A Boolean grammar for a simple programming language

Alexander Okhotin
okhotin@cs.queensu.ca

Technical report 2004–478

School of Computing, Queen's University,
Kingston, Ontario, Canada K7L 3N6

March 2004

Abstract

A toy procedural programming language is defined, and a Boolean grammar for the set of well-formed programs in this language is constructed. This is apparently the first specification of a programming language entirely by a formal grammar.

Contents

1 Introduction

Formal methods have been employed in the specification of programming languages since the early days of computer science. Already the Algol 60 report [5, 6] used a context-free grammar to specify the complete lexical contents of the language, and, partially, its syntax. But still many actually syntactical requirements, such as the rules requiring the declaration of variables, were described in plain words together with the semantics of the language.

The hope of inventing a more sophisticated context-free grammar that would specify the entire syntax of Algol 60 was buried by Floyd [3], who used a formal language-theoretic argument based upon the non-context-freeness of the language $\{a^n b^n c^n \mid n \geqslant 0\}$ to prove that identifier checking cannot be performed by a context-free grammar, and hence the set of well-formed Algol 60 programs is not context-free.

The appealing prospects of giving a completely formal specification of the syntax of programming languages were not entirely abandoned, though. In the aftermath of Floyd's destructive result, many extensions of context-free grammars pertaining to cope with this problem were developed. One of the most promising attempts was represented by van Wijngaarden's two-level grammars [14, 15], which were actually used in the specification of Algol 68. Sadly, they were soon proved to be computationally universal, which demonstrated them hardly fit for practical use.

Among the formalisms proposed at that time that were more successful in evading Turing completeness, let us mention Aho's indexed grammars [1], which can denote all of the abstract non-context-free languages typically given to illustrate the non-context-freeness of programming languages [2]. However, it turned out that the possibility of denoting these constructs does not at all imply that a grammar for any programming language can in fact be constructed. Also, no polynomial-time parsing algorithm for indexed grammars has been found. As a result, indexed grammars remained an abstract model in the computation theory, while the syntax of new programming languages continued to be defined in the partially formal way introduced in the Algol 60 report [5, 6].

This paper gives a completely grammatical definition for the set of well-formed programs in a tiny procedural programming language. The recently introduced Boolean grammars [10] are used as the language specification formalism. The actual grammar is constructed and explained; this grammar has also been tested with a prototype parser generator [9], demonstrating square-time performance.

2 Definition of the language

Following is a complete definition of the syntax of the model language, i.e., a specification of the set of well-formed programs.

[I] Lexical conventions.

- **[I.1]** A program is a finite string over the following alphabet of 55 characters:
  - **[I.1.1]** 26 letters: `a`, . . . , `z`.
  - **[I.1.2]** 10 digits: `0`, . . . , `9`.
  - **[I.1.3]** 2 whitespace characters: space ␣ and newline ¶.
  - **[I.1.4]** 17 punctuators: `(`, `)`, {, }, "`,`", "`;`", `+`, `-`, `*`, `/`, `%`, `&`, `|`, `!`, `=`, `<`, `>`.
- **[I.2]** The following finite strings over the alphabet *[I.1]* are called *lexemes*:
  - **[I.2.1]** Five keywords: `var`, `if`, `else`, `while`, `return`.
  - **[I.2.2]** An identifier: a finite nonempty sequence of letters *[I.1.1]* and digits *[I.1.2]* that starts from a digit and does not coincide with any of the keywords *[I.2.1]*.
  - **[I.2.3]** A number: a finite nonempty sequence of digits *[I.1.2]*.
  - **[I.2.4]** 13 binary infix operators: `+`, `-`, `*`, `/`, `%`, `&`, `|`, `<`, `>`, `<=`, `>=`, `==`, `!=`.
  - **[I.2.5]** 2 unary prefix operators: `-`, `!`.
  - **[I.2.6]** Assignment operator: `=`.
  - **[I.2.7]** Parentheses: `(`, `)`.
  - **[I.2.8]** Compound statement delimiters: {, }.
  - **[I.2.9]** Comma and semicolon: "`,`", "`;`".
- **[I.3]** Separation of character stream into lexemes.
  - **[I.3.1]** The input *[I.1]* is processed from left to right.
  - **[I.3.2]** Each time the longest possible sequence of characters that forms a lexeme *[I.2]* is consumed, or a whitespace character *[I.1.3]* is discarded.
  - **[I.3.3]** Whitespace characters act as explicitly stated boundaries between the lexemes.
  - **[I.3.4]** According to the basic syntax defined below *[II]*, the only essential uses of whitespace in a well-formed program are:
    - in a declaration statement *[II.2.3]*, to separate the keyword `var` *[I.2.1]* from the next identifier *[I.2.2]*;
    - in a conditional statement *[II.2.4]*, to separate the keyword `else` *[I.2.1]* from a keyword, an identifier *[I.2.2]* or a number *[I.2.3]* that possibly start the inner statement *[II.2]*;
    - in a return statement *[II.2.6]*, to separate the keyword `return` *[I.2.1]* from the next identifier *[I.2.2]* or a number *[I.2.3]* that possibly opens an expression *[II.1]*.

[II] Basic syntax.

[II.1] Expressions.

[II.1.1] An identifier *[I.2.2]* is an expression.

[II.1.2] A number *[I.2.3]* is an expression.

[II.1.3] `(` e `)` is an expression for every expression e.

[II.1.4] Function call: f `(` e_1 `,` ... `,` e_k `)` is an expression for every identifier f *[I.2.2]* and expressions $e_1, \ldots, e_k$ ($k \geqslant 0$). This is a call of the function f with the arguments $e_1, \ldots, e_k$.

[II.1.5] $e_1\, op\, e_2$ is an expression for every binary operator op *[I.2.4]* and expressions e_1, e_2.

[II.1.6] $op\, e$ is an expression for every expression e and unary operator op *[I.2.5]*.

[II.1.7] Assignment: x `=` e is an expression for every identifier x *[I.2.2]* and expression e.

[II.2] Statements.

[II.2.1] Expression-statement: e `;` is a statement for every expression e *[II.1]*.

[II.2.2] Compound statement: `{` $s_1\, s_2\, \ldots\, s_k$ `}` is a statement for every $k \geqslant 0$ and for all statements $s_1, \ldots, s_k$.

[II.2.3] Declaration statement: `var` x_1 `,` ... `,`x_k `;` is a statement for every $k \geqslant 1$ and for all identifiers $x_1, \ldots, x_k$ *[II.1.1]*.

[II.2.4] Conditional statement: `if (` e `)` s and `if (` e `)` s `else` s' are statements for every expression e *[II.1]* and statements s, s'.

[II.2.5] Iteration statement: `while (` e `)` s is a statement for every expression e *[II.1]* and statement s.

[II.2.6] Return statement: `return` e `;` is a statement for every expression e *[II.1]*.

[II.3] Functions.

[II.3.1] Function header: `f (` x_1 `,` ... `,`x_k `)` is called a *function header* for every $k \geqslant 0$ and for all identifiers $f, x_1, \ldots, x_k$ *[II.1.1]*. The identifier f is the *name* of the function, the identifiers $x_1, \ldots, x_k$ are the *formal arguments* of the function.

[II.3.2] A *function declaration* is a function header *[II.3.1]* followed by a compound statement *[II.2.2]*. This compound statement is called the *body* of the function.

[II.4] A *program* is a finite sequence of function declarations *[II.3.2]*.

[III] Additional syntactical constraints on a program *[II.4]*.

[III.1] Function declarations.

[III.1.1] A program may not contain multiple declarations of functions *[II.3.2]* sharing the same name *[II.3.1]*.

[III.1.2] A program must contain a declaration of a function *[II.3.2]* with the name `main`, which must have exactly one formal argument *[II.3.1]*.

[III.2] Calls to functions.

[III.2.1] Any call to a function f *[II.1.4]* should be preceded by a function header *[II.3.1]* that opens a declaration of a function with the name f *[II.3.2]*.

[III.2.2] The number of arguments in a call to a function f *[II.1.4]* should match the number of formal arguments in the function header of f *[II.3.1]*.

[III.3] Variable declarations and their scope.

[III.3.1] *Variables* are declared in function headers *[II.3.1]* and in declaration statements *[II.2.3]*. Each identifier *[I.2.2]* in the corresponding lists declares a variable, and is called the *name* of the variable.

[III.3.2] With every variable declaration *[III.3.1]*, a certain nonempty set of statements *[II.2]* is associated, called *the scope of the declaration*.

[III.3.3] The scope of a declaration of a variable as a formal argument of a function *[II.3.1]* is the whole body of the function *[II.3.2]*.

[III.3.4] The scope of a declaration in a `var` statement *[II.2.3]* located within a compound statement *[II.2.2]* covers all statements in this compound statement starting from the `var` statement in question:

$$\{s_1 \ \ldots \ s_{i-1} \ \underbrace{\overbrace{\texttt{var}\ x_1\texttt{,}\ \ldots\ \texttt{,}x_k\texttt{;}}^{\text{a declaration}} \ s_{i+1} \ \ldots \ s_k}_{\text{the scope of this declaration}}\}$$

[III.3.5] The scope of a declaration in a `var` statement *[II.2.3]* located directly within a conditional *[II.2.4]* or an iteration statement *[II.2.5]* is confined to this `var` statement.

[III.4] The scopes *[III.3.2]* of any two variables sharing the same name *[III.3.1]* must be disjoint.

[III.5] Any use of an identifier x *[I.2.2]* as an atomic expression *[II.1.1]* or on the left hand side of an assignment operator *[II.1.7]* should be in the scope *[III.3]* of a variable with the name x *[III.3.1]*.

[III.6] Every function should return a value in the following sense:

[III.6.1] Let us call a statement *[II.2]* *properly returning* if it is either a `return` statement *[II.2.6]*, or a compound statement *[II.2.2]*, such that the last of the statements inside it is a properly returning statement, or a conditional statement *[II.2.4]* with an `else` clause, such that each of the alternatives is a properly returning statement.

[III.6.2] The body of every function *[II.3.2]* must be a properly returning statement.

This definition of the syntax is, strictly speaking, ambiguous: the precedence of operators is undefined and the dangling else ambiguity is unresolved. However, since its sole purpose is to define the set of well-formed programs, these issues can be ignored.

The semantics of the language is left completely undefined; the set of well-formed programs gets a purely syntactical definition. It is worth note that if the semantics *were* defined in the natural way, with the domain of the sole data type set to be the set of all positive integers, then the language would be Turing-complete.

3 Boolean grammars

This section gives a quick review of *Boolean grammars*, a generalization of context-free grammars that will be used to give a formal specification of the syntax of the model programming language defined in the previous section.

A Boolean grammar [10] is a quadruple $G = (\Sigma, N, P, S)$, where Σ and N are disjoint finite nonempty sets of terminal and nonterminal symbols respectively; P is a finite set of rules of the form

$$A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n \quad (m + n \geqslant 1,\ \alpha_i, \beta_i \in (\Sigma \cup N)^*), \tag{1}$$

while $S \in N$ is the start symbol of the grammar. For each rule (1), the objects $A \to \alpha_i$ and $A \to \neg\beta_j$ (for all i, j) are called *conjuncts* positive and negative respectively.

A Boolean grammar is called a conjunctive grammar [7], if negation is never used, i.e., $n = 0$ for every rule (1). It degrades to a familiar context-free grammar if neither negation nor conjunction are allowed, i.e., $m = 1$ and $n = 0$ for all rules.

Multiple rules $A \to \varphi_1, \ldots, A \to \varphi_k$ for a single nonterminal will be written using the standard notation

$$A \to \varphi_1 \mid \ldots \mid \varphi_k$$

The semantics of Boolean grammars is defined using language equations, similarly to the well-known characterization of the context-free grammars by another simpler class language equations [4]. Let $G = (\Sigma, N, P, S)$ be a Boolean grammar. The system of language equations associated with G is a resolved system of language equations over Σ in variables N, in which the equation for each variable $A \in N$ is

$$A = \bigcup_{A \to \alpha_1 \& \ldots \& \alpha_m \& \neg\beta_1 \& \ldots \& \neg\beta_n \in P} \left[\bigcap_{i=1}^{m} \alpha_i \cap \bigcap_{j=1}^{n} \overline{\beta_j} \right] \quad (\text{for all } A \in N) \tag{2}$$

A system (2) can have no solutions or multiple pairwise incomparable solutions; this leads to substantial problems with the formal definition of the semantics of Boolean grammars of the general form. These problems have been resoved using quite an elaborate technique [10]; however, in the practical cases (which exclude artistry like using a nonterminal A with a rule $A \to \neg A \& B$), a system associated with a grammar will have a unique solution $(L_1, \ldots, L_n)$, and the language $L_G(A_i)$ generated by every i-th nonterminal A can be defined as L_i then. The language of the grammar is $L(G) = L_G(S)$.

Let us give several examples of Boolean grammars for abstract languages. The following grammar [7] employs intersection to generate $\{a^n b^n c^n \mid n \geqslant 0\}$, which is precisely the language Floyd [3] used to prove the non-context-freeness of the programming languages:

$$\begin{aligned}
&S \to AB \& DC \\
&A \to aA \mid \varepsilon \\
&B \to bBc \mid \varepsilon \\
&C \to cC \mid \varepsilon \\
&D \to aDb \mid \varepsilon
\end{aligned}$$

Note that negation was not used in this grammar, but the use of intersection is essential, as the language is not context-free.

Another non-context-free language $\{ww \mid w \in \{a,b\}^*\}$ can be denoted, this time using negation, as follows [10]:

$$\begin{array}{ll}
S \to \neg AB \& \neg BA \& C & \\
A \to XAX & C \to XXC \\
A \to a & C \to \varepsilon \\
B \to XBX & X \to a \\
B \to b & X \to b
\end{array}$$

A very similar language, $\{wcw \mid w \in \{a,b\}^*\}$, can be denoted even without negation [7]:

$$\begin{aligned}
&S \to C \& D \\
&C \to aCa \mid aCb \mid bCa \mid bCb \mid c \\
&D \to aA \& aD \mid bB \& bD \mid cE \\
&A \to aAa \mid aAb \mid bAa \mid bAb \mid cEa \\
&B \to aBa \mid aBb \mid bBa \mid bBb \mid cEb \\
&E \to aE \mid bE \mid \varepsilon
\end{aligned}$$

The next grammar [10] defines the language $\{a^{2^n} \mid n \geqslant 0\}$, one more well-known non-context-free language:

$$\begin{array}{llll}
S \to X \& \neg aX & X \to aX'X' & Y \to aaY'Y' & Z \to Y \\
S \to \neg X \& aX & X' \to \neg X''X'' & Y' \to \neg Y''Y'' \& T & Z \to aY \\
S \to Z \& \neg aZ & X'' \to \neg X'''X''' & Y'' \to \neg Y'''Y''' \& T & T \to aaT \\
S \to \neg Z \& aZ & X''' \to \neg X & Y''' \to \neg Y \& T & T \to \varepsilon
\end{array}$$

Note that all the abstract languages considered above can be as well generated by indexed grammars [1]. However, Boolean grammars have an advantage of their own, that of being parsable in at most cubic time [10], which is as fast as the best practically usable algorithms for general context-free grammars.

Although both indexed and Boolean grammars apparently generate the same noteworthy abstract languages, their descriptive means in fact differ: one have stacks attached to nonterminals, the other have logical connectives. The descriptive machinery native to Boolean grammars shall now be used to specify the syntax of a programming language.

4 A Boolean grammar for the language

Let the alphabet of terminal symbols be $\Sigma = \{$`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`, `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, ␣, "`(`", "`)`", `{`, `}`, "`,`", "`;`", `+`, `-`, `*`, `/`, `%`, "`&`", "`|`", `!`, `=`, `<`, `>`$\}$. This is the alphabet of the model programming language *[I.1]*, in which the space and the newline characters are merged into a single symbol ␣, reflecting the fact that the definition of the language makes no distinction between these symbols *[I.3.2]*.

The promised Boolean grammar over this alphabet will now be constructed. It is convenient to classify its nonterminals into the following five groups:

The start symbol $\{S\}$ corresponds to the notion of a well-defined program.

The alphabet group defines basic types of strings over the alphabet that are referred to in the main grammar. There are 11 nonterminals in this group: anyletter, anydigit, anyletterdigit, anypunctuator, anypunctuatorexceptrightpar, anypunctuatorexceptbraces, anychar, anystring, anyletterdigits, anystringwithoutbraces, safeendingstring.

The lexical group defines all types of lexemes and handles the whitespace *[I]*, something that is typically done by a finite-automaton-based lexical analyzer. The nonterminals in this group are: WS, tVar, tIf, tElse, tWhile, tReturn, Keyword, tId, tId-aux, tNum, tNum-aux, tPlus, tMinus, tStar, tSlash, tMod, tAnd, tOr, tLessThan, tGreaterThan, tLessEqual, tGreaterEqual, tEqual, tNotEqual, BinaryOp, tUnaryMinus, tNot, UnaryOp, tLeftPar, tRightPar, tLeftBrace, tRightBrace, tAssign, tSemicolon, tComma – altogether 35 nonterminals.

The main syntax group contains nonterminals that represent the main syntactical notions of the language *[II]*. These are the nonterminals **Expr ExprFunctionCall**, **ListOfExpr**, **ListOfExpr1**, **Statement**, **ExprSt**, **CompoundSt**, **Statements**, **VarSt**, **CondSt**, **IterationSt**, **ReturnSt**, **Function**, **FunctionHeader**, **FunctionArguments**, **ListOfIds** and **Functions** – 17 in total.

Identifier comparison group defines a nonterminal C that generates the language

$$\{wxw \mid w \text{ is a nonempty string that consists of letters and digits;} \\ x \text{ is nonempty, it starts and ends with anything but letters or digits}\} \tag{3}$$

The following auxiliary nonterminals are used to define this language: *Clen*, *Citerate*, *Cmid*, C_a, $\ldots$, C_z, C_0, $\ldots$, C_9. In total, there are $4 + 36 = 40$ nonterminals in this group.

Advanced syntax group consists of several independent components that check different syntactical constraints *[III]*. It contains the following 19 nonterminals: *duplicate-functions*, *duplicate-functions-here*, *all-functions-defined*, *all-functions-defined-skip*, *check-function-call*, *check-function-call-here*, *check-name*, *check-number-of-arguments*, *n-of-arg-match*, *all-variables-defined*, *all-variables-defined-skip*, *this-variable-defined*, *this-variable-defined2*, *this-variable-defined3*, *this-variable-defined3-afterskip*, *this-variable-defined4*, *these-variables-not-defined*, *has-main-function* and *returns-a-value*.

The names of nonterminals from different groups are distinguished with different fonts. Define $N = \{S\} \cup N_{alph} \cup N_{lex} \cup N_{main} \cup N_{wcw} \cup N_{adv}$. The cardinality of this set is $|N| = 1 + 11 + 35 + 17 + 40 + 19 = 123$.

Here are the simple rules for the nonterminals from the alphabet group:

anyletter → `a` | … | `z`
anydigit → `0` | … | `9`
anyletterdigit → anyletter | anydigit
anypunctuator → "`(`" | "`)`" | { | } | "`,`" | "`;`" | `+` | `-` | `*` | `/` | "`&`" | "`|`" | `!` |
`=` | `<` | `>` | `%`
anypunctuatorexceptrightpar → "`(`" | { | } | "`,`" | "`;`" | `+` | `-` | `*` | `/` | "`&`" |
"`|`" | `!` | `=` | `<` | `>` | `%`
anypunctuatorexceptbraces → "`(`" | "`)`" | "`,`" | "`;`" | `+` | `-` | `*` | `/` | "`&`" |
"`|`" | `!` | `=` | `<` | `>` | `%`
anychar → ␣ | anyletter | anydigit | anypunctuator
anystring → anystring anychar | ε
anyletterdigits → anyletterdigits anyletterdigit | ε
anystringwithoutbraces → anystringwithoutbraces ␣ |
anystringwithoutbraces anyletter |
anystringwithoutbraces anydigit |
anystringwithoutbraces anypunctuatorexceptbraces |
ε
safeendingstring → anystring anypunctuator | anystring ␣ | ε

Rules for the lexical group do what is usually done by lexical analyzers. First, define the whitespace:

$\mathrm{WS} \to \mathrm{WS}\ \text{␣} \mid \varepsilon$ *[I.3.2]*

Now for every type of lexemes define the set of valid lexemes of this type possibly followed by whitespace.

$\mathrm{Keyword} \to \mathrm{tVar} \mid \mathrm{tIf} \mid \mathrm{tElse} \mid \mathrm{tWhile} \mid \mathrm{tReturn}$ *[I.2.1]*
$\mathrm{tVar} \to \mathtt{v\ a\ r}\ \mathrm{WS}$
$\mathrm{tIf} \to \mathtt{i\ f}\ \mathrm{WS}$
$\mathrm{tElse} \to \mathtt{e\ l\ s\ e}\ \mathrm{WS}$
$\mathrm{tWhile} \to \mathtt{w\ h\ i\ l\ e}\ \mathrm{WS}$
$\mathrm{tReturn} \to \mathtt{r\ e\ t\ u\ r\ n}\ \mathrm{WS}$

Let us use the extended descriptive power of Boolean grammars generously, and afford the luxury of specifying the set of identifiers precisely according to its definition:

$\mathrm{tId} \to \mathrm{tId\text{-}aux}\ \mathrm{WS}\ \&\ \neg\mathrm{Keyword}$ *[I.2.2]*
$\mathrm{tId\text{-}aux} \to \mathrm{anyletter} \mid \mathrm{tId\text{-}aux}\ \mathrm{anyletter} \mid \mathrm{tId\text{-}aux}\ \mathrm{anydigit}$

This could have been done by simply simulating a fairly small finite automaton, though. The rest of the rules for the lexical group are dull:

$\mathrm{tNum} \to \mathrm{tNum\text{-}aux}\ \mathrm{WS}$ *[I.2.3]*
$\mathrm{tNum\text{-}aux} \to \mathrm{tNum\text{-}aux}\ \mathrm{anydigit} \mid \mathrm{anydigit}$
$\mathrm{BinaryOp} \to \mathrm{tPlus} \mid \mathrm{tMinus} \mid \mathrm{tStar} \mid \mathrm{tSlash} \mid \mathrm{tMod} \mid \mathrm{tAnd} \mid$
$\quad \mathrm{tOr} \mid \mathrm{tLessThan} \mid \mathrm{tGreaterThan} \mid \mathrm{tLessEqual} \mid$
$\quad \mathrm{tGreaterEqual} \mid \mathrm{tEqual} \mid \mathrm{tNotEqual}$ *[I.2.4]*
$\mathrm{tPlus} \to \mathtt{+}\ \mathrm{WS}$
$\mathrm{tMinus} \to \mathtt{-}\ \mathrm{WS}$
$\mathrm{tStar} \to \mathtt{*}\ \mathrm{WS}$
$\mathrm{tSlash} \to \mathtt{/}\ \mathrm{WS}$
$\mathrm{tMod} \to \mathtt{\%}\ \mathrm{WS}$
$\mathrm{tAnd} \to \text{“}\mathtt{\&}\text{”}\ \mathrm{WS}$
$\mathrm{tOr} \to \text{“}\mathtt{|}\text{”}\ \mathrm{WS}$
$\mathrm{tLessThan} \to \mathtt{<}\ \mathrm{WS}$
$\mathrm{tGreaterThan} \to \mathtt{>}\ \mathrm{WS}$
$\mathrm{tLessEqual} \to \mathtt{<\ =}\ \mathrm{WS}$
$\mathrm{tGreaterEqual} \to \mathtt{>\ =}\ \mathrm{WS}$
$\mathrm{tEqual} \to \mathtt{=\ =}\ \mathrm{WS}$
$\mathrm{tNotEqual} \to \mathtt{!\ =}\ \mathrm{WS}$
$\mathrm{UnaryOp} \to \mathrm{tUnaryMinus} \mid \mathrm{tNot}$ *[I.2.5]*
$\mathrm{tUnaryMinus} \to \mathtt{-}\ \mathrm{WS}$
$\mathrm{tNot} \to \mathtt{!}\ \mathrm{WS}$
$\mathrm{tAssign} \to \mathtt{=}\ \mathrm{WS}$ *[I.2.6]*
$\mathrm{tLeftPar} \to \text{“}\mathtt{(}\text{”}\ \mathrm{WS}$ *[I.2.7]*

tRightPar → ")" WS
tLeftBrace → { WS *[I.2.8]*
tRightBrace → } WS
tSemicolon → ";" WS *[I.2.9]*
tComma → "," WS

Next there comes the syntax group, which is composed mostly like a typical context-free grammar that partially defines the syntax of a programming language. However, note how the nonterminal **Function** invokes nonterminals from the advanced syntax group to check the variable scoping rules *[III.3]* and the placement of return statements *[III.6]*.

Expr → tId *[II.1.1]*
Expr → tNum *[II.1.2]*
Expr → tLeftPar **Expr** tRightPar *[II.1.3]*
Expr → **ExprFunctionCall** *[II.1.4]*
ExprFunctionCall → tId tLeftPar **ListOfExpr** tRightPar
ListOfExpr → **ListOfExpr1**
ListOfExpr → ε
ListOfExpr1 → **ListOfExpr1** tComma **Expr** | **Expr**
Expr → **Expr** BinaryOp **Expr** *[II.1.5]*
Expr → UnaryOp **Expr** *[II.1.6]*
Expr → tId tAssign **Expr** *[II.1.7]*

Statement → **ExprSt** | **CompoundSt** | **VarSt** | **CondSt** |
IterationSt | **ReturnSt** *[II.2]*
ExprSt → **Expr** tSemicolon *[II.2.1]*
CompoundSt → tLeftBrace **Statements** tRightBrace *[II.2.2]*
Statements → **Statements** **Statement** | ε
VarSt → tVar ␣ **ListOfIds** tSemicolon *[II.2.3]*
CondSt → tIf tLeftPar **Expr** tRightPar **Statement** | *[II.2.4]*
tIf tLeftPar **Expr** tRightPar **Statement** tElse **Statement**
IterationSt → tWhile tLeftPar **Expr** tRightPar **Statement** *[II.2.5]*
ReturnSt → tReturn **Expr** tSemicolon *[II.2.6]*

FunctionHeader → tId **FunctionArguments** *[II.3.1]*
FunctionArguments → tLeftPar **ListOfIds** tRightPar |
tLeftPar tRightPar
ListOfIds → **ListOfIds** tComma tId | tId
Function → **FunctionHeader** **CompoundSt** & *[II.3.2]*
tId tLeftPar *all-variables-defined* & *[III.4] [III.5]*
FunctionHeader *returns-a-value* *[III.6]*

Functions → **Functions** **Function** | ε *[II.4]*

The identifier comparison group defines the language (3), which is extensively used by the advanced syntax group. These rules implement the idea of the grammar for the abstract language $\{wcw \mid w \in \{a,b\}^*\}$ [7], which is reproduced in Section 3.

$$
\begin{aligned}
&C \to Clen \mathbin{\&} Citerate \mid C \text{ WS}\\
&Clen \to \text{anyletterdigit } Clen \text{ anyletterdigit} \mid\\
&\qquad\qquad \text{anyletterdigit } Cmid \text{ anyletterdigit}\\
&C_a \to \text{anyletterdigit } C_a \text{ anyletterdigit} \mid \texttt{a} \text{ anyletterdigits } Cmid\\
&\quad \vdots\\
&C_z \to \text{anyletterdigit } C_z \text{ anyletterdigit} \mid \texttt{z} \text{ anyletterdigits } Cmid\\
&C_0 \to \text{anyletterdigit } C_0 \text{ anyletterdigit} \mid \texttt{0} \text{ anyletterdigits } Cmid\\
&\quad \vdots\\
&C_9 \to \text{anyletterdigit } C_9 \text{ anyletterdigit} \mid \texttt{9} \text{ anyletterdigits } Cmid\\
&Citerate \to C_a \ \texttt{a} \mathbin{\&} \text{Citerate } \texttt{a} \mid \ldots \mid C_z \ \texttt{z} \mathbin{\&} \text{Citerate } \texttt{z} \mid\\
&\qquad\qquad C_0 \ \texttt{0} \mathbin{\&} \text{Citerate } \texttt{0} \mid \ldots \mid C_9 \ \texttt{9} \mathbin{\&} \text{Citerate } \texttt{9} \mid\\
&\qquad\qquad \text{anyletterdigits } Cmid\\
&Cmid \to \text{anypunctuator anystring anypunctuator} \mid \text{␣ anystring ␣} \mid\\
&\qquad\qquad \text{␣ anystring anypunctuator} \mid \text{anypunctuator anystring ␣} \mid\\
&\qquad\qquad \text{anypunctuator} \mid \text{␣}
\end{aligned}
$$

Finally we reach the advanced syntax group, which specifies the rules *[III]* that are traditionally checked by manually written program code rather than using language specification formalisms of any kind. Most of these syntactical conditions are inherently non-context-free.

Using the negation native to Boolean grammars, the rule on the uniqueness of function definition *[III.1.1]* can be most naturally checked by first expressing the condition that a program *has* multiply declared functions and then by putting $\neg$ over it. The existence of duplicate function declarations is specified as follows:

$$
\begin{aligned}
&\textit{duplicate-functions} \to \textbf{Functions } \textit{duplicate-functions-here} \textbf{ Functions}\\
&\textit{duplicate-functions-here} \to C \ \textbf{FunctionArguments CompoundSt}
\end{aligned}
$$

A group of functions where this error occurs is "chosen" by the nonterminal *duplicate-functions*, while *duplicate-functions-here* invokes the nonterminal C from the identifier comparison group to ensure that the names of the first and the last functions in this group coincide. Thus a text of a program will be generated by *duplicate-functions* only if two or more functions sharing the same name are found. This is explained in Figure 1.

The requirements on the function `main` can be specified by simple context-free rules:

$$
\begin{aligned}
\textit{has-main-function} \to\ & \textbf{Functions}\\
& \texttt{m a i n} \text{ tLeftPar tId tRightPar } \textbf{CompoundSt}\\
& \textbf{Functions} \ \textit{[III.1.2]}
\end{aligned}
$$

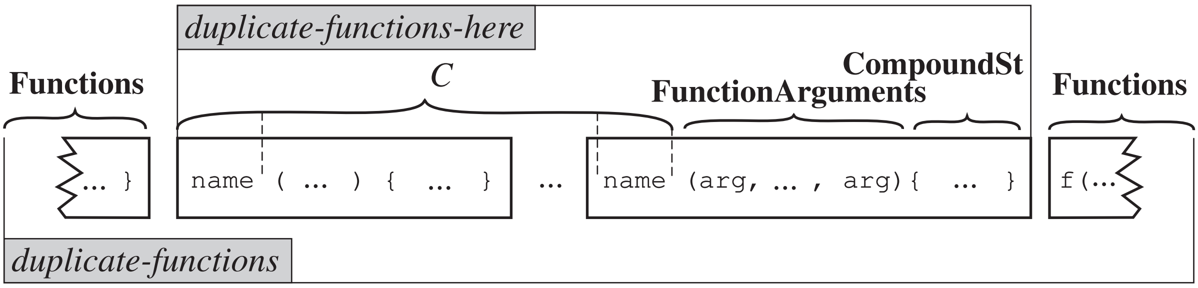

Figure 1: How *duplicate-functions* finds a pair of declarations.

Checking the correctness of all function calls *[III.2]* requires entering function bodies and analyzing the program lexeme by lexeme and even symbol by symbol.

all-functions-defined $\rightarrow$ *all-functions-defined* ␣ |
  all-functions-defined anypunctuatorexceptrightpar |
  all-functions-defined-skip Keyword |
  all-functions-defined-skip tId |
  all-functions-defined-skip tNum |
  check-function-call & *all-functions-defined* tRightPar |
  ε
all-functions-defined-skip $\rightarrow$ *all-functions-defined* & safeendingstring
check-function-call $\rightarrow$ **Functions FunctionHeader** |
  anystring & ¬safeendingstring **ExprFunctionCall** |
  Functions *check-function-call-here*
check-function-call-here $\rightarrow$ *check-name* & *check-number-of-arguments*
check-name $\rightarrow$ *C* tLeftPar **ListOfExpr** tRightPar *[III.2.1]*
check-number-of-arguments $\rightarrow$ tId tLeftPar
  n-of-arg-match tRightPar *[III.2.2]*
n-of-arg-match $\rightarrow$ tId *n-of-arg-match* **Expr** |
  tComma *n-of-arg-match* tComma |
  tRightPar anystring tLeftPar

The nonterminal *all-functions-defined* considers all prefixes of the program and recognizes function calls *[II.1.4]* at the right end of the strings being considered. For each function call thus found, the nonterminal *check-function-call* is invoked, which attempts to match the call to a preceding declaration of a function. While the nonterminal *check-function-call* itself merely considers candidates for being such a declaration, another nonterminal *check-function-call-here* does the actual matching (see Figure 2), using the nonterminal *check-name* to ensure that the function defined at the beginning in the current substring is the same function as the one called at the end of the current substring, and at the same time the nonterminal *check-number-of-arguments* checks that

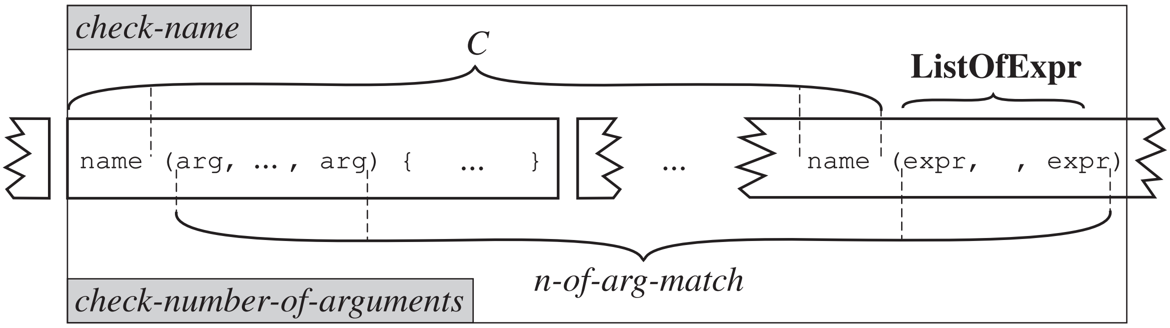

Figure 2: How *check-function-call-here* matches a declaration *[II.3.1]* to a use *[II.1.4]*.

the number of arguments in the call corresponds to the number of formal arguments in the declaration.

Now consider the rules for the nonterminal *all-variables-defined*, which has already been used in the rules for the syntax group.

all-variables-defined → *all-variables-defined* ␣ |
  all-variables-defined anypunctuator & ¬safeendingstring **VarSt** |
  all-variables-defined-skip tId tLeftPar |
  all-variables-defined-skip tNum |
  all-variables-defined-skip Keyword |
  these-variables-not-defined tSemicolon &
    all-variables-defined-skip **VarSt** |
  this-variable-defined & *all-variables-defined-skip* tId |
  these-variables-not-defined tRightPar
all-variables-defined-skip → *all-variables-defined* & safeendingstring
this-variable-defined → *C* | tId tComma *this-variable-defined* |
  tId tRightPar tLeftBrace *this-variable-defined2*
this-variable-defined2 → **Statement** *this-variable-defined2* |
  tLeftBrace *this-variable-defined2* |
  tIf tLeftPar **Expr** tRightPar
    tLeftBrace *this-variable-defined2* |
  tIf tLeftPar **Expr** tRightPar **Statement** tElse
    tLeftBrace *this-variable-defined2* |
  tWhile tLeftPar **Expr** tRightPar
    tLeftBrace *this-variable-defined2* |
  tVar ␣ *this-variable-defined3* |
  tIf tLeftPar **Expr** tRightPar tVar ␣ *this-variable-defined4* |
  tIf tLeftPar **Expr** tRightPar **Statement** tElse
    tVar ␣ *this-variable-defined4* |
  tWhile tLeftPar **Expr** tRightPar tVar ␣ *this-variable-defined4*

this-variable-defined3 → tId tComma *this-variable-defined3* |
C & anystringwithoutbraces *this-variable-defined3-afterskip*
this-variable-defined3-afterskip → anystringwithoutbraces
Statement *this-variable-defined3-afterskip* |
tLeftBrace *this-variable-defined3-afterskip* |
this-variable-defined4 → tId tComma *this-variable-defined4* |
C & **ListOfIds**;

The nonterminal *all-variables-defined* considers all prefixes of the body of a function and recognizes declarations statements *[II.2.3]* and references to variables *[II.1.1]* at the right ends of these prefixes:

- Whenever an atomic expression *[II.1.1]* is found at the right, the nonterminal *this-variable-defined* is used to match it to a declaration of a variable *[III.3]* at the left. First it tries to find such a declaration among the formal arguments of the function *[III.3.3]*. If failed, the nonterminal *this-variable-defined2* starts looking for a declaration in a `var` statement, observing the scoping rules. `var` statements located inside compound statements *[III.3.4]* are handled by the nonterminal *this-variable-defined3*, while *this-variable-defined4* takes care of declarations directly inside `if` and `while` statements *[III.3.5]*.

- When a declaration statement *[II.2.3]* appears at the right, the nonterminal *these-variables-not-defined* is used to ensure that none of the declared variables is in the scope of a variables with the same name.

Consider the rules for *these-variables-not-defined*. Here negation comes particularly handy, since the condition that an identifier is in the scope of a variable with the same name has already been expressed by the nonterminal *this-variable-defined*, and it suffices to invert it.

these-variables-not-defined → *these-variables-not-defined* tComma tId &
¬*this-variable-defined* |
safeendingstring tVar ␣ tId & ¬*this-variable-defined* |
tId | ε

The condition that a function returns a value can be checked by simple context-free rules as follows:

returns-a-value → tLeftBrace **Statements** *returns-a-value* tRightBrace |
tIf tLeftPar **Expr** tRightPar *returns-a-value*
tElse *returns-a-value* |
ReturnSt

Finally, a single rule for the start symbol specifies what a well-formed program is:

S → WS **Functions** & *[II.4]*
WS *all-functions-defined* & *[III.2]*
WS *has-main-function* & *[III.1.2]*
¬WS *duplicate-functions* *[III.1.1]*

This completes the grammar, which consists of 123 nonterminals and 368 rules, and specifies the language as defined in Section 2. Since it is known that every language generated by a Boolean grammar can be parsed in time $O(n^3)$, this grammar directly yields a polynomial-time correctness checker for the model programming language.

The given grammar has in fact been extensively tested using a parser generator [9], which implements a yet unpublished extension of the Generalized LR parsing algorithm [13, 8] for Boolean grammars [12]. The observed parsing time was $O(n^2)$. The details of the tests and the actual performance results are given in Appendixes A through C.

5 Conclusion

A specification of a programming language by a formal grammar that belongs a computationally feasible class of grammars was obtained, apparently for the first time ever.

The given Boolean grammar for the model language was admittedly composed under an influence of the standard compiler construction methods that separate lexical analysis, context-free syntax backbone and non-context-free syntactical restrictions [2]. Perhaps a much better grammar for the same model language could be constructed using some other principles that would make better use of the expanded descriptive means of Boolean grammars. Realizing such principles and coming up with improved grammar composition techniques might be of interest.

Another obvious topic for further study would be to investigate the formal properties of Boolean grammars [10] and to develop parsing algorithms for them. Some algorithms are already being researched [11, 12], and their prototype implementations are available [9]. It will require more work to make these algorithms practical.

One more direction for future work is to use the experience of this study and come up with a new grammar formalism with better properties, which would still allow to specify simple programming languages. That might be an efficiently parsable and still sufficiently expressive subclass of Boolean grammars, or that might be an entirely new formalism.

In any case, it was shown that formal grammars are better suited for specifying the syntax of programming languages than it has been believed for the last three decades, and thus deserve renewed attention.

References

[1] A. V. Aho, "Indexed grammars", *Journal of the ACM*, 15:4 (1968), 647–671.

[2] A. V. Aho, R. Sethi, J. D. Ullman, *Compilers: principles, techniques and tools*, Addison-Wesley, Reading, Mass., 1986.

[3] R. W. Floyd, "On the non-existence of a phrase structure grammar for ALGOL 60", *Communications of the ACM*, 5 (1962), 483–484.

[4] S. Ginsburg, H. G. Rice, "Two families of languages related to ALGOL", *Journal of the ACM*, 9 (1962), 350–371.

[5] P. Naur (Ed.), J. W. Backus, J. H. Wegstein, A. van Wijngaarden, M. Woodger, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, "Report on the algorithmic language ALGOL 60", *Communications of the ACM*, 3:5 (1960), 299–314.

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

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

[8] A. Okhotin, "LR parsing for conjunctive grammars", *Grammars*, 5:2 (2002), 81–124.

[9] A. Okhotin, "Whale Calf, a parser generator for conjunctive grammars", *Implementation and Application of Automata* (Proceedings of CIAA 2002, Tours, France, July 3–5, 2002), LNCS 2608, 213–220. The software is available at

`http://www.cs.queensu.ca/home/okhotin/whalecalf/`

[10] A. Okhotin, "Boolean grammars", *Information and Computation*, to appear.

- Preliminary version in: *Developments in Language Theory* (Proceedings of DLT 2003, Szeged, Hungary, July 7–11, 2003), LNCS 2710, 398–410.

[11] A. Okhotin, "An extension of recursive descent parsing for Boolean grammars", Technical Report 2004–475, School of Computing, Queen's University, Kingston, Ontario, Canada.

[12] A. Okhotin, "LR parsing for Boolean grammars", in progress.

[13] M. Tomita, “An efficient augmented context-free parsing algorithm”, *Computational Linguistics*, 13:1 (1987), 31–46.

[14] A. van Wijngaarden, “Orthogonal design and description of a formal language”, Technical Report MR 76, Mathematisch Centrum, Amsterdam, 1965.

[15] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, R. G. Fisker, “Revised report on the algorithmic language ALGOL 68”, *Acta Informatica*, 5 (1975), 1–236.

Appendix.

A A parser

Using the Whale Calf parser generator [9], a parser for the model programming language has been automatically created from the Boolean grammar given in Section 4.

The following main module was used, which simply loads the input file (translating the character set), invokes the generated parser, and afterwards prints the result of recognition.

mlc.cpp

```
#include <fstream>
#include <vector>
#include "ml_parser.h"
using namespace std;

void preprocessor(istream &is, vector<int> &v);

int main(int argc, char *argv[])
{
    ifstream is(argv[1]);
    if(!is) { cerr << "Unable to open "
                   << argv[1] << "\n";
              return 1; }
    vector<int> v;
    preprocessor(is, v);
    cout << "The input is " << v.size()
         << " symbols long.\n";

    WhaleCalfLRParser whale_calf(Ml_parser::grammar);
    cout << whale_calf.copyright_notice() << "\n";
    whale_calf.read(v);
    bool result=whale_calf.recognize();
    if(result)
        cout << "This is a well-formed program.\n";
    else
        cout << "Syntax errors found.\n";
    return result;
}

void preprocessor(istream &is, vector<int> &v)
{
    for(int i=0;; i++)
    {
        char c=is.get();
        if(is.eof()) return;

        int x=0;
        if(c==' ' || c=='\t' || c=='\n')
            x=Ml_parser::Terminal_space;
        else if(c>='a' && c<='z')
            x=Ml_parser::Terminal_a+int(c-'a');
        else if(c>='0' && c<='9')
            x=Ml_parser::Terminal_0+int(c-'0');
        else if(c=='(')
            x=Ml_parser::Terminal_leftpar;
        else if(c==')')
            x=Ml_parser::Terminal_rightpar;
        else if(c=='{')
            x=Ml_parser::Terminal_leftbrace;
        else if(c=='}')
            x=Ml_parser::Terminal_rightbrace;
        else if(c==',')
            x=Ml_parser::Terminal_comma;
        else if(c==';')
            x=Ml_parser::Terminal_semicolon;
        else if(c=='+')
            x=Ml_parser::Terminal_plus;
        else if(c=='-')
            x=Ml_parser::Terminal_minus;
        else if(c=='*')
            x=Ml_parser::Terminal_star;
        else if(c=='/')
            x=Ml_parser::Terminal_slash;
        else if(c=='%')
            x=Ml_parser::Terminal_percent;
        else if(c=='&')
            x=Ml_parser::Terminal_and;
        else if(c=='|')
            x=Ml_parser::Terminal_or;
        else if(c=='!')
            x=Ml_parser::Terminal_excl;
        else if(c=='=')
            x=Ml_parser::Terminal_eq;
        else if(c=='<')
            x=Ml_parser::Terminal_lt;
        else if(c=='>')
            x=Ml_parser::Terminal_gt;
        else
        {
            cerr << "Invalid symbol at offset "
                 << i << ".\n";
            throw false;
        }

        v.push_back(x);
    }
}
```

The following grammar description file for Whale Calf was used to specify the parser. The grammar is identical to the one given in Section 4.

ml.whc

```
algorithm=LR;

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;

S -> WS Functions &
    WS all_functions_defined &
    WS has_main_function &
    ~WS duplicate_functions;

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;
anypunctuatorexceptrightpar -> _leftpar |
    _leftbrace | _rightbrace | _comma | _semicolon |
    _plus | _minus | _star | _slash | _and | _or |
    _excl | _eq | _lt | _gt | _percent;
anypunctuatorexceptbraces -> _leftpar | _rightpar |
    _comma | _semicolon | _plus | _minus | _star |
    _slash | _and | _or | _excl | _eq | _lt | _gt |
    _percent;
anychar -> _space | anyletter | anydigit |
    anypunctuator;
anystring -> anystring anychar | e;
anyletterdigits -> anyletterdigits anyletterdigit |
    e;
anystringwithoutbraces ->
    anystringwithoutbraces _space |
    anystringwithoutbraces anyletter |
    anystringwithoutbraces anydigit |
    anystringwithoutbraces
        anypunctuatorexceptbraces |
    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;

ListOfIds -> ListOfIds tComma tId | 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;
BinaryOp -> tPlus | tMinus | tStar | tSlash | tMod |
    tAnd | tOr | tLessThan | tGreaterThan |
    tLessEqual | tGreaterEqual | tEqual | tNotEqual;

tUnaryMinus -> _minus WS;
tNot -> _excl WS;
UnaryOp -> tUnaryMinus | tNot;

tLeftPar -> _leftpar WS;
tRightPar -> _rightpar WS;
tLeftBrace -> _leftbrace WS;
tRightBrace -> _rightbrace WS;
tAssign -> _eq WS;
tSemicolon -> _semicolon WS;
tComma -> _comma WS;

Expr -> tId;
Expr -> tNum;
Expr -> tLeftPar Expr tRightPar;
Expr -> ExprFunctionCall;
Expr -> Expr BinaryOp Expr;
Expr -> UnaryOp Expr;
Expr -> tId tAssign Expr;

ExprFunctionCall ->
    tId tLeftPar ListOfExpr tRightPar;
ListOfExpr -> ListOfExpr1;
ListOfExpr -> e;
ListOfExpr1 -> ListOfExpr1 tComma Expr | Expr;

Statement -> ExprSt | CompoundSt | VarSt | CondSt |
    IterationSt | ReturnSt;
ExprSt -> Expr tSemicolon;
CompoundSt -> tLeftBrace Statements tRightBrace;
Statements -> Statements Statement | e;
VarSt -> tVar _space ListOfIds tSemicolon;
CondSt -> tIf tLeftPar Expr tRightPar Statement |
    tIf tLeftPar Expr tRightPar Statement
        tElse Statement;
IterationSt -> tWhile tLeftPar Expr tRightPar
        Statement;
ReturnSt -> tReturn Expr tSemicolon;

Function -> FunctionHeader CompoundSt &
    tId tLeftPar all_variables_defined &
    FunctionHeader returns_a_value;
FunctionHeader -> tId FunctionArguments;
FunctionArguments -> tLeftPar ListOfIds tRightPar |
    tLeftPar tRightPar;
```

```
Functions -> Functions Function | e;

safeendingstring -> anystring anypunctuator |
    anystring _space | e;

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

duplicate_functions ->
    Functions duplicate_functions_here Functions;
duplicate_functions_here ->
    C FunctionArguments CompoundSt;

all_functions_defined -> all_functions_defined
        anypunctuatorexceptrightpar |
    all_functions_defined _space |
    all_functions_defined_skip Keyword |
    all_functions_defined_skip tId |
    all_functions_defined_skip tNum |
    check_function_call &
        all_functions_defined tRightPar |
    e;
all_functions_defined_skip ->
    all_functions_defined & safeendingstring;
```

```
check_function_call -> Functions FunctionHeader |
    anystring & ~safeendingstring ExprFunctionCall |
    Functions check_function_call_here;

check_function_call_here ->
    check_name & check_number_of_arguments;
check_name -> C tLeftPar ListOfExpr tRightPar;
check_number_of_arguments ->
    tId tLeftPar n_of_arg_match tRightPar;
n_of_arg_match -> tId n_of_arg_match Expr |
    tComma n_of_arg_match tComma |
    tRightPar anystring tLeftPar;

all_variables_defined ->
    all_variables_defined _space |
    all_variables_defined anypunctuator &
        ~safeendingstring VarSt |
    all_variables_defined_skip tId tLeftPar |
    all_variables_defined_skip tNum |
    all_variables_defined_skip Keyword |
    these_variables_not_defined tSemicolon &
        all_variables_defined_skip VarSt |
    this_variable_defined &
        all_variables_defined_skip tId |
    these_variables_not_defined tRightPar;
all_variables_defined_skip ->
    all_variables_defined & safeendingstring;

this_variable_defined -> C |
    tId tComma this_variable_defined |
    tId tRightPar tLeftBrace this_variable_defined2;
this_variable_defined2 ->
    Statement this_variable_defined2 |
    tLeftBrace this_variable_defined2 |
    tIf tLeftPar Expr tRightPar tLeftBrace
        this_variable_defined2 |
    tIf tLeftPar Expr tRightPar Statement
        tElse tLeftBrace this_variable_defined2 |
    tWhile tLeftPar Expr tRightPar tLeftBrace
        this_variable_defined2 |
    tVar _space this_variable_defined3 |
    tIf tLeftPar Expr tRightPar
        tVar _space this_variable_defined4 |
    tIf tLeftPar Expr tRightPar Statement
        tElse tVar _space this_variable_defined4 |
    tWhile tLeftPar Expr tRightPar
        tVar _space this_variable_defined4;
this_variable_defined3 ->
    tId tComma this_variable_defined3 |
    C & anystringwithoutbraces
        this_variable_defined3_afterskip;
this_variable_defined3_afterskip ->
    Statement this_variable_defined3_afterskip |
    tLeftBrace this_variable_defined3_afterskip |
    anystringwithoutbraces;
this_variable_defined4 ->
    tId tComma this_variable_defined4 |
    C & ListOfIds;

these_variables_not_defined ->
    these_variables_not_defined tComma tId &
        ~this_variable_defined |
    safeendingstring tVar _space tId &
        ~this_variable_defined |
    tId | e;

has_main_function -> Functions _m _a _i _n
        tLeftPar tId tRightPar CompoundSt Functions;

returns_a_value ->
    tLeftBrace Statements
        returns_a_value tRightBrace |
    tIf tLeftPar Expr tRightPar returns_a_value
        tElse returns_a_value |
    ReturnSt;
```

The generated parser uses the Generalized LR algorithm [12]. The parsing table contains 743 states.

B Test suite

This appendix contains an extensive collection of programs in the model language, which attempts to cover all aspects of its syntax. These are the tests the author used to convince himself that the grammar is correct, and they are given here in hope to convince the reader of the same.

Each example has a file name, which ends with **-yes** if the program is well-formed, with **-no** otherwise. This name is given in a margin note that accompanies every listing. Every example was tested with the parser, and its computation time[1] and the actual result of the computation (acceptance or rejection) are included in the corresponding margin note.

The first group of tests covers the lexical component of the language *[I]*. Following is a well-formed program that makes use of all the symbols from the alphabet *[I.1]*, contains at least one lexeme *[I.2]* of each type, and plenty of whitespace *[I.3]*:

I-a0-yes
3.17 sec., Acc

```
main(arg)
{
```

[1] A Pentium 4/1.5 GHz system running Windows 2000 was used. The parser was built with Borland C++ 5.5.1 command-line compiler using the options –O2 –6.

```
    var x, x1az, x1;
    var qwertyuiop, asdfghjkl, z1x23c445vbn7m890;

    x=x1az*100/(1+x1)-x%arg;
    if(x1>x1az & !(x<x1az | x>x1az | x==x1az))
        while(2+2!=4)
            x=-x;
    else
        x=0;

    qwertyuiop=asdfghjkl=z1x23c445vbn7m890=x;
    return x;
}
```

Let us now remove all nonessential whitespace from this program:

I-a1-yes
2.51 sec., Acc

```
main(arg){var x,x1az,x1;var qwertyuiop,asdfghjkl
,z1x23c445vbn7m890;x=x1az*100/(1+x1)-x%arg;if(x1
>x1az&!(x<x1az|x>x1az|x==x1az))while(2+2!=4)x=-x
;else x=0;qwertyuiop=asdfghjkl=z1x23c445vbn7m890
=x;return x;}
```

Only four essential spaces remain. The removal of any of them renders the program ill-formed:

I-a2-no
1.11 sec., Rej

```
main(arg){varx,x1az,x1;var qwertyuiop,asdfghjkl,
z1x23c445vbn7m890;x=x1az*100/(1+x1)-x%arg;if(x1>
x1az&!(x<x1az|x>x1az|x==x1az))while(2+2!=4)x=-x;
else x=0;qwertyuiop=asdfghjkl=z1x23c445vbn7m890=
x;return x;}
```

The next shorter example is a well-formed program:

I-b0-yes
0.43 sec., Acc

```
main(arg)
{
    var x, elsex;
    if(arg!=1) 1; else x=1;
    return 1;
}
```

Now let us remove the space between `var` and `x`, changing this pair of a keyword *[I.2.1]* and of an identifier *[I.2.2]* to a single identifier. The resulting program thus contains an incorrect statement formed by an identifier `varx`, a comma *[I.2.9]*, an identifier and a semicolon:

I-b1-no
0.31 sec., Rej

```
main(arg)
{
    varx, elsex;
    if(arg!=1) 1; else x=1;
    return 1;
}
```

The removal of a space between the keyword `return` *[I.2.1]* and the number `1` *[I.2.3]* turns these two lexemes into a single identifier `return1` *[I.2.2]*, which forms an atomic expression *[II.1.1]*, valid in itself. However, as there is no variable with this name, this violates one of the syntactical rules *[III.5]*, and also the function no longer returns a value *[III.6]*:

I-b2-no
0.42 sec., Rej

```
main(arg)
{
    var x, elsex;
    if(arg!=1) 1; else x=1;
    return1;
}
```

Let us try to spoil [I-b0-yes] in the same way by removing the space between the keyword `else` *[I.2.1]* and the identifier `else`, turning them into a single identifier `elsex`. Surprisingly, this time the program remains well-formed, as the `if` statement is still correct after losing its else clause *[II.2.4]*, while `elsex=1;` is a valid expression-statement *[II.2.1]*, because the variable `elsex` happens to be declared *[III.3.1]* before.

I-b3-yes
0.42 sec., Acc

```
main(arg)
{
    var x, elsex;
    if(arg!=1) 1; elsex=1;
    return 1;
}
```

Splitting lexemes in [I-b0-yes] by whitespace characters makes the program ill-formed:

I-b4-no
0.42 sec., Rej

```
main(arg)
{
    var x, elsex;
    if(arg! =1) 1; else x=1;
    return 1;
}
```

I-b5-no
0.43 sec., Rej

```
main(arg)
{
    var x, elsex;
    if(arg!=1) 1; else x=1;
    ret urn 1;
}
```

Keywords *[I.2.1]* do not work as identifiers *[I.2.2]*:

I-b6-no
0.42 sec., Rej

```
main(arg)
{
    var x, while;
    if(arg!=1) 1; else x=1;
    return 1;
}
```

Identifiers cannot start from digits:

I-b7-no
0.42 sec., Rej

```
main(arg)
{
    var x, 7elsex;
    if(arg!=1) 1; else x=1;
    return 1;
}
```

The next group of tests covers all types of syntactical constructs *[II]* defined in the language. Let us start from the syntax of the expressions *[II.1]*:

II.1-a0-yes
0.57 sec., Acc

```
f(x, y)
{
    return x+y;
}

main(arg)
{
    var x;
    x=f(arg+1, x=1)*-x/(x+arg);
    return x;
}
```

Its corrupted version with one parenthesis removed *[II.1.3]* is ill-formed:

II.1-a1-no
0.55 sec., Rej

```
f(x, y)
{
    return x+y;
}

main(arg)
{
    var x;
    x=f(arg+1, x=1)*-x/x+arg);
    return x;
}
```

Another corrupted version of [II.1-a0-yes] tries to use the symbol `*` as a prefix unary operator *[II.1.6]*, which it isn't *[I.2.5]*.

II.1-a2-no
0.56 sec., Rej

```
f(x, y)
{
    return x+y;
}

main(arg)
{
    var x;
    x=f(arg+1, x=1)-*x/(x+arg);
    return x;
}
```

The next corrupted expression attempts a postfix use of the minus:

II.1-a3-no
0.43 sec., Rej

```
f(x, y)
{
    return x+y-;
}

main(arg)
{
    var x;
    x=f(arg+1, x=1)*-x/(x+arg);
    return x;
}
```

The following modification violates the format of function calls *[II.1.4]*:

II.1-a4-no
0.56 sec., Rej

```
f(x, y)
{
    return x+y;
}

main(arg)
{
    var x;
    x=f(arg+1 x=1)*-x/(x+arg);
    return x;
}
```

Assignment expression syntax *[II.1.7]* is not observed here:

II.1-a5-no
0.57 sec., Rej

```
f(x, y)
{
    return x+y;
}

main(arg)
{
    var x;
    x=f(arg+1, x*2=1)*-x/(x+arg);
    return x;
}
```

Now let us turn to the syntax of statements *[II.2]*. This well-formed program contains all types of statements defined in the language:

II.2-a0-yes
0.73 sec., Acc

```
main(arg)
{
    var x;
```

```
    if(x>x)
        while(x==x)
            x=x;
    else
    {
        if(x==arg)
            return x;
        { }
        x;
    }
    return x;
}
```

Consider a short correct program, and remove a semicolon that is a part of an expression-statement *[II.2.1]*:

II.2-b0-yes
0.20 sec., Acc

```
main(arg)
{
    arg=arg;
    return arg;
}
```

II.2-b1-no
0.18 sec., Rej

```
main(arg)
{
    arg=arg
    return arg;
}
```

Use a semicolon outside of a statement (there are no C-like empty statements in the language):

II.2-b2-no
0.15 sec., Rej

```
main(arg)
{
    ;
    return arg;
}
```

Use a compound statement *[II.2.2]*, and then forget one of its delimiters:

II.2-b3-yes
0.21 sec., Acc

```
main(arg)
{
    { arg=arg; }
    return arg;
}
```

II.2-b4-no
0.20 sec., Rej

```
main(arg)
{
    { arg=arg;
    return arg;
}
```

Add a declaration *[II.2.3]*, and then violate its syntax:

II.2-b5-yes
0.28 sec., Acc

```
main(arg)
{
    var x;
    arg=arg;
    return arg;
}
```

II.2-b6-no
0.25 sec., Rej

```
main(arg)
{
    var ;
    arg=arg;
    return arg;
}
```

Consider another small well-formed program:

II.2-c0-yes
0.25 sec., Acc

```
main(arg)
{
    if(1)
        arg=arg;

    return 1;
}
```

Supply an else clause to the conditional statement *[II.2.4]*:

II.2-c1-yes
0.36 sec., Acc

```
main(arg)
{
    if(1)
        arg=arg;
    else
        arg=-arg;

    return 1;
}
```

Use two else clauses, which is not allowed *[II.2.4]*:

II.2-c2-no
0.45 sec., Rej

```
main(arg)
{
    if(1)
        arg=arg;
    else
        arg=-arg;
    else
        arg=arg+arg;

    return 1;
}
```

Use an else clause outside a conditional statement *[II.2.4]*:

II.2-c3-no
0.23 sec., Rej

```
main(arg)
{
    else
        arg=arg;

    return 1;
}
```

Omit the parentheses:

II.2-c4-no
0.21 sec., Rej

```
main(arg)
{
    if 1
        arg=arg;

    return 1;
}
```

Try an iteration statement *[II.2.5]* instead of a conditional, and supply an unexpected else clause *[II.2.4]*:

II.2-c5-yes
0.26 sec., Acc

```
main(arg)
{
    while(1)
        arg=arg;

    return 1;
}
```

II.2-c6-no
0.31 sec., Rej

```
main(arg)
{
    while(1)
        arg=arg;
    else
        1;

    return 1;
}
```

To conclude the list of possible errors in the syntax of statements *[II.2]*, consider a valid return statement *[II.2.6]*, a return statement with forgotten semicolon, and a return statement without a value.

II.2-d0-yes
0.14 sec., Acc

```
main(arg)
{
    return 2+2;
}
```

II.2-d1-no
0.14 sec., Rej

```
main(arg)
{
    return 2+2
}
```

II.2-d2-no
0.11 sec., Rej

```
main(arg)
{
    return;
}
```

The next two small programs show valid function declarations *[II.3.2]*:

II.3-a0-yes
0.25 sec., Acc

```
f()
{
    return 1;
}

main(arg)
{
    return f();
}
```

II.3-a1-yes
0.29 sec., Acc

```
f(x, y)
{
    return 1;
}

main(arg)
{
    return f(1, 1);
}
```

Using differently ill-formed function headers *[II.3.1]* leads to incorrect programs:

II.3-a2-no
0.21 sec., Rej

```
f(x,)
{
    return 1;
}

main(arg)
{
    return 1;
}
```

II.3-a3-no
0.21 sec., Rej

```
f(x y)
{
    return 1;
}

main(arg)
{
    return 1;
}
```

II.3-a4-no
0.18 sec., Rej

```
(x)
{
    return 1;
}

main(arg)
{
    return 1;
}
```

II.3-a5-no
0.20 sec., Rej

```
f(,)
{
    return 1;
}

main(arg)
{
    return 1;
}
```

Also, if a valid function header is followed by anything but a compound statement, the program becomes ill-formed:

II.3-a6-no
0.18 sec., Rej

```
f()
    return 1;

main(arg)
{
    return 1;
}
```

II.3-a7-no
0.15 sec., Rej

```
main(arg)
{
    return 1;
}

f()
```

Possible violations of the definition of a program *[II.4]* as a sequence of functions could be illustrated by the following single statement outside a function:

II-4-a0-no
0.04 sec., Rej

```
return 1;
```

Let us now proceed to the most interesting part of the tests – those covering the additional syntactical constraints *[III]*, most of which cannot be described using context-free grammars.

This well-formed program contains declarations of three functions with different names:

III.1-a0-yes
0.46 sec., Acc

```
function()
{
    return 1;
}

main(arg)
{
    return 1;
}

fonction(x, y)
{
    return 1;
}
```

If one symbol is changed in the French name of the third function, making it identical to the name of the first function, the unfortunate fruit of this anglicization is a program that violates the rule of the uniqueness of function declaration *[III.1.1]*:

III.1-a1-no
0.47 sec., Rej

```
function()
{
    return 1;
}

main(arg)
{
    return 1;
}

function(x, y)
{
    return 1;
}
```

Making the headers of these two functions identical is not of much help:

III.1-a2-no
0.45 sec., Rej

```
function()
{
    return 1;
}

main(arg)
{
    return 1;
}

function()
{
    return 1;
}
```

Now consider the simplest possible program:

III.1-b0-yes
0.14 sec., Acc

```
main(arg)
{
    return 1;
}
```

Changing the number of arguments of the function `main` makes the program ill-formed *[III.1.2]*:

III.1-b1-no
0.19 sec., Rej

```
main(first, second)
{
    return 1;
}
```

and so does renaming the main function *[III.1.2]*:

III.1-b2-no
0.17 sec., Rej

```
principale(arg)
{
    return 1;
}
```

The last way to violate this rule *[III.1.2]* is to consider a program without any function declarations *[II.4]*, i.e., a possibly empty sequence of whitespace characters:

III.1-c0-no
0.00 sec., Rej

```

```

Let us now make some ill-formed function calls *[III.2]*. First, a well-formed base program:

III.2-a0-yes
0.78 sec., Acc

```
argumentless()
{
    return 1;
}

function(x, y, z)
{
    return x*y*z;
}

main(x)
{
    return argumentless()+function(x-1, x, x+1);
}
```

Changing one letter in the name of the function being called yields a syntactically incorrect program, because now a call to an undeclared function `fonction` is being attempted *[III.2.1]*:

III.2-a1-no
0.76 sec., Rej

```
argumentless()
{
    return 1;
}

function(x, y, z)
{
    return x*y*z;
}

main(x)
{
    return argumentless()+fonction(x-1, x, x+1);
}
```

Let the function `function` now again be called by its proper name, but move its declaration after its use. The resulting program is ill-formed, because declarations should precede calls *[III.2.1]*:

III.2-a2-no
0.75 sec., Rej

```
argumentless()
{
    return 1;
}

main(x)
{
    return argumentless()+function(x-1, x, x+1);
}

function(x, y, z)
{
    return x*y*z;
}
```

Calling properly declared functions with an incorrect number of arguments is also not allowed *[III.2.2]*,

III.2-a3-no
0.78 sec., Rej

```
argumentless()
{
    return 1;
}

function(x, y, z)
{
    return x*y*z;
}

main(x)
{
    return argumentless(x)+function(x-1, x, x+1);
}
```

whether more or less arguments than needed are supplied:

III.2-a4-no
0.75 sec., Rej

```
argumentless()
{
    return 1;
}

function(x, y, z)
{
    return x*y*z;
}

main(x)
{
    return argumentless()+function(x-1, x+1);
}
```

A function can call itself, because in this case its header precedes the reference to it in its body *[III.2.1]*:

III.2-a5-yes
1.04 sec., Acc

```
argumentless()
{
    return argumentless()+1;
}

function(x, y, z)
{
    return function(x*y*z, 1, argumentless());
}

main(x)
{
    return argumentless()+function(x-1, x, x+1);
}
```

The variable scoping rules *[III.3]* can be violated in two ways: by defining two variables with the same name and with overlapping scopes *[III.4]*, and by using variables without a prior declaration *[III.5]*.

The first example is a correct program. There are two variables with the same name `fifth`, but their scopes are disjoint:

III.4-a0-yes
1.70 sec., Acc

```
f(first, second, third)
{
    var fourth;

    if(first==second)
    {
        var fifth;
        fifth=second;
        fourth=fifth;
    }
    else
        return third;

    var fifth;
    fifth=fourth;

    return fifth;
}

main(arg)
{
    return f(arg-1, arg, arg+1);
}
```

Let us define another variable named `fourth` inside the `if` statement. The scope of this variable overlaps with the scope of the outer `fourth`, and hence the program becomes ill-formed *[III.4]*:

III.4-a1-no
1.73 sec., Rej

```
f(first, second, third)
{
    var fourth;

    if(first==second)
    {
        var fourth, fifth;
        fifth=second;
        fourth=fifth;
    }
    else
        return third;

    var fifth;
    fifth=fourth;

    return fifth;
}

main(arg)
{
    return f(arg-1, arg, arg+1);
}
```

In the next example a variable `second` is defined in a declaration statement *[III.3.4]*, which is itself in the scope of the formal argument `second` *[III.3.3]*. Thus the program is ill-formed *[III.4]*:

III.4-a2-no
1.73 sec., Rej

```
f(first, second, third)
{
    var fourth, second;

    if(first==second)
    {
        var fifth;
        fifth=second;
        fourth=fifth;
    }
    else
        return third;

    var fifth;
    fifth=fourth;

    return fifth;
}

main(arg)
{
    return f(arg-1, arg, arg+1);
}
```

The following example declares two more variables called `fifth` *[III.3.5]*, bringing the total count of these variables to 4. However, their scopes are pairwise disjoint, and hence the program remains correct:

III.4-a3-yes
2.12 sec., Acc

```
f(first, second, third)
{
    var fourth;

    if(first==second)
    {
        var fifth;
        fifth=second;
        fourth=fifth;
    }
    else
        return third;

    if(1)
        var fifth;
    else
        while(1)
            var fifth;

    var fifth;
    fifth=fourth;

    return fifth;
}

main(arg)
{
    return f(arg-1, arg, arg+1);
}
```

Let us now leave only one variable with the name `fifth`. The program becomes incorrect, because the identifier `fifth` in the return statement is now outside the scope of the only remaining variable with this name *[III.5]*:

III.4-a4-no
1.38 sec., Rej

```
f(first, second, third)
{
    var fourth;

    if(first==second)
    {
        var fifth;
        fifth=second;
        fourth=fifth;
    }
    else
        return third;

    return fifth;
}

main(arg)
{
    return f(arg-1, arg, arg+1);
}
```

Following are some simpler examples of multiple variable declarations *[III.4]*:

III.4-b0-yes
0.31 sec., Acc

```
main(arg)
{
    var longidentifier, x;
    return 1;
}
```

III.4-b1-no
0.32 sec., Rej

```
main(arg)
{
    var longidentifier, arg;
    return 1;
}
```

III.4-b2-no
0.43 sec., Rej

```
main(arg)
{
    var longidentifier, x, longidentifier;
    return 1;
}
```

III.4-c0-yes
0.40 sec., Acc

```
main(arg)
{
    if(1)
        var identifier;
    var identifier;
    return 1;
}
```

III.4-c1-no
0.36 sec., Rej

```
main(arg)
{
    if(1)
        var identifier, identifier;
    return 1;
}
```

III.4-c2-no
0.43 sec., Rej

```
main(arg)
{
    if(1)
        var identifier, arg;
    var identifier;
    return 1;
}
```

III.4-c3-no
0.45 sec., Rej

```
main(arg)
{
    var identifier;
    if(1)
        var identifier;
    return 1;
}
```

III.4-d0-yes
0.28 sec., Acc

```
f(x, y, z)
{
    return 1;
}

main(arg)
{
    return 1;
}
```

III.4-d1-no
0.25 sec., Rej

```
f(x, y, x)
{
    return 1;
}

main(arg)
{
    return 1;
}
```

The next group of examples refers to the rule of declaration of variables before their use *[III.5]*. Here is how a simple change of identifiers turns a correct program into ill-formed:

III.5-a0-yes
0.55 sec., Acc

```
main(argument)
{
    var anothervariable;

    return argument+anothervariable;
}
```

III.5-a1-no
0.56 sec., Rej

```
main(argument)
{
    var anothervariable;

    return argumenth+anothervariable;
}
```

III.5-a2-no
0.56 sec., Rej

```
main(argument)
{
    var anothervariable;

    return argument+uneautrevariable;
}
```

Here is a slightly more complicated case involving scoping rules. The first program is correct:

III.5-b0-yes
0.64 sec., Acc

```
main(argument)
{
    var first;

    if(1)
    {
        var second;
        second=first;
    }

    return first;
}
```

If the expression in the return statement is changed to `second`, the program ceases to be correct, because the scope of this variable is confined to the compound statement inside the conditional statement *[III.3.4]*:

III.5-b1-no
0.64 sec., Rej

```
main(argument)
{
    var first;

    if(1)
    {
        var second;
        second=first;
    }

    return second;
}
```

Swapping the two statements inside the inner compound statement introduces an error, because the scope of the variable `second` *[III.3.4]* will begin after its use, and hence the use is ill-formed *[III.5]*:

III.5-b2-no
0.61 sec., Rej

```
main(argument)
{
    var first;

    if(1)
    {
        second=first;
        var second;
    }

    return first;
}
```

One more special case to test is when the name of a variable coincides with the name of some function. According to the definition of the language, this coincidence is irrelevant, and this is what the grammar demonstrates. In the first well-formed example the name `identifier` is shared by a function and a variable without any side effects:

III.5-c0-yes
0.63 sec., Acc

```
identifier() { return 1; }

main(arg)
{
    var identifier;
    identifier();
    return identifier;
}
```

Modifying a return statement to refer to the function instead of the variable does not spoil the program:

III.5-c1-yes
0.64 sec., Acc

```
identifier() { return 1; }

main(arg)
{
    var identifier;
    identifier();
    return identifier();
}
```

However, an earlier function declaration and a function call together will not atone an undefined variable!

III.5-c2-no
0.43 sec., Rej

```
identifier() { return 1; }

main(arg)
{
    identifier();
    return identifier;
}
```

The last requirement is that every function should return a value *[III.6]*. The following code manifestly violates this rule:

III.6-a0-no
0.09 sec., Rej

```
main(x)
{
    x=1;
}
```

Here is a well-formed program that can be spoiled in various ways:

III.6-b0-yes
0.56 sec., Acc

```
main(x)
{
    x=x;

    if(2+2<4)
        return 1;
    else if(2+2>4)
    {
        x+x*x;
        return 2;
    }
    else
        return 0;
}
```

If the `return` statement *[II.2.6]* in any of the three branches of the nested conditional statements *[II.2.4]* is replaced by anything but a properly returning statement *[III.6.1]*, the program becomes ill-formed *[III.6.2]*:

III.6-b1-no
0.56 sec., Rej

```
main(x)
{
    x=x;

    if(2+2<4)
        x=4-(2+2);
    else if(2+2>4)
    {
        x+x*x;
        return 2;
    }
    else
        return 0;
}
```

III.6-b2-no
0.65 sec., Rej

```
main(x)
{
    x=x;

    if(2+2<4)
        while(2+2==4)
            return 1;
    else if(2+2>4)
    {
        x+x*x;
        return 2;
    }
    else
        return 0;
}
```

If the `return` statement and the expression-statement *[II.2.1]* in the properly returning compound statement *[III.6.1]* in the second branch of the code [III.6-b0-yes] are swapped, then this compound statement ceases to be properly returning, and the program is again incorrect:

III.6-b3-no
0.56 sec., Rej

```
main(x)
{
    x=x;

    if(2+2<4)
        return 1;
    else if(2+2>4)
    {
        return 2;
        x+x*x;
    }
    else
        return 0;
}
```

Taking a simpler well-formed program

III.6-c0-yes
0.26 sec., Acc

```
main(x)
{
```

```
    if(1)
        return 1;
    else
        return 0;
}
```

if the conditional statement loses its else clause *[II.2.4]*, the body of the function `main` is no longer properly returning *[III.6]*:

III.6-c1-no
0.15 sec., Rej

```
main(x)
{
    if(1)
        return 1;
}
```

The same happens if the conditional statement is replaced with an iteration statement *[II.2.5]*:

III.6-c2-no
0.17 sec., Rej

```
main(x)
{
    while(1)
        return 1;
}
```

Let us conclude with a reasonable example of a program in this language (reasonable to the extent a program in a language without semantics could be considered reasonable!).

factorial
0.65 sec., Acc

```
factorial(n)
{
    if(n>1)
        return n*factorial(n-1);
    else
        return 1;
}

main(arg)
{
    return factorial(arg);
}
```

If the semantics of the language were defined in the natural way, this program could be said to compute the factorial of a given number.

C Performance

In order to get some impression of the parsing speed as a function of the length of the input, consider a family of programs, each comprised of many simple functions calling each other. The family is parametrized by the number of functions in a program. Here is the program number ten:

test10
3.14 sec., Acc

```
f00bv(a)
{
    return 1;
}

f01ke(i)
{
    return f00bv(i)*f00bv(i);
}

f02dn(sj)
{
    return f01ke(sj)*f01ke(sj);
}

f03mc(e)
{
    return f02dn(e)*f01ke(e);
}

f04rd(s3)
{
    return f03mc(s3)*f02dn(s3);
}

f05yw(i)
{
    return f04rd(i)*f02dn(i);
}

f06nn(a)
{
    return f05yw(a)*f03mc(a);
}

f07qi(uq)
{
    return f06nn(uq)*f03mc(uq);
}

f08nn(q)
{
    return f07qi(q)*f04rd(q);
}

f09ki(o8)
{
    return f08nn(o8)*f04rd(o8);
}

main(arg)
{
    return f09ki(arg);
}
```

The results are given in Table 1 and plotted in Figure 3; they leave an impression of square time performance.

name	*number of symbols*	*time (sec.)*	*name*	*number of symbols*	*time (sec.)*
test01	59	0.33	**test60**	2550	32.81
test03	144	0.88	**test70**	2972	41.59
test05	229	1.48	**test80**	3394	51.33
test10	440	3.14	**test90**	3816	62.11
test15	651	5.05	**test100**	4238	73.98
test20	862	7.23	**test115**	4901	94.43
test25	1073	9.62	**test130**	5564	117.58
test30	1284	12.25	**test150**	6448	153.22
test40	1706	18.17	**test175**	7553	204.70
test50	2128	25.03	**test200**	8658	265.19

Table 1: Results of performance tests.

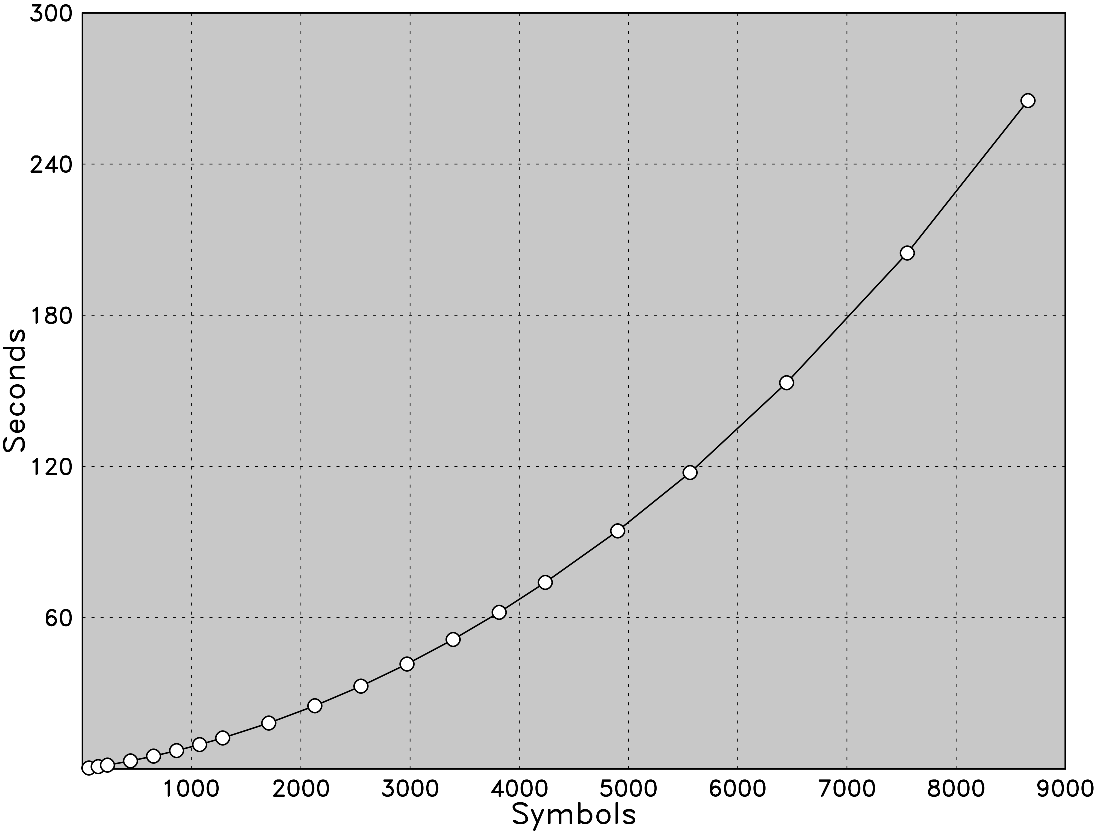

Figure 3: Plotted results of performance tests.