{else} \ s'\) are statements for every expression \(e\) and statements \(s, s'\).

\[
S \rightarrow \text{if } (E) S
\]

\[
S \rightarrow \text{if } (E) S \text{ else } S
\]
**Iteration statement:** while (e) s is a statement for every expression e and statement s.

\[ S \rightarrow \text{while} \ (E) \ S \]

**Declaration statement:** var \(x_1, \ldots, x_k\); is a statement for every \(k \geq 1\) and for all identifiers \(x_1, \ldots, x_k\). Declaration statements are also denoted by a separate nonterminal symbol \(S^{\text{var}}\).

\[ S \rightarrow S^{\text{var}} \]
\[ S^{\text{var}} \rightarrow \text{var} \ Z_{\text{distinct id}} \]

**Return statement:** return e ; is a statement for every expression e.

\[ S \rightarrow S^{\text{return}} \]
\[ S^{\text{return}} \rightarrow \text{return} \ E \ | \ 	ext{returnstatementfix} \]
\[ \text{returnstatementfix} \rightarrow \text{return} \ anypunctuator \ anystring | \text{return} \ anystring \]

A conjunction with returnstatementfix is an awkward solution to the problem that an expression statement returnable; should never be mistaken for a return statement return able; This could be done without using conjunction, but with more complications.

The grammar also requires a subclass of returning statements (\(S_r\)), which may terminate their execution only by a return statement. A return statement itself is returning.

\[ S_r \rightarrow S^{\text{return}} \]

A conditional statement is returning, if both branches are returning.

\[ S_r \rightarrow \text{if} \ (E) \ S_r \ 	ext{else} \ S_r \]

A compound statement is returning, if so is its last constituent. The set of returning compound statements is also denoted by a separate nonterminal symbol, \(S^{\text{compound}}_r\).

\[ S_r \rightarrow S^{\text{compound}}_r \]
\[ S^{\text{compound}}_r \rightarrow \{ \ S^* \ S_r \ \} \]

### 4.6. Function declarations

A function declaration begins with a \textit{header} of the form \(f (x_1, \ldots, x_k)\), where \(k \geq 0\) and \(f, x_1, \ldots, x_k\) are identifiers. The identifier \(f\) is the \textit{name} of the function, and the identifiers \(x_1, \ldots, x_k\) are its \textit{formal arguments}.

\[ F_{\text{header}} \rightarrow \text{ID} \ ([ \ Z_{\text{distinct id}} \ ] \) \]
\[ F_{\text{header}} \rightarrow \text{ID} \ ([ \ ] \) \]

A \textit{function declaration} (\(F\)) is a header \(F_{\text{header}}\) followed by a returning compound statement \(S^{\text{compound}}_r\), called the \textit{body} of the function.

\[ F \rightarrow F_{\text{header}} S^{\text{compound}}_r \ 	ext{& ID} \ ([ \ all-variables-declared \ ] \]

The second conjunct in the latter rule refers to the nonterminal symbol \textit{all-variables-declared} representing the conditions on variable declaration. Intuitively, one can interpret this rule so as \(F_{\text{header}} S^{\text{compound}}_r\) first defines a certain structure, and then \textit{all-variables-declared} processes this structure to verify declaration of variables before use. Then, for every string being defined by \textit{all-variables-declared}, one can assume that it is already of the form \(F_{\text{header}} S^{\text{compound}}_r\). This makes the rules for \textit{all-variables-declared}, presented below, easier to construct and understand.
4.7. Declaration of variables before use

For each function, the goal is to check that every reference to a variable in the function body is preceded
by a declaration of a variable with the same name. A reference is an identifier occurring in an expression.
Declarations take place in the list of function arguments and in var statements; in the latter case, the
reference should be in the scope of this declaration, that is, within the same compound statement as the var
statement or in any nested statements, and occurring later than the var statement. Another related thing
to check is that no declaration is in the scope of another declaration of the same variable. The purpose of
the nonterminal symbol all-variables-declared is to check all these conditions for a particular function.

The rules for the nonterminal all-variables-declared iterate over all prefixes (of the function body) that
end with an identifier, as illustrated in Figure 4. This is done in generally the same way as in Example 1,
with the following details to note. First, the function body is split into tokens again, and all irrelevant
tokens are skipped. Special care has to be exercised when skipping a number or a keyword, because the
characters forming them might actually be a suffix of an identifier to be checked; this possibility is ruled out
in the rule for all-variables-declared-safe.

\[
\begin{align*}
\text{all-variables-declared} & \rightarrow \text{all-variables-declared-safe} \text{NUM} \\
\text{all-variables-declared} & \rightarrow \text{all-variables-declared-safe} \text{Keyword} \\
\text{all-variables-declared} & \rightarrow \text{all-variables-declared-safe} \text{ID } ( \\
\text{all-variables-declared-safe} & \rightarrow \text{all-variables-declared & safeendingstring}
\end{align*}
\]

Any punctuator character is skipped, unless it is a semicolon concluding a var statement.

\[
\begin{align*}
\text{all-variables-declared} & \rightarrow \text{all-variables-declared} \text{ anypunctuator WS & ~safeendingstring } S^{\text{var}} \\
\text{It is important to distinguish between identifiers representing declarations and identifiers representing}
\end{align*}
\]

references. Once a var statement is found, these-variables-not-declared shall verify that none of the variables
defined here have previously been defined.

\[
\begin{align*}
\text{all-variables-declared} & \rightarrow \text{these-variables-not-declared } \text{; & all-variables-declared-safe } S^{\text{var}} \\
\text{For each reference found, the rules invoke another nonterminal this-variable-declared to check that this}
\end{align*}
\]

variable has an earlier declaration.

\[
\begin{align*}
\text{all-variables-declared} & \rightarrow \text{this-variable-declared & all-variables-declared-safe } ID
\end{align*}
\]
Finally, once the whole function body is processed, only the list of arguments in its header remains. This list may be empty, hence two terminating rules.

\[
\begin{align*}
\text{all-variables-declared} & \rightarrow Z \text{distinct id} \ |
\end{align*}
\]

Thus, for every prefix ending with a reference to a variable, the nonterminal \textit{this-variable-declared} is used to match it to a declaration of a variable that occurs inside this prefix, as shown in Figure 5. This time, the rules iterate over all suffixes of the current prefix, beginning at different tokens and ending with the identifier being checked. First, the rules for \textit{this-variable-declared} search for a declaration among the function’s arguments, and if it is found, it is left to match the identifiers using \textit{C}.

\[
\begin{align*}
\text{this-variable-declared} & \rightarrow \text{id} , \text{this-variable-declared} \ | \ C \\
\text{this-variable-declared} & \rightarrow \text{id} , \text{this-variable-declared} \ |
\end{align*}
\]

If failed, the nonterminal \textit{declared-inside-function} is invoked to look for a declaration in a suitable \textit{var} statement.

\[
\begin{align*}
\text{this-variable-declared} & \rightarrow \text{id} \ |
\end{align*}
\]

If the function has no arguments, the search for a \textit{var} statement in the body begins by the following rule.

\[
\begin{align*}
\text{this-variable-declared} & \rightarrow \ |
\end{align*}
\]

While searching for a \textit{var} statement, variable scopes have to be observed, and for that purpose, the rules for \textit{declared-inside-function} parse the function body according to the nested structure of statements. First, any complete statement may be ignored: this means that the desired variable is not declared there.

\[
\begin{align*}
\text{declared-inside-function} & \rightarrow S \text{ declared-inside-function} \\
\text{declared-inside-function} & \rightarrow \ |
\end{align*}
\]

If the reference being checked is inside a compound statement, then the following rule moves the search one level deeper into a nested compound statement.

\[
\begin{align*}
\text{declared-inside-function} & \rightarrow \ |
\end{align*}
\]

For \textit{if} and \textit{while} statements, a nested scope is entered through an extra nonterminal \textit{declared-inside-function-nested}, which indicates that the current statement is not directly within a compound statement.

\[
\begin{align*}
\text{declared-inside-function} & \rightarrow \text{if} \ (E) \ \text{declared-inside-function-nested} \\
\text{declared-inside-function} & \rightarrow \text{if} \ (E) \ S \ \text{else} \ \text{declared-inside-function-nested} \\
\text{declared-inside-function} & \rightarrow \text{while} \ (E) \ \text{declared-inside-function-nested}
\end{align*}
\]
The rules for `declared-inside-function-nested` process potentially nested `if` and `while` statements and get back to `declared-inside-function` as soon as a compound statement begins.

\[
\begin{align*}
\text{declared-inside-function-nested} & \rightarrow \{ \text{declared-inside-function} \\
\text{declared-inside-function-nested} & \rightarrow \text{if} \ (E) \text{ declared-inside-function-nested} \\
\text{declared-inside-function-nested} & \rightarrow \text{if} \ (E) \ S \ \text{else} \text{ declared-inside-function-nested} \\
\text{declared-inside-function-nested} & \rightarrow \text{while} \ (E) \text{ declared-inside-function-nested}
\end{align*}
\]

If a `var` statement is encountered, the desired declaration may be there. In this case, the nonterminal `declared-inside-function` is used to find the correct declaration among the variables listed in this `var` statement, and `C` is invoked to match identifiers.

\[
\begin{align*}
\text{declared-inside-function} & \rightarrow \text{var} \ \text{declared-in-this-statement} \\
\text{declared-in-this-statement} & \rightarrow \text{id} \ , \ \text{declared-in-this-statement} \\
\text{declared-in-this-statement} & \rightarrow C \ \& \ \text{ignore-remaining-variables} \ \text{skip-part-of-this-scope}
\end{align*}
\]

After skipping the remaining variables declared in this `var` statement (`ignore-remaining-variables`), the middle part between the declaration and the reference is described by a nonterminal `skip-part-of-this-scope`, which ensures that the reference stays in the scope of the declaration.

\[
\begin{align*}
\text{ignore-remaining-variables} & \rightarrow \text{id} \ , \ \text{ignore-remaining-variables} \ | \ \text{id} \ ; \\
\text{skip-part-of-this-scope} & \rightarrow \text{skip-part-of-this-scope} \{ \ S^* \} \\
\text{skip-part-of-this-scope} & \rightarrow \text{skip-part-of-this-scope} \{ \} \\
\text{skip-part-of-this-scope} & \rightarrow \text{skip-part-of-this-scope} \text{anycharexceptbracespace} \text{ws}
\end{align*}
\]

Now consider the other nonterminal `these-variables-not-declared`, which is used in the rules for `all-variables-declared` for any prefix ending with a declaration, in order to ensure that none of these variables have been declared before. Here negation comes in particularly useful, because the condition that an identifier is in the scope of a variable with the same name has already been expressed as `this-variable-declared`, and now it is sufficient to negate it. The rules for `these-variables-not-declared` iterate over all variables declared at this point.

\[
\begin{align*}
\text{these-variables-not-declared} & \rightarrow \text{these-variables-not-declared} \ | \ \text{id} \ & \neg \text{this-variable-declared}
\end{align*}
\]

The iteration terminates after checking the first variable declared in this `var` statement.

\[
\begin{align*}
\text{these-variables-not-declared} & \rightarrow \text{safeendingstring} \ \text{var} \ \text{ID} \ \& \ \neg \text{this-variable-declared}
\end{align*}
\]

4.8. Declaration of functions before use

Another kind of references to be matched to their declarations are calls to functions. Whenever a function is called, somewhere earlier in the program there should be a function header that opens a declaration of a function with the same name and with the same number of arguments. Furthermore, a program may not contain multiple declarations of functions sharing the same name and the same number of arguments.

Checking these conditions requires matching each function call to a suitable earlier function declaration. Similarly to the rules for `all-variables-declared`, this is done by considering all prefixes of the program that end with a function call or a function header. For that purpose, all tokens except right parentheses are being
Whenever a right parenthesis is found, there are three possibilities, each handled in a separate rule for `function-declarations`.

First, this right parenthesis could be the last symbol of a function call expression, for which one should find a matching function declaration. This case is identified by the following two conditions: the current substring should end with a function call expression (`E_{call}`), and at the same time the entire substring should not be of the form `F^* F_{header}`. The latter condition is essential, because otherwise (in this model programming language) a function call is indistinguishable from a list of arguments in a function header.

This case is illustrated in Figure 6. The concatenation in the last conjunct of the rule splits the current prefix of the program into zero or more irrelevant function declarations (`F^*`) followed by a substring that begins with a header of the desired function and ends with the function call expression, with these two sharing the same name and having same number of arguments (`this-function-declared-here`). The comparison of identifiers (`same-function-name`) and of the number of arguments (`same-number-of-arguments`) is carried out.

Figure 6: How `function-declarations` handles a call to a function `name()`.
Figure 7: How function-declarations detects duplicate declarations.

out in the following rules.

\[
\text{this-function-declared-here} \rightarrow \text{same-function-name} \& \text{same-number-of-arguments}
\]

\[
\text{same-function-name} \rightarrow C \left( \begin{array}{c} Z_{expr} \end{array} \right)
\]

\[
\text{same-number-of-arguments} \rightarrow \text{id} \left( \text{n-of-arg-equal-0} \right)
\]

\[
\text{n-of-arg-equal-0} \rightarrow \text{anystringnotbeginningwithspace} \left( \text{n-of-arg-equal} \right) \text{E}
\]

\[
\text{n-of-arg-equal} \rightarrow \text{id} \left( \text{n-of-arg-equal} \right) \text{E}
\]

The second possibility with a right parenthesis encountered in function-declarations is when it is the last symbol of a function header. In this case, the grammar should ensure that that no other functions with the same name are declared. The second conjunct of the following rule verifies that the current substring ends with a function header rather than with a function call, whereas the third conjunct negates the condition of having an earlier declaration of the same function.

\[
\text{function-declarations} \rightarrow \text{function-declarations} \left( \right) \& F^* \text{Fheader} \& \neg F^* \text{this-function-declared-here}
\]

Figure 7 demonstrates how this rule detects multiple declarations of the same function, so that the substring is not defined by function-declarations. For it to be defined, it should have no partition into $F^*$this-function-declared-here.

The last case in function-declarations is when a substring ends with a right parenthesis, but it neither marks an end of a function call expression, nor is a part of a function header. This can happen in several ways: it could be a subexpression enclosed in parentheses, or a part of a for or a while statement. In each case, there is nothing to check, and the right parenthesis is skipped like any other token. The case is identified by not ending with $E^{call}$.

\[
\text{function-declarations} \rightarrow \text{function-declarations} \left( \right) \& \neg \text{safeendingstring} E^{call}
\]

4.9. Programs

It remains to give the rules describing the set of well-formed programs in the model programming language. A program is a finite sequence of function declarations, which contains a function with the name main, with one argument.

A sequence of function declarations and a declaration of the main function are defined by the following rules.

\[
F^* \rightarrow F^* F \mid \varepsilon
\]

\[
F_{main} \rightarrow \text{main WS} \left( \begin{array}{c} \text{id} \end{array} \right) S_{\text{compound}} \& \text{id} \left( \text{all-variables-declared} \right)
\]
Finally, a single rule for the initial symbol $\text{Program}$ defines what a well-formed program is. This rule also defines possible whitespace characters occurring before the first token.

$$\text{Program} \to \text{ws} \, F^* \, F_{\text{main}} \, F^* \, \& \, \text{ws} \, \text{function-declarations}$$

This completes the grammar.

**Proposition 1.** The set of well-formed programs in the model programming language is described by a Boolean grammar with 118 nonterminal symbols and 365 rules.

The grammar constructed above can be used with any of the several known parsing algorithms for Boolean grammars. First, there is a simple extension of the Cocke–Kasami–Younger algorithm, with the running time $O(n^3)$ in the length of the input [20]. Like in the case of ordinary grammars, this algorithm can be accelerated to run in time $O(n^\omega)$ [31], where $\omega < 3$ is the exponent in the complexity of matrix multiplication. The most practical algorithm is the GLR [27], which has worst-case running time $O(n^4)$, but may run faster for some grammars and inputs, if a particular parse goes on partially deterministically.

The grammar has been tested on a large set of positive and negative examples using one of the existing implementations of GLR parsing for Boolean grammars [23]. The parser contains 760 states and operates in generally the same way as GLR parsers for ordinary grammars.

5. Eliminating negation and ambiguity

The grammar for the model programming language given in Section 4 uses negation several times and contains quite a lot of syntactical ambiguity. Disregarding these shortcomings made grammar construction easier.

In general, unintended ambiguity is always undesirable, and the use of negation can also be viewed as an unnecessary complication. The purpose of this section is to explain how the grammar can be transformed to an unambiguous conjunctive grammar describing exactly the same language. Besides being a conceptually clearer model, unambiguous conjunctive grammars also allow faster parsing.

5.1. Negation in auxiliary definitions

At two occasions, the grammar uses the negation in the definitions of basic constructs. The same definitions now have to be reformulated without the negation.

First, in Section 4.2 the rule defining identifiers [10] uses negation to describe a regular language $\{a, \ldots, z\} \{a, \ldots, z, 0, \ldots, 9\}^* \setminus \{\text{var, if, else, while, return}\}$. The same language can be recognized by a 21-state finite automaton, which can in turn be simulated in the grammar.

A more interesting use of negation is in the rule for $\text{no-multiple-declaration}$, where it is applied to $C$ in order to state identifier inequality. To eliminate the negation here, one should define a new nonterminal symbol $\tilde{C}$ that would describe all strings $uxvy$, where $u$ and $v$ are distinct identifiers, $x$ is the middle part of the program between these two identifiers, and $y$ is a possibly empty sequence of whitespace characters. The first possibility for $u$ and $v$ not to be equal is if they are of different length; this case is handled in the rules for $\tilde{C}_{\text{len}<}$ ($|u| < |v|$) and for $\tilde{C}_{\text{len}>}$ ($|u| > |v|$).

$$\tilde{C} \to \tilde{C}_{\text{len}<} \mid \tilde{C}_{\text{len}>} \mid \tilde{C}_{\text{iterate}}$$

$$\tilde{C}_{\text{len}>} \to \text{anyletterdigit} \, \tilde{C}_{\text{len}>} \mid \text{anyletterdigit} \, C_{\text{len}}$$

$$\tilde{C}_{\text{len}<} \to \tilde{C}_{\text{len}<} \, \text{anyletterdigit} \mid C_{\text{len}} \, \text{anyletterdigit}$$

Otherwise, if $|u| = |v|$, then $\tilde{C}_{\text{iterate}}$ begins comparing the symbols of $u$ and $v$ in the same way as done in $C_{\text{iterate}}$, using $C_\sigma$ to check each symbol.

$$\tilde{C} \to C_{\text{len}} \& \tilde{C}_{\text{iterate}}$$

$$\tilde{C}_{\text{iterate}} \to C_\sigma \sigma \& \tilde{C}_{\text{iterate}} \sigma \quad \text{(for all } \sigma \in \{a, \ldots, z, 0, \ldots, 9\}\text{)}$$

19
The iteration in $\tilde{C}_{iterate}$ is stopped when a pair of distinct symbols is encountered, that is, if $u = u'\sigma w$ and $v = v'\tau w$, for some $u'$ and $v'$.

$$\tilde{C}_{iterate} \rightarrow C_\sigma \tau$$

(for all $\sigma, \tau \in \{a, \ldots, z, 0, \ldots, 9\}$, with $\sigma \neq \tau$)

5.2. Two standard ambiguous constructs

The rules in Section 4.4 include two standard cases of ambiguous definitions in programming languages. The first of them concerns the expressions. The given definition is ambiguous, because the precedence and the associativity of operators are not defined. One could make the rules for $E$ unambiguous by rewriting them in the standard way, introducing a new nonterminal symbol for each level of precedence.

The rules defining the conditional operator feature another classical kind of ambiguity, known as the “dangling else” ambiguity. Indeed, a string such as if($x$) if($y$) $s$; else $t$; can be parsed in two different ways, depending on whether the last else clause binds to the first or to the second if statement. This ambiguity can also be resolved in the standard way, by introducing a variant of $S$ called $S_{not\ if-then}$, which should define all statements except those of the form if-then, without an else clause. Then, all rules for $S$ describing statements other than conditional statements are preserved, whereas the rules describing conditional statements take the following form.

$$S \rightarrow \text{if } ( E ) S$$

$$S \rightarrow \text{if } ( E ) S_{not\ if-then} \text{ else } S$$

The new nonterminal symbol $S_{not\ if-then}$ does not have a rule of the former type (if-then), but otherwise, it has all the same rules as $S$.

5.3. Variable declarations

The rules requiring declaration of variables before reference, as given in Section 4.7, essentially use negation to ensure that a variable has not been declared before ($\neg this\-variable\-declared$). Furthermore, there is some subtle ambiguity in the rules for $this\-variable\-declared$, declared-inside-function and declared-in-this-statement. Both shortcomings shall now be corrected by reimplementing parts of the grammar.

First, consider the ambiguity, which manifests itself on any program containing a variable with multiple declarations. Even though any such program is ultimately considered ill-formed, according to Definition 5, this is still ambiguity, and it affects parsing complexity. Consider the following ill-formed function declaration.

```plaintext
f( arg, arg)
{ var arg; var arg, arg; arg
\text{this-variable-declared} =0;
}
```

When checking the reference to arg, the nonterminal this-variable-declared has to handle the underlined substring. First, there is an ambiguity between the rules $this\-variable\-declared \rightarrow C$ and $this\-variable\-declared \rightarrow ID$ $\neg this\-variable\-declared$. The former matches the first argument of the function to the reference, whereas the latter ignores the first argument and looks for a declaration of arg later. For this particular string, both conditions hold at the same time, hence the ambiguity. Next, there is a similar ambiguity between matching the last argument of the function ($this\-variable\-declared \rightarrow C$) and looking for a declaration of arg inside the function body ($this\-variable\-declared \rightarrow ID$ $\neg declared\-inside\-function$). Later on in the grammar, there is an ambiguity between the first and the second var statements: in other words, ambiguity in the choice between declared-inside-function $\rightarrow S$ declared-inside-function and declared-inside-function $\rightarrow var$ declared-in-this-statement. Finally, when the second var statement is analyzed by declared-in-this-statement, one can use the first or the second identifier in it: this is the ambiguity between the two rules for declared-in-this-statement.

In each case, ambiguity could be resolved by using negation to set the precedence of the two conditions explicitly. In general, given two rules $A \rightarrow \alpha$ and $A \rightarrow \beta$, and assuming that the former has higher
precedence, the latter rule would be replaced with $A \rightarrow \beta \& \neg \alpha$ [29 Prop. 2]. The goal is to do the same without using negation.

The solution proposed here is to introduce a negative counterpart for each of the three nonterminals that need to be negated (this-variable-declared, declared-inside-function, declared-in-this-statement). The rules for this negative counterpart implement a dual version of the rules for its original positive version, and define the opposite condition. A general transformation that achieves this effect is known only for the special class of linear conjunctive grammars [24 Thm. 5]. Although the grammar considered here is not exactly linear, the idea of that general method shall be used to dualize the particular rules of this grammar.

The new nonterminal symbols representing negations of the required conditions shall be called this-variable-not-declared, not-declared-inside-function and not-declared-in-this-statement. First, consider how these nonterminals shall be used in the existing grammar. The rule for these-variables-not-declared with an explicit negation is rewritten using the negative nonterminal.

$$these-variables-not-declared \rightarrow these-variables-not-declared \& id, this-variable-not-declared$$

The ambiguity between the first three rules for this-variable-declared is resolved by allowing earlier declarations ($C$) only if there are no later declarations ($id$, this-variable-declared or $id$) {declared-inside-function). Thus, the rule this-variable-declared $\rightarrow C$ is replaced with the following two rules.

$$this-variable-declared \rightarrow C \& id, this-variable-not-declared$$
$$this-variable-declared \rightarrow C \& id \{ not-declared-inside-function$$

This prioritizes later declarations over earlier declarations. The same principle is followed in disambiguating the definitions of declared-inside-function and declared-in-this-statement. For the former, every var statement is processed for declarations only if this variable is not declared later.

$$declared-inside-function \rightarrow var \& declared-in-this-statement \& S \& not-declared-inside-function$$

For declared-in-this-statement, a declaration is matched to a reference ($C$) only if no later declaration in this var statement is suitable.

$$declared-in-this-statement \rightarrow C \& ignore-remaining-variables skip-part-of-this-scope \& \& id \{ not-declared-in-this-statement$$

$$declared-in-this-statement \rightarrow C \& id \{ \{ skip-part-of-this-scope$$

It remains to define the rules for the negative versions of the three nonterminals in question. The rules for this-variable-not-declared ensure that the identifier being checked is different from each of the function’s arguments.

$$this-variable-not-declared \rightarrow \tilde{C} \& id \& this-variable-not-declared$$

Then the search proceeds into the body of the function.

$$this-variable-not-declared \rightarrow \tilde{C} \& id \{ not-declared-inside-function$$
$$this-variable-not-declared \rightarrow \{ not-declared-inside-function$$

Turning to not-declared-inside-function, let $S_{\text{not var}}$ be a new nonterminal that denotes all well-formed statements except var statements. These are the statements that not-declared-inside-function is allowed to skip without consideration: whatever declarations are made inside such a statement, the reference being checked is not in their scope.

$$not-declared-inside-function \rightarrow S_{\text{not var}} \& not-declared-inside-function$$
The rules for $S_{\text{not var}}$ can be defined by reusing the rules for $S$. Next, $\text{not-declared-inside-function}$ navigates through the nested structure of statements in the same way as $\text{declared-inside-function}$. 

\[
\text{not-declared-inside-function} \rightarrow \{ \text{not-declared-inside-function} \\
\text{not-declared-inside-function} \rightarrow \text{if} \ (E) \ \} \ \text{not-declared-inside-function-nested} \\
\text{not-declared-inside-function} \rightarrow \text{if} \ (E) \ S \ \text{else} \ \} \ \text{not-declared-inside-function-nested} \\
\text{not-declared-inside-function} \rightarrow \text{while} \ (E) \ \} \ \text{not-declared-inside-function-nested}
\]

When a $\text{var}$ statement is encountered, $\text{not-declared-in-this-statement}$ is invoked to verify that this statement does not declare the variable under consideration. At the same time, $\text{not-declared-inside-function}$ recursively refers to itself to make sure that this variable is also not declared in any subsequent statements.

\[
\text{not-declared-inside-function} \rightarrow \text{var} \ \text{not-declared-in-this-statement} \ \& \ S_{\text{var}} \ \text{not-declared-inside-function}
\]

Unlike the nonterminal $\text{declared-inside-function}$, which ends the iteration by finding a suitable declaration, here the iteration ends in a short substring without variable declarations.

\[
\text{not-declared-inside-function} \rightarrow \text{anystringwithoutbracesandsemicolons} \\
\text{not-declared-inside-function-nested} \rightarrow \text{anystringwithoutbracesandsemicolons}
\]

Finally, $\text{not-declared-in-this-statement}$ applies $\tilde{C}$ to each identifier in the current $\text{var}$ statement, in order to ensure that all of them are different from the identifier in the end of the substring.

\[
\text{not-declared-in-this-statement} \rightarrow \tilde{C} \ \& \ \text{id} ; \ \text{anystringwithoutbraces skip-part-of-this-scope}
\]

5.4. Function declarations

The rules describing declaration of functions, given in Section 4.8, suffer from the same kind of problems as the rules for variable declarations. This part of the grammar shall be reconstructed similarly to what was done in the above Section 5.3.

First, consider the conjunct $F^* \ \text{this-function-declared-here}$ in one of the rules for $\text{function-declarations}$, which concatenates a prefix with zero or more irrelevant function declarations to a substring that begins with a declaration of the desired function, and ends with a call to that function. This concatenation is ambiguous, because the function being called may have multiple declarations, as demonstrated in the following example.

\[
\text{function()} \ {\{} \ \text{return} \ 0 ; \ {\}} \ \text{function()} \ {\{} \ \text{return} \ 1 ; \ {\}} \ \text{main}(arg) \ {\{} \ \text{return} \ \text{function}(); \ {\}}
\]

Here the underlined substring has two partitions as $F^* \ \text{this-function-declared-here}$, corresponding to the first and the second declaration of $\text{function()}$.

In order to look up functions unambiguously, instead of using concatenation to get to the desired declaration at once, one should process all declarations iteratively, one by one, in the same way as for variable declarations.
declarations. This shall be done in a new nonterminal \textit{this-function-declared}, which reimplements the concatenation $F^\ast \textit{ this-function-declared-here}$. The rules for \textit{this-function-declared} shall iteratively consider all substrings that begin with various function declarations and end with the reference being checked, and apply \textit{this-function-declared-here} to every such substring. Furthermore, doing this unambiguously by the same method as in Section 5.3 requires negative counterparts of these two nonterminals, called \textit{this-function-not-declared} and \textit{this-function-not-declared-here}.

According to this plan, the three rules for \textit{function-declarations} dealing with the right parenthesis are rewritten as follows. First, if this is a function call, then the new nonterminal \textit{this-function-declared} verifies that there is a declaration of the function being called. In order to make sure that the rule indeed deals with a function call rather than with a declaration, an extra conjunct states that this prefix of the program is a sequence of function declarations followed by a header and an incomplete compound statement, using the nonterminal \textit{skip-part-of-this-scope} defined in Section 4.7.

\begin{verbatim}
function-declarations \to function-declarations \& safeendingstring \textit{E} \textit{call} \&
\& F^\ast \textit{F}_{\text{header}} \{ \textit{skip-part-of-this-scope} \& this-function-declared
\end{verbatim}

Second, if this is a function header, then the negative version of the new nonterminal (\textit{this-function-not-declared}) shall ensure that there are no earlier declarations of any functions with the same name and the same number of arguments.

\begin{verbatim}
function-declarations \to function-declarations \& safeendingstring \textit{E} \textit{call} \&
\& F^\ast \textit{F}_{\text{header}} \& this-function-not-declared
\end{verbatim}

The third case is when the string ending with a right parenthesis is not of the form “safeendingstring \textit{E} \textit{call}”. Since negation is no longer allowed, one has to list all possibilities of how a prefix of a well-formed program could be of such a form. First of all, in could end with an expression enclosed in brackets; in this case, that expression must be preceded by some punctuator character (which is either an operator or a bracket within a larger expression, or the last character of some syntactical unit other than an expression).

\begin{verbatim}
function-declarations \to function-declarations \& anystring anypunctuator \textit{ws} \{ \textit{E} \}
function-declarations \to function-declarations \& safeendingstring \textit{if} \{ \textit{E} \}
function-declarations \to function-declarations \& safeendingstring \textit{while} \{ \textit{E} \}
\end{verbatim}

The next goal is to define the rules for the new nonterminals \textit{this-function-declared} and \textit{this-function-not-declared}. The first rule for \textit{this-function-declared} skips any function declarations, as long as there is still a declaration of this function later on.

\begin{verbatim}
this-function-declared \to F \textit{this-function-declared}
\end{verbatim}

The second rule describes a substring that begins with a desired function declaration and ends with a reference to the same function: this condition is checked by the nonterminal \textit{this-function-declared-here}, using the rules defined in Section 4.8.

\begin{verbatim}
this-function-declared \to this-function-declared-here \& F \textit{this-function-not-declared}
\end{verbatim}

In order to keep the grammar unambiguous, the second conjunct of this rule ensures that no later function declaration matches this reference. The last rule handles a special case, where the substring contains a function header and a part of the same function’s body ending with its recursive call to itself.

\begin{verbatim}
this-function-declared \to this-function-declared-here \& \textit{F}_{\text{header}} \{ \textit{skip-part-of-this-scope}
\end{verbatim}
Here the nonterminal `skip-part-of-this-scope`, reused from Section 4.7, ensures that the substring contains no other function declarations.

Now consider checking a function for having no declaration (`this-function-not-declared`). This is done for strings of the same form as for `this-function-not-declared`: that is, for a substring beginning with zero or more function declarations, and continued either with a function header or with an incomplete function declaration ending with a function call. One has to verify that the function in the end of the substring has no earlier declarations. For substrings that begin with a function declaration, the first rule states that the function is declared neither here, nor later.

\[
\text{this-function-not-declared} \rightarrow \text{this-function-not-declared-here} \& F \text{ this-function-not-declared}
\]

Once all earlier function declarations are processed in this way, eventually there is one of two base cases to handle. First, `this-function-not-declared` may have to deal with a substring that consists of just one function header. It has already been checked that this function has no earlier declarations, and the iteration ends here.

\[
\text{this-function-not-declared} \rightarrow F_{\text{header}}
\]

The second case is when a substring begins with a function header and continues with an incomplete function body ending with a function call expression. Then, the following rule ensures that this is not a valid recursive call to itself.

\[
\text{this-function-not-declared} \rightarrow \text{this-function-not-declared-here} \& F_{\text{header}} \{ \text{skip-part-of-this-scope} \}
\]

Finally, it remains to write down the rules for `this-function-not-declared-here`. Dually to the rule for `this-function-declared-here` from Section 4.8, here one can say that either the two functions have different names, or they have the same name but a different number of arguments.

\[
\begin{align*}
\text{this-function-not-declared-here} & \rightarrow \text{different-function-name} \\
\text{this-function-not-declared-here} & \rightarrow \text{same-function-name} \& \text{different-number-of-arguments}
\end{align*}
\]

Name mismatch is tested using \(\tilde{C}\).

\[
\text{different-function-name} \rightarrow \tilde{C} \{ Z_{\text{expr}} \}
\]

Mismatch in the number of arguments is described without using conjunction.

\[
\begin{align*}
\text{different-number-of-arguments} & \rightarrow \text{id (n-of-arg-less)} \\
n-of-arg-less & \rightarrow n-of-arg-less E \mid n-of-arg-match E \mid n-of-arg-match E \\
n-of-arg-greater & \rightarrow \text{id (n-of-arg-greater)} \mid \text{id (n-of-arg-match)} \mid \text{id (n-of-arg-match)}
\end{align*}
\]

5.5. The main function

The rules for the main function defined in Section 4.9 are ambiguous, because the program may contain multiple declarations for the main function. To be precise, the concatenation `\(F^* F_{\text{main}} F^*\)` in the rule for `Program` is ambiguous, because there are as many partitions as there are main functions declared in a program (which is obviously ill-formed).

The rule for `Program` is therefore rewritten by using a new nonterminal symbol `\(F^{\prime\prime*}\)` that represents any sequence of function declarations that contains a main function.

\[
\text{Program} \rightarrow \text{ws } F^{\prime\prime*} \& \text{ws } \text{function-declarations}
\]

24
The rules for $F^{\ast n_\ast}$ add function declarations, one by one, as long as they are not for a main function. Once the last declaration of a main function in the program is located, it does not matter which functions are defined before it.

$$F^{\ast n_\ast} \rightarrow F^{\ast n_\ast} F_{\text{main}} \mid F^{\ast} F_{\text{main}}$$

(note that if the first rule were replaced with $F^{\ast n_\ast} \rightarrow F^{\ast n_\ast} F$, then the grammar would become ambiguous again) Finally, it remains to define a declaration of a function that is not a main function, and to do this without using the negation. The first possibility is that the function’s name is not main. Assume that the set of all identifiers other than main is defined in a new nonterminal symbol $id_{\neg \text{main}}$ with its rules simulating a finite automaton.

$$F_{\neg \text{main}} \rightarrow id_{\neg \text{main}} \text{ ws } \{ \textit{Z}\text{distinct id} \} \text{ scompound } \& \text{ id } \{ \text{ all-variables-declared} \}$$

The other case is when the function is called main, but has none or at least two arguments.

$$F_{\neg \text{main}} \rightarrow \text{main} \text{ ws } \{ \textit{2}^{2_+}\text{distinct id} \} \text{ scompound } \& \text{ id } \{ \text{ all-variables-declared} \}$$

This completes the last correction to the grammar.

**Proposition 2.** The set of well-formed programs in the model programming language is described by an unambiguous conjunctive grammar with 189 nonterminal symbols and 3832 rules.

There are so many rules because of two finite automaton simulations for $id$ and for $id_{\neg \text{main}}$, and also because of many rules of the form $\tilde{C}_{\text{iterate}} \rightarrow C_{\sigma \tau}$ needed to compare identifiers for inequality.

The parsing complexity of unambiguous conjunctive grammars, as well as of unambiguous Boolean grammars, is much lower than in their general, ambiguous case. Using a variant of the Kasami–Torii algorithm [29], one can parse the model programming language in time $O(n^2)$, where $n$ is the length of the program. It is also likely that the Generalized LR also works in square time on any unambiguous Boolean grammar, although that has not been formally proved.

### 5.6. A possible linear conjunctive grammar

It is possible that the grammar presented in this section could be further reconstructed into a linear conjunctive grammar. This is a simpler model notable for its equivalence to a family of cellular automata [25]. The best known complexity upper bound for linear conjunctive grammars is still $O(n^2)$, so this transformation is of a pure theoretical interest.

Some parts of the grammar, such as identifier comparison in Section 4.3, are already almost linear conjunctive, and require only very obvious transformations. Another kind of inessential non-linearity is caused by the nonterminal symbols representing tokens, which are being concatenated throughout the grammar. Since each of them defines a regular language, their meaning can be directly substituted into the rules.

The rules defining the nested structure of expressions, statements and functions, given in Sections 4.4–4.6, are essentially non-linear. However, that structure is simple enough to be recognized by input-driven pushdown automata [19], also known under the name of visibly pushdown automata [3]—and those automata can in turn be simulated by linear conjunctive grammars [32].

The rules concerned with declaration before use, in their final form given in Sections 5.3–5.4, have many essentially non-linear concatenations. As a representative example, consider the rule

$$\text{not-declared-inside-function} \rightarrow \text{if } \{ E \} \{ \text{ not-declared-inside-function-nested} \}$$

Strictly speaking,
it concatenates 6 nonterminal symbols. However, four of them represent tokens and accordingly define only regular sets, whereas $E$ is a bracketed construction recognized by an input-driven pushdown automaton. It is conjectured that this and all other such concatenations in the grammar can be simulated by linear conjunctive rules by further elaborating the known simulation of input-driven pushdown automata \[32\].

If the suggested transformation works out, the resulting linear conjunctive grammar will be large and incomprehensible. It should then be regarded as a kind of “machine code” implementing the grammar given in Sections 4.1–5.5.

6. Afterthoughts

The main purpose of this paper was to demonstrate that the expressive power of conjunctive and Boolean grammars is sufficient to describe some of the harder syntactic elements in programming languages. This becomes the first successful experience of constructing a complete formal grammar from a practically parsable class for a programming language.

As a first experience, it has many shortcomings. The model programming language is quite restricted, and enriching its syntax even a little bit, while staying within Boolean grammars, would be challenging, if at all possible. Some parts of the resulting description are quite natural, some are admittedly awkward. Square-time parsing may be fast enough for some applications, but it is still too slow in comparison with standard parsers for programming languages. These obvious observations suggest the following research directions.

First, could there exist a substantially faster parsing algorithm that would still be applicable to (some variant of) the grammar given in this paper? The existing linear-time algorithms for subclasses of conjunctive grammars \[2, 28\] are too restrictive. If the model programming language defined here were parsed by a standard human-written program, that parser would maintain a symbol table of some sort, filling in new entries upon reading declarations, and looking it up for every reference. This suggests that if a prospective parsing algorithm is to parse this language substantially faster than in square time, then it will most likely need some advanced data structures. Would it be possible to adapt the GLR algorithm to index its graph-structured stack using a symbol table? The grammar in Section 5 demonstrates the kind of rules such an algorithm is expected to handle.

The second research direction is motivated by the awkward parts of the grammar and by the limitations of the model programming language. These can be regarded as signs of imperfection of conjunctive and Boolean grammars, and from this perspective, the goal is to find a new grammar formalism, in which all that could be done better. Any such formalisms must maintain efficient parsing algorithms, and would likely be found by further extending conjunctive or Boolean grammars. Over the years this paper has been under preparation, Barash \[4\] found out that the grammar in this paper can be improved and the model language extended, if the grammar model is augmented with operators for referring to the contexts of a substring. The resulting new grammar model has a purely logical definition and still has a cubic-time parsing algorithm \[5\].

It would be interesting to see any other new attempts to define a suitable grammar model.

The last suggested topic concerns conjunctive and Boolean grammars as they are, and their applications. The grammar constructed in this paper is an extreme case of using these models, meant to demonstrate the possibility of describing a number of constructs. It remains to find more optimal ways of using these grammar construction methods. They should not be difficult to implement in existing projects, since the widely used GLR parsing algorithm can handle conjunctive and Boolean rules \[18, 27\].

References


Both grammars presented in this paper actually work. This appendix gives the exact grammars, as they were used with a GLR parser generator [23].

A. Appendix: the Boolean grammar

```
algorithm=GLR;

terminal _space;
terminal _a, _b, _c, _d, _e, _f, _g, _h, _i, _j, _k, _l, _m, _n, _o, _p, _q,
_r, _s, _t, _u, _v, _w, _x, _y, _z;
terminal _leftpar, _rightpar, _leftbrace, _rightbrace, _comma, _semicolon;
terminal _plus, _minus, _star, _slash, _percent, _and, _or, _excl, _eq, _lt, _gt;

Program -> WS Functions FunctionMain Functions & WS function_declarations;
Functions -> Functions F | e;
F -> FunctionHeader CompoundStmts & tId tLeftPar all_variables_declared;
FunctionMain -> _m _a _i _n WS tLeftPar tId tRightPar CompoundStmts & tId tLeftPar all_variables_declared;
FunctionHeader -> tId tLeftPar ListOfDistinctIds tRightPar | tId tLeftPar tRightPar;
WS -> WS _space | e;
anyletter -> _a | _b | _c | _d | _e | _f | _g | _h | _i | _j | _k | _l | _m | _n | _o | _p | _q |
_r | _s | _t | _u | _v | _w | _x | _y | _z;
anymultiplicator -> _leftpar | _rightpar | _leftbrace | _rightbrace | _comma | _semicolon | _plus | _minus | _star | _slash | _and | _or | _excl | _eq | _lt | _gt | _percent;
anymultiplicator_except_rightpar -> _leftpar | _leftbrace | _comma | _semicolon | _plus | _minus | _star | _slash | _and | _or | _excl | _eq | _lt | _gt | _percent;
anycarexceptbracespace -> _leftpar | _rightpar | _comma | _semicolon | _plus | _minus | _star | _slash | _and | _or | _excl | _eq | _lt | _gt | _percent | anyletter | anydigit;
anymultiplicator_except_bracespace -> _leftpar | _rightpar | _comma | _semicolon | _plus | _minus | _star | _slash | _and | _or | _excl | _eq | _lt | _gt | _percent;
anystringnotbeginningwithspace -> anylettern anystring | anydigit anystring | anypunctuator anystring | e;
safeendingstring -> anystring anypunctuator | anystring _space | e;
tVar -> _v _a _r WS;
tIf -> _i _f WS;
tElse -> _e _l _s _e WS;
tWhile -> _w _h _i _l _e WS;
tReturn -> _r _e _t _u _r _n WS;
Keyword -> tVar | tIf | tElse | tWhile | tReturn;
tId -> tId aux WS & "Keyword;";
tId aux -> anyletter | tId aux anyletter | tId aux anydigit;
tNum -> tNum aux WS;
tNum aux -> tNum aux anydigit | anydigit;

ListOfDistinctIds -> ListOfDistinctIds tComma tId & no_multiple_declarations | tId;
no_multiple_declarations -> tId tComma no_multiple_declarations & "C" | tId;
```
tPlus -> _plus WS;
tMinus -> _minus WS;
tStar -> _star WS;
tSlash -> _slash WS;
tMod -> _percent WS;
tAnd -> _and WS;
tOr -> _or WS;
tLessThan -> _lt WS;
tGreaterThan -> _gt WS;
tLessEqual -> _lt _eq WS;
tGreaterEqual -> _gt _eq WS;
tEqual -> _eq _eq WS;
tNotEqual -> _excl _eq WS;
tUnaryMinus -> _minus WS;
tNot -> _excl WS;
tLeftPar -> _leftpar WS;
tRightPar -> _rightpar WS;
tLeftBrace -> _leftbrace WS;
tRightBrace -> _rightbrace WS;
tAssign -> _eq WS;
tSemicolon -> _semicolon WS;
tComma -> _comma WS;

E -> tId | tNum;
E -> tLeftPar E tRightPar;
E -> E tPlus E | E tMinus E | E tStar E | E tSlash E | E tMod E | E tAnd E | E tOr E | E tLessThan E | E tGreaterThan E | E tLessEqual E | E tGreaterEqual E | E tEqual E | E tNotEqual E;
E -> tUnaryMinus E | tNot E;
E -> tId tAssign E;
E -> EFunctionCall;
EFunctionCall -> tId tLeftPar ListOfExpr tRightPar;
ListOfExpr -> ListOfExpr1 | e;
ListOfExpr1 -> ListOfExpr1 tComma E | E;
S -> E tSemicolon |
    tLeftBrace S_star tRightBrace |
    tIf tLeftPar E tRightPar S |
    tWhile tLeftPar E tRightPar S |
    VarSt | ReturnSt;
S_star -> S_star S | e;
VarSt -> tVar _space ListOfDistinctIds tSemicolon;
ReturnSt ->
tReturn E tSemicolon & returnstatementfix;
returnstatementfix ->
    _r _e _t _u _r _n anypunctuator anystring |
    _r _e _t _u _r _n _space anystring;
SRet -> CompoundStRet |
    tIf tLeftPar E tRightPar SRet tElse SRet | ReturnSt;
CompoundStRet ->
tLeftBrace S_star SRet tRightBrace;
C -> Clen & Citerate | C _space;
Clen -> anyletterdigit Clen anyletterdigit | anyletterdigit Cmid anyletterdigit;
C_a -> anyletterdigit C_a anyletterdigit | _a anyletterdigit Cmid;
C_b -> anyletterdigit C_b anyletterdigit | _b anyletterdigit Cmid;
C_c -> anyletterdigit C_c anyletterdigit | _c anyletterdigit Cmid;
C_d -> anyletterdigit C_d anyletterdigit | _d anyletterdigit Cmid;
C_e -> anyletterdigit C_e anyletterdigit | _e anyletterdigit Cmid;
C_f -> anyletterdigit C_f anyletterdigit | _f anyletterdigit Cmid;

30
C_g -> anyletterdigit C_g anyletterdigit | _g anyletterdigits Cmid;
C_h -> anyletterdigit C_h anyletterdigit | _h anyletterdigits Cmid;
C_i -> anyletterdigit C_i anyletterdigit | _i anyletterdigits Cmid;
C_j -> anyletterdigit C_j anyletterdigit | _j anyletterdigits Cmid;
C_k -> anyletterdigit C_k anyletterdigit | _k anyletterdigits Cmid;
C_l -> anyletterdigit C_l anyletterdigit | _l anyletterdigits Cmid;
C_m -> anyletterdigit C_m anyletterdigit | _m anyletterdigits Cmid;
C_n -> anyletterdigit C_n anyletterdigit | _n anyletterdigits Cmid;
C_o -> anyletterdigit C_o anyletterdigit | _o anyletterdigits Cmid;
C_p -> anyletterdigit C_p anyletterdigit | _p anyletterdigits Cmid;
C_q -> anyletterdigit C_q anyletterdigit | _q anyletterdigits Cmid;
C_r -> anyletterdigit C_r anyletterdigit | _r anyletterdigits Cmid;
C_s -> anyletterdigit C_s anyletterdigit | _s anyletterdigits Cmid;
C_t -> anyletterdigit C_t anyletterdigit | _t anyletterdigits Cmid;
C_u -> anyletterdigit C_u anyletterdigit | _u anyletterdigits Cmid;
C_v -> anyletterdigit C_v anyletterdigit | _v anyletterdigits Cmid;
C_w -> anyletterdigit C_w anyletterdigit | _w anyletterdigits Cmid;
C_x -> anyletterdigit C_x anyletterdigit | _x anyletterdigits Cmid;
C_y -> anyletterdigit C_y anyletterdigit | _y anyletterdigits Cmid;
C_z -> anyletterdigit C_z anyletterdigit | _z anyletterdigits Cmid;
C_0 -> anyletterdigit C_0 anyletterdigit | _0 anyletterdigits Cmid;
C_1 -> anyletterdigit C_1 anyletterdigit | _1 anyletterdigits Cmid;
C_2 -> anyletterdigit C_2 anyletterdigit | _2 anyletterdigits Cmid;
C_3 -> anyletterdigit C_3 anyletterdigit | _3 anyletterdigits Cmid;
C_4 -> anyletterdigit C_4 anyletterdigit | _4 anyletterdigits Cmid;
C_5 -> anyletterdigit C_5 anyletterdigit | _5 anyletterdigits Cmid;
C_6 -> anyletterdigit C_6 anyletterdigit | _6 anyletterdigits Cmid;
C_7 -> anyletterdigit C_7 anyletterdigit | _7 anyletterdigits Cmid;
C_8 -> anyletterdigit C_8 anyletterdigit | _8 anyletterdigits Cmid;
C_9 -> anyletterdigit C_9 anyletterdigit | _9 anyletterdigits Cmid;
Citerate ->
  C_a & Citerate _a | C_b & Citerate _b | C_c & Citerate _c | C_d & Citerate _d |
  C_e & Citerate _e | C_f & Citerate _f | C_g & Citerate _g | C_h & Citerate _h |
  C_i & Citerate _i | C_j & Citerate _j | C_k & Citerate _k | C_l & Citerate _l |
  C_m & Citerate _m | C_n & Citerate _n | C_0 & Citerate _0 | C_1 & Citerate _1 |
  C_2 & Citerate _2 | C_3 & Citerate _3 | C_4 & Citerate _4 | C_5 & Citerate _5 |
  C_6 & Citerate _6 | C_7 & Citerate _7 | C_8 & Citerate _8 | C_9 & Citerate _9;
Cmid -> anypunctuator anystring anypunctuator |
  _space anystring anypunctuator |
  anypunctuator anystring _space |
  _space anystring _space | anypunctuator |
  _space;

/* Function declarations */

function_declarations ->
  function_declarations anypunctuatorexceptrightpar WS |
  function_declarations_safe Keyword |
  function_declarations_safe tId |
  function_declarations_safe tNum |
  function_declarations tRightPar & ~safeendingstring EFunctionCall |
  function_declarations tRightPar & Functions FunctionHeader & ~Functions this_function_declared_here |
  function_declarations tRightPar & safeendingstring EFunctionCall & Functions this_function_declared_here & ~Functions FunctionHeader |
  e;
function_declarations_safe -> function_declarations & safeendingstring;

this_function_declared_here -> same_function_name & same_number_of_arguments;

31
same_function_name -> C tLeftPar ListOfExpr tRightPar;
same_number_of_arguments ->
tId tLeftPar n_of_arg_equal_0 tRightPar |
tId tLeftPar n_of_arg_equal tRightPar;
n_of_arg_equal_0 -> tRightPar anystringnotbeginningwithspace tLeftPar;
n_of_arg_equal ->
tId tLeftPar n_of_arg_equal tComma E |
tId tRightPar anystringnotbeginningwithspace tLeftPar E;

/* Variable declarations */
all_variables_declared ->
  all_variablesDeclared anypunctuator WS & ~safeendingstring VarSt |
  all_variables_declared_safe tId tLeftPar |
  all_variables_declared_safe tNum |
  all_variables_declared_safe Keyword |
  these_variables_not_declared tSemicolon & all_variables_declared_safe VarSt |
  this_variable_declared & all_variables_declared_safe tId |
 ListOfDistinctIds tRightPar |
  tRightPar;
all_variables_declared_safe ->
  all_variables_declared & safeendingstring;
this_variable_declared -> C |
  tId tComma this_variable_declared |
  tId tRightPar tLeftBrace declared_inside_function |
  tRightPar tLeftBrace declared_inside_function;
declared_inside_function ->
  S declared_inside_function |
  tLeftBrace declared_inside_function |
  tIf tLeftPar E tRightPar declared_inside_function_1 |
  tIf tLeftPar E tRightPar S tElse declared_inside_function_1 |
  tWhile tLeftPar E tRightPar declared_inside_function_1 |
  tVar _space declared_in_this_statement;
declared_inside_function_1 ->
  tLeftBrace declared_inside_function |
  tIf tLeftPar E tRightPar declared_inside_function_1 |
  tIf tLeftPar E tRightPar S tElse declared_inside_function_1 |
  tWhile tLeftPar E tRightPar declared_inside_function_1;
declared_in_this_statement ->
  tId tComma declared_in_this_statement |
  C & ignore_remaining_variables skip_part_of_this_scope;
ignore_remaining_variables ->
  tId tComma ignore_remaining_variables |
  tId tSemicolon;
skip_part_of_this_scope ->
  skip_part_of_this_scope tLeftBrace S_star tRightBrace |
  skip_part_of_this_scope tLeftBrace |
  skip_part_of_this_scope anycharexceptbracespace WS |
  e;
these_variables_not_declared ->
  these_variables_not_declared tComma tId & ~this_variable_declared |
  safeendingstring tVar _space tId & ~this_variable_declared;

B. Appendix: the unambiguous conjunctive grammar

algorithm=GLR;

terminal _space;
terminal _a, _b, _c, _d, _e, _f, _g, _h, _i, _j, _k, _l, _m, _n, _o, _p, _q, _r, _s, _t, _u, _v, _w, _x, _y, _z;
terminal _0, _1, _2, _3, _4, _5, _6, _7, _8, _9;
terminal _leftpar, _rightpar, _leftbrace, _rightbrace, _comma, _semicolon;
terminal _plus, _minus, _star, _slash, _percent, _and, _or, _excl, _eq, _lt, _gt;

Program -> WS Functions_with_main & WS function_declarations;
Functions_with_main -> Functions_with_main FunctionNotMain | Functions FunctionMain;
Functions -> Functions F | e;
F -> FunctionHeader CompoundStRet & tId tLeftPar all_variables_declared;
FunctionHeader -> tId tLeftPar ListOfDistinctIds _leftpar | tId tLeftPar tRightPar;
FunctionMain -> _m _a _i _n WS tLeftPar tId tRightPar CompoundStRet & tId tLeftPar all_variables_declared;
FunctionNotMain ->
  tIdnotmain tLeftPar ListOfDistinctIds tRightPar CompoundStRet
  & tId tLeftPar all_variables_declared |
  tIdnotmain tLeftPar tRightPar CompoundStRet & tId tLeftPar all_variables_declared |
  _m _a _i _n WS tLeftPar ListOfDistinctIds_not_1 tRightPar CompoundStRet
  & tId tLeftPar all_variables_declared |
  _m _a _i _n WS tLeftPar tRightPar CompoundStRet & tId tLeftPar all_variables_declared;
WS -> WS _space | e;
anyletter -> _a | _b | _c | _d | _e | _f | _g | _h | _i | _j | _k | _l | _m | _n | _o | _p | _q | _r | _s | _t | _u | _v | _w | _x | _y | _z;
anydigit -> _0 | _1 | _2 | _3 | _4 | _5 | _6 | _7 | _8 | _9;
anyletterdigit -> anyletter | anydigit;
anypunctuator -> _leftpar | _rightpar | _leftbrace | _rightbrace | _comma | _semicolon | _plus | _minus | _star | _slash | _and | _or | _excl | _eq | _lt | _gt | _percent;
anypunctuatorexceptcomma -> _leftpar | _rightpar | _leftbrace | _rightbrace | _comma | _semicolon | _plus | _minus | _star | _slash | _and | _or | _excl | _eq | _lt | _gt | _percent;
anypunctuatorexceptleftparsemicolon -> _rightpar | _leftbrace | _rightbrace | _comma | _semicolon | _plus | _minus | _star | _slash | _and | _or | _excl | _eq | _lt | _gt | _percent;
anypunctuatorexceptrightpar -> _leftpar | _leftbrace | _rightbrace | _comma | _semicolon | _plus | _minus | _star | _slash | _and | _or | _excl | _eq | _lt | _gt | _percent;
anypunctuatorexceptleftparencolon -> _rightpar | _leftbrace | _rightbrace | _comma | _plus | _minus | _star | _slash | _and | _or | _excl | _eq | _lt | _gt | _percent;
anychar -> _space | anyletter | anydigit | anypunctuator;
anypunctuator -> anypunctuator anystring | e;
anypunctuatorexceptbracespace -> anypunctuator | anystring | anypunctuator anystring |
  e;
anypunctuatorexceptbraces -> anypunctuator anychar | e;
anypunctuatorexceptbracesandsemicolon -> anypunctuator anychar | e;
anypunctuatorexceptbracesandsemicolons -> anypunctuator anychar | e;
anypunctuatorexceptbracesandsemicolons_spacespace -> anypunctuator anychar | e;
anypunctuatorexceptbracesandsemicolons_spacespace anypunctuator anystring |
  e;
tWhile -> _v_ _h_ i _l_ e _W_ S;
tReturn -> _r_ e _t_ u _r_ n _W_ S;
Keyword -> tVar | tIf | tElse | tWhile | tReturn;
Id -> id all a | id all b | id all c | id all d | id e e | id f f | id all g | id all h | id all i | id all j | id all k | id all l | id all m | id all n | id all o | id all p | id all q | r _s _t |
_id -> id all a | id all b | id all c | id all d | id all e | id all f | id all g | id all h | id all i | id all j | id all k | id all l | id all m | id all n | id all o | id all p | id all q |
_id_e -> id all a | id all b | id all c | id all d | id all e | id all f | id all g | id all h | id all i | id all j | id all k | id all l | id all m | id all n | id all o | id all p | id all q |
_id_eturn -> id all a | id all b | id all c | id all d | id all e | id all f | id all g | id all h | id all i | id all j | id all k | id all l | id all m | id all n | id all o | id all p | id all q |
_id_ile -> id all a | id all b | id all c | id all d | id all e | id all f | id all g | id all h | id all i | id all j | id all k | id all l | id all m | id all n | id all o | id all p | id all q |
_id_lse -> id all a | id all b | id all c | id all d | id all e | id all f | id all g | id all h | id all i | id all j | id all k | id all l | id all m | id all n | id all o | id all p | id all q |
_id_n -> id all a | id all b | id all c | id all d | id all e | id all f | id all g | id all h | id all i | id all j | id all k | id all l | id all m | id all n | id all o | id all p | id all q |
_id_ -> id all a | id all b | id all c | id all d | id all e | id all f | id all g | id all h | id all i | id all j | id all k | id all l | id all m | id all n | id all o | id all p | id all q |
_id_thing principally the same as id; it is used to denote a general class of identifiers, which can be further qualified by an additional specifier.

34
idall_q | idall_r | idall_s | idall_t | idall_u | idall_v | idall_w | idall_x | idall_y |
idall_z | idall_0 | idall_1 | idall_2 | idall_3 | idall_4 | idall_5 | idall_6 | idall_7 |
idall_8 | idall_9 | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
Dr | q | r | x | s | t | u | v | w | x | y | z;
idnotmain_urn -> idall_a | idall_b | idall_c | idall_d | idall_e | idall_f | idall_g | idall_h | idall_i | idall_j | idall_k | idall_l | idall_m | idall_n | idall_o | idall_p | idall_q |
idall_r | idall_s | idnotmain_turn | t | idall_u | idall_v | idall_w | idall_x | idall_y | idall_z | idall_0 | idall_1 | idall_2 | idall_3 | idall_4 | idall_5 | idall_6 | idall_7 |
idall_8 | idall_9 | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
p | q | r | x | s | t | u | v | w | x | y | z;
tNum -> tNum_aux WS;
tNum_aux -> tNum_aux anydigit | anydigit;
ListOfDistinctIds -> ListOfDistinctIds tComma tId & no_multiple_declarations | tId;
no_multiple_declarations -> tId tComma no_multiple_declarations & Cneg | tId;
ListOfDistinctIds_not_1 -> ListOfDistinctIds_not_1 tComma tId & no_multiple_declarations |
tId tComma tId & no_multiple_declarations;
tPlus -> .plus WS;
tMinus -> .minus WS;
tStar -> .star WS;
tSlash -> .slash WS;
tMod -> .percent WS;
tAnd -> .and WS;
tOr -> .or WS;
tLessThan -> .lt WS;
tGreaterThan -> .gt WS;
tLessEqual -> .lt _eq WS;
tGreaterEqual -> .gt _eq WS;
tEqual -> .eq _eq WS;
tNotEqual -> .excl _eq WS;
tUnaryMinus -> .minus WS;
tNot -> .excl WS;
tLeftPar -> .leftpar WS;
tRightPar -> .rightpar WS;
tLeftBrace -> .leftbrace WS;
tRightBrace -> .rightbrace WS;
tAssign -> .eq WS;
tSemicolon -> .semicolon WS;
tComma -> .comma WS;
E -> tId tAssign E | E1;
E1 -> E1 tOr E2 | E2;
E2 -> E2 tAnd E3 | E3;
E3 -> E3 tLessThan E4 | E3 tGreaterThan E4 | E3 tLessEqual E4 | E3 tEqual E4 | E3 tNotEqual E4 | E4;
E4 -> E4 tPlus E5 | E4 tMinus E5 | E5;
E5 -> E5 tStar E6 | E5 tSlash E6 | E5 tMod E6 | E6;
E6 -> tUnaryMinus E6 | tNot E6 | tId | tNum | tLeftPar E tRightPar | EFunctionCall;
EFunctionCall ->
tId tLeftPar ListOfExpr tRightPar;
ListOfExpr -> ListOfExpr1 | e;
ListOfExpr1 -> ListOfExpr1 tComma E | E;
S -> ExprSt | CompoundSt | VarSt | IterationSt | ReturnSt |
tIf tLeftPar E tRightPar S |
tIf tLeftPar E tRightPar SNotifthen tElse S;
SNotifthen -> ExprSt | CompoundSt | VarSt | IterationSt | ReturnSt |
tIf tLeftPar E tRightPar SNotifthen tElse S;
SNotvar -> ExprSt | CompoundSt | IterationSt | ReturnSt |
tIf tLeftPar E tRightPar S |
tIf tLeftPar E tRightPar SNotifthen tElse S;
ExprSt -> E tSemicolon;
CompoundSt -> tLeftBrace S_star tRightBrace;
S_star -> S_star S | e;
VarSt -> tVar _space ListOfDistinctIds tSemicolon;
IterationSt -> tWhile tLeftPar E tRightPar S;
ReturnSt -> tReturn E tSemicolon & returnstatementfix;
returnstatementfix ->
    _r _e _t _u _r _n anypunctuator anystring |
    _r _e _t _u _r _n _space anystring;
SRet -> CompoundStRet |
tIf tLeftPar E tRightPar SRet tElse SRet |
ReturnSt;
CompoundStRet -> tLeftBrace S_star SRet tRightBrace;
C -> Clen & Citerate | C _space;
Cneg -> Clenlt | Clengt | Clen & Citerateneg | Cneg _space;
Clenlt -> Clenlt C_iterate | C_a anyletterdigit | C_b anyletterdigit | C_c anyletterdigit | C_d anyletterdigit | C_e anyletterdigit | C_f anyletterdigit | C_g anyletterdigit | C_h anyletterdigit | C_i anyletterdigit | C_j anyletterdigit | C_k anyletterdigit | C_l anyletterdigit | C_m anyletterdigit | C_n anyletterdigit | C_o anyletterdigit | C_p anyletterdigit | C_q anyletterdigit | C_r anyletterdigit | C_s anyletterdigit | C_t anyletterdigit | C_u anyletterdigit | C_v anyletterdigit | C_w anyletterdigit | C_x anyletterdigit | C_y anyletterdigit | C_z anyletterdigit | C_0 anyletterdigit | C_1 anyletterdigit | C_2 anyletterdigit | C_3 anyletterdigit | C_4 anyletterdigit | C_5 anyletterdigit | C_6 anyletterdigit | C_7 anyletterdigit | C_8 anyletterdigit | C_9 anyletterdigit;
C_iterate ->
    C_a _a & C_iterate _a | C_b _b & C_iterate _b | C_c _c & C_iterate _c | C_d _d & C_iterate _d |
    C_e _e & C_iterate _e | C_f _f & C_iterate _f | C_g _g & C_iterate _g | C_h _h & C_iterate _h |
    C_i _i & C_iterate _i | C_j _j & C_iterate _j | C_k _k & C_iterate _k | C_l _l & C_iterate _l |
/** Function declarations */

function_declarations ->
    function_declarations anypunctuatorexceptrightpar WS |
    function_declarations_safe Keyword |
    function_declarations_safe tId |
    function_declarations_safe tNum |
    function_declarations tRightPar & anystring anypunctuatorexceptrightpar |
    function_declarations_safe tIf tLeftPar E tRightPar |
    function_declarations_safe tWhile tLeftPar E tRightPar |
    function_declarations_safe Functions FunctionHeader & this_function_not_declared |
    function_declarations_safe tIf tLeftBrace skip_part_of_this_scope |
    function_declarations_safe tWhile tLeftBrace skip_part_of_this_scope |
    e;

function_declarations_safe -> function_declarations & safeendingstring;

this_function_declared -> F this_function_declared |
    this_function_declared_here & F this_function_not_declared |
    this_function_declared_here & FunctionHeader tLeftBrace skip_part_of_this_scope |

this_function_not_declared ->
    this_function_not_declared_here & F this_function_not_declared |
    FunctionHeader |
this_function_notDeclaredHere FunctionHeader tLeftBrace skip_part_of_this_scope;

this_function_notDeclaredHere -> same_function_name & same_number_of_arguments;
this_function_notDeclaredHere ->
  different_function_name |
  same_function_name & different_number_of_arguments;

same_function_name -> C tLeftPar ListOfExpr tRightPar;
different_function_name -> Cneg tLeftPar ListOfExpr tRightPar;
same_number_of_arguments ->
  tId tLeftPar n_of_arg_0 tRightPar;
different_number_of_arguments ->
  tId tLeftPar n_of_arg_1 tRightPar |
  tId tLeftPar n_of_arg_1 tRightPar |
  tId tLeftPar n_of_arg_gt0 tRightPar |
  tId tLeftPar n_of_arg_gt tRightPar;
  n_of_arg_0 ->
    n_of_arg_equal |
    tRightPar anystringnotbeginningwithspace tLeftPar;
  n_of_arg_equal ->
    tId tComma n_of_arg_equal tComma E |
    tId tRightPar anystringnotbeginningwithspace tLeftPar E;
  n_of_arg_1t0 ->
    n_of_arg_1t0 tComma E |
    tRightPar anystringnotbeginningwithspace tLeftPar E;
  n_of_arg_1t ->
    n_of_arg_1t tComma E |
    n_of_arg_equal tComma E;
  n_of_arg_gt0 ->
    tId tComma n_of_arg_gt0 |
    tId tRightPar anystringnotbeginningwithspace tLeftPar;
  n_of_arg_gt ->
    tId tComma n_of_arg_gt |
    tId tComma n_of_arg_equal;
/

// Variable declarations */

all_variables_declared ->
  all_variables_declared_safe tId tLeftPar |
  all_variables_declared_safe Keyword tLeftPar |
  all_variables_declared_safe2 tLeftPar |
  all_variables_declared tSemicolon & does_not_end_with_var tSemicolon |
  all_variables_declared anypunctuationexceptleftparenssemicolon WS |
  all_variables_declared_safe tNum |
  all_variables_declared_safe Keyword |
  these_variables_notDeclared tSemicolon & all_variables_declared_safe VarSt |
  this_variableDeclared & all_variables_declared_safe tId |
  ListOfDistinctIds tRightPar |
  tRightPar;
all_variables_declared_safe ->
  all_variables_declared & safeendingstring;
all_variables_declared_safe2 ->
  all_variables_declared & anystring anypunctuation WS;
does_not_end_with_var ->
  anystring anypunctuationexceptcomma WS tId |
  safeendingstring tReturn _space tId |
  safeendingstring tNum |
  anystring anypunctuation WS;
this_variableDeclared ->
  C & tId tComma this_variable_notDeclared |
  C & tId tRightPar tLeftBrace notdeclared_inside_function |


```plaintext

// This is a grammar for parsing declarations in a programming language.
// The grammar is defined using Backus-Naur Form (BNF) notation.

// The grammar rules are as follows:

// 1. Variable declaration inside a function:
//    - `tId tComma this_variable_declared | tId tRightPar tLeftBrace declared_inside_function`:
//      - This rule allows for the declaration of a variable inside a function.
//    - `this_variable_not_declared`:
//      - Indicates that a variable has not been declared.

// 2. Variable declaration inside a function (continued):
//    - `Cneg & tId tComma this_variable_not_declared | Cneg & tId tRightPar tLeftBrace not_declared_inside_function`:
//      - Negates the declaration of a variable.
//    - `tRightPar tLeftBrace not_declared_inside_function;`:
//      - Ends the declaration of a variable inside a function.

// 3. Declared inside function:
//    - `declared_inside_function`:
//      - Marks a variable as declared inside a function.
//    - `S declared_inside_function | tLeftBrace declared_inside_function | tIf tLeftPar E tRightPar declared_inside_function_1 | tIf tLeftPar E tRightPar S tElse declared_inside_function_1 | tWhile tLeftPar E tRightPar declared_inside_function_1`:
//      - Multiple rules for declaring variables inside a function.

// 4. Not declared inside function:
//    - `not_declared_inside_function`:
//      - Marks a variable as not declared inside a function.
//    - `SNotvar not_declared_inside_function | tLeftBrace not_declared_inside_function | tIf tLeftPar E tRightPar not_declared_inside_function_1 | tIf tLeftPar E tRightPar S tElse not_declared_inside_function_1 | tWhile tLeftPar E tRightPar not_declared_inside_function_1`:
//      - Multiple rules for marking variables as not declared.

// 5. Declared in this statement:
//    - `declared_in_this_statement`:
//      - Marks a variable as declared in the current statement.
//    - `tId tComma declared_in_this_statement | C & hide_remaining_variables skip_part_of_this_scope`:
//      - Multiple rules for declaring variables in a statement.

// 6. Ignore remaining variables:
//    - `ignore_remaining_variables`:
//      - Ignores remaining variables in a statement.
//    - `tId tComma ignore_remaining_variables | tId tSemicolon`:

// 7. Not declared in this statement:
//    - `not_declared_in_this_statement`:
//      - Marks a variable as not declared in the current statement.
//    - `ignore_remaining_variables -> tId tComma not_declared_in_this_statement | tId tSemicolon`:

// 8. Other statements:
//    - `these_variables_not_declared`:
//      - Additional rules for handling variables.

// The grammar is designed to handle various scenarios of variable declaration and usage.
```

43