token: K1
  T1 <- K1.
  ^self
]

Class Skip Simple_instruction

Class Program Object L1
  Methods Program
    new
      ^List new
    |
      Instruction: K1
      L1 addLast: K1.
      ^self
]

Class Instruction Object Label Simple_instruction
  Methods Instruction
    new
      ^self
    |
      Label: K1
      Label <- K1.
      ^self
    |
      Simple_instruction: K2
      Simple_instruction <- K2.
      ^self
]
class Program {
    public:
        PtrList<Instruction>& V1;
        Program(PtrList<Instruction>& V1);
    }

class Instruction {
    public:
        Label* V1;
        Simple_instruction* V2;
        Instruction(Label* V1, Simple_instruction* V2);
    }

A.3  A Trivial Program

    A: goto X;
    skip;
    C: goto U

A.4 Generated Instantiation in C++

Program(
    PtrList<Instruction>(
        Instruction(Label("A"), Goto(Label("X"))),
        PtrList<Instruction>(
            Instruction(Label()), Skip()),
        PtrList<Instruction>(
            Instruction(Label("C"), Goto(Label("U"))),
            NULL)))

A.5 Generated Class definition in SMALLTALK

Class Goto Simple_instruction Label
Methods Goto
    new
        ^self
        |
        Label: K1
        Label <- K1.
        ^self
    ]

Class Simple_instruction Object

Class Label Object T1
Methods Label
    new
        ^self
    |
A A Trivial Language Example

Section A.1 is a simple but complete example of a parsing-grammar. The language used is the same trivial language employed in [16, Page xviii] for exposition purposes. Section A.2 lists the automatically generated class definition in C++ for this grammar. A generated instantiation for the trivial program in Section A.3 is shown in Section A.4. Finally Section A.5 lists an automatically generated class definition in SMALLTALK, generated by SOOP when the C++ grammar in Figure 7 (page 18) is replaced with a synthesizing-grammar for producing SMALLTALK code.

A.1 A Parsing Grammar $G_S$

Program = {Instruction ";" ...}
Instruction = [Label ":" ] Simple_instruction
Simple_instruction = Skip | Goto
  Skip = skip
  Goto = goto Label
  Label = <many(&letters)>

A.2 Generated Class Definition in C++

```cpp
class Goto : public Simple_instruction {
public:
    Label* V1;
    Goto(Label* V1);
};

class Simple_instruction {
public:
};

class Label {
public:
    token* V1;
    Label(token* V1);
};

class Skip : public Simple_instruction {
public:
};
```


25
References


4.4 Other Adjustments

Some minor difficulties encountered are listed below:

- Not all the keywords of EIFFEL are found in grammar, e.g., div and mod.
- The usage of double and single quotes is inconsistent, e.g., “+” and ‘+’.
- Binary derivations are not quoted: (SOOP was confused by the ‘=’...)
  \[ Binary = \wedge | = | / = | < | > | <= | >= \]
- The derivation of some constructs is a single keyword, e.g.,
  \[ Nochange = nochange \]
- Binary expression and expression are mixed.

5 Conclusions and Further Research

This paper presented SOOP, a versatile tool for the use of builders of source code tools. A tool builder can configure SOOP to work with a programming language of choice in writing the tool and for handling source code written in diverse programming languages. SOOP automatically deduces a class hierarchy of the syntactical constructs of an input programming language. The parse information of a program is represented as a natural ‘has-a’ hierarchy which captures semantic structure and ignores syntactical details. By using the notion of streams, we were able to extend the BNF grammar to also handle production of output.

SOOP was designed to separate the parser, the parsing information, and the tool which may use this information. Implementation was in the ICON programming language, which enabled a very rapid prototyping (less than 6×person×month). An interesting recurring theme in the working of SOOP is the transformation between representation of information as data files and as program text. Using SOOP to represent EIFFEL was a rewarding experiment for us.

Future work includes the actual building of tools using SOOP. In particular, we hope to be able to implement SOOP under itself. Possible directions for the development of SOOP are: expansion to attributed context free grammars, specifications of streams in the parsing grammar, and perhaps even inclusion of symbol table data in the information that SOOP provides.

Acknowledgment We thank Ricardo Szmit for his helpful comments on an early versions of this paper.
Expression* V1;
Multiary* V2;
Multiary_expression* V3;
Multiary_separated_list(Expression* V1, Multiary* V2, Multiary_expression* V3);
...
}

Note that Multiary_separated_list is a Multiary_expression, and thus allowed in place of V3. The expression is associated from left to right regardless of the Multiary operator. The solution is too lengthy to be clearly presented here.

- Surprisingly, some of the classes generated for Eiffel have multiple inheritance. The precise classes may be spotted by filtering the generated file through

  grep ", public"

We get the following classes headers:

class Current : public Anchor, public Unqualified_expression {
class Unqualified_call : public Call, public Unqualified_expression {
class Constant : public Feature_value, public Unqualified_expression {
class Minus : public Sign, public Unary, public Multiary {
class Expression : public Expression_list, public Multiary_expression {
class Plus : public Sign, public Unary, public Multiary {

The reason for this behavior is that the same construct is derived in an inheritance rule from several different constructs. Let's look into the case of class Constant. By filtering the Eiffel parsing grammar specification through 'grep Constant' we get:

    Feature_value = Constant|Routine
    Constant = Integer_constant|Character_constant|Boolean_constant|
          Real_constant|String_constant
Unqualified_expression = Constant|Entity|Unqualified_call|Current|Old_value|
          Nochange|Operator_expression

Clearly, Constant is not both a Feature_value and a Unqualified_expression. At worst, it is either that or the other. This means that what we really have here is exclusive inheritance. However, conceptually Constant is not limited to being either Feature_value or Unqualified_expression, (although that is its only usage so far). A more correct solution seems to be inclusion. One of the possibilities for Feature_value is that Feature_value has a Constant, and one of the possibilities for Unqualified_expression is that Unqualified_expression has a Constant. This is exactly the buy-or-inherit question for rules of the form \( A \rightarrow A' \).
types are entirely uppercase, were assumed mandatory. Thus simple strings are all that was added.

- It is quite possible that a grammar specification would be erroneous. In the EIFFEL grammar, as a matter of fact, the construct \textit{Local\_variables} is defines as

\[
\text{Local\_variables} = \text{Entity\_declaration\_list}
\]

The keyword \textit{local} is missing in front of \textit{Entity\_declaration\_list}. This was quickly revealed when the first EIFFEL program was attempted through the SOOP generated parser.

- Left recursion might cause a recursive-descent parser to loop endlessly and therefore must be eliminated from the grammar. The definition of \textit{Expression} in the EIFFEL grammar is indeed such a case:

\begin{verbatim}
Expression    =     [\{Unqualified\_expression “.”\}]
Unqualified\_expression    =     Constant\|Entity\|Unqualified\_call\|Current\|Old\_value\|Nochange \|Operator\_expression
Binary\_expression    =     Expression\|Binary\_expression
Operator\_expression    =     Unary\_expression\|Binary\_expression
\end{verbatim}

\subsection*{4.3 Semantic Adjustments}

Completing the adjustments up to this point and assuming the predefined class-synthesizing grammar (Figure 7) is used, SOOP readily produces a class definition in C++. The class hierarchy reflects the different constructs of the EIFFEL grammar and their mutual inheritance-composition relations. Some of the grammar properties become clearer with this representation. The grammar may be fixed as desired and SOOP re-run to produce an improved class definition. We address two properties that need correction: operator precedence and multiple inheritance.

- Operator precedence is listed separately from the EIFFEL grammar. The current version of SOOP does not support precedence other than that applied by the grammar. This can be spotted by looking at the automatically generated class \texttt{Multiary\_separated\_list}. (Recall the syntactical adjustment Sect. 4.1, Page 20).

\begin{verbatim}
//
//  Multiary\_separated\_list = Expression Multiary\_expression
//  Multiary\_expression = Expression\|Multiary\_separated\_list
//
class Multiary\_separated\_list : public Multiary\_expression {
public:
\end{verbatim}
where $Separator = \{\text{";"}, \text{";"}\}$. SOOP notational simplifications support only a single separator. The few such cases need to be converted to a set of rules that derive the list explicitly (Sect. 3.2, Page 15). For the above example, the set of rules would be:

- $Expression\_list = Expression | Expression\_tail$
- $Expression\_tail = Expression Expression\_separator Expression\_list$
- $Expression\_separator = Comma | Semi\_Colon$
- $Comma = \{\text{","}\}$
- $Semi\_Colon = \{\text{",";"}\}$

As a result, the expression separator becomes a meaningful terminal. In some cases, the separator is indeed meaningful, such as for $Multiary\_expression$:

- $Multiary\_expression = \{Expression Multiary\ldots\}^+$
- $Multiary = \{\text{"+"|\text{"-"|\text{"*"|\text{"/"|\text{"and|\text{"and|\text{"or|\text{"or else.\}

All that is actually required is to allow the separator to be a construct, not just a terminal.

### 4.2 Lexical and Grammar Adjustments

There is not a separate phase of lexical analysis in SOOP. Tokens are expected to express their own lexical structure explicitly. Since lexical information is not always listed as clearly as the grammar, some effort might be required here. For EIFFEL, the text needs to be searched in order to find it. The lexical analysis is attempted by the recursive-descent generated parser only when needed. Care must be taken that the grammar has no left-recursion within it. Otherwise, the recursive-descent parser might recurse endlessly.

- The EIFFEL grammar does not specify rules for deriving the predefined constructs $Identifier$, $Real$, $Integer$, and $String$. Thus, matching expressions need to be added for each one of them. For example (Sect. 3.2, Page 13),

$$Identifier = \langle\text{any(\&letter)} \& \text{many(\&letters \& \&digits \& \text{";"})}\rangle$$

Moreover, since these constructs are written in the same font as other constructs, it is difficult to pinpoint them. Failing to specify a rule for one of them is like missing out any other rule. For brevity and clarity of this demonstration, $Integer$ and $Real$ where defined simply as $\langle\text{many(\&digits)}\rangle$, and $String$ as $\langle\text{many(\&ascii)}\rangle$.

- Letter case is insignificant in EIFFEL. Predefined names, such as $\text{Current}$, should not therefore be expressed as simply $\text{current} = \{\text{"Current"}\}$. Deriving all the possible ways in which $\text{Current}$ can be spelled with interleaving lower- and uppercase letters, is not an encouraged solution either. Instead, a proper matching expression in ICON has to be written. Once again, this was not done for the sake of simplicity and clarity. The conventions that predefined names start with an uppercase letter, and that predefined
4.1 Syntactical Adjustments

In the design of SOOP an effort was made to keep the notational simplifications expressible in plain text format. In the EIFFEL grammar, however, special symbols and change of font play an important part in the description. While this makes the grammar readable to the human eye, it requires conversion to the notations of SOOP. Some examples are:

- In the EIFFEL grammar \{Construct \section{\ldots}\}^+ is used to describe a sequence of one or more instances of \textit{Construct} separated from each other, if more than one, by the separator \section{.}. For instance, \texttt{Then\_part\_list} = \{\texttt{Then\_part \texttt{elsef} \ldots}\}^+. SOOP, on the other hand, eliminates the need for \\
\texttt{“+”}:
  \[
  \texttt{Then\_part\_list} = \{\texttt{Then\_part “elsef” \ldots}\}
  \]
Consequently, a sequence of zero or more instances, as in \texttt{Compound} = \{\texttt{Instruction “;” \ldots}\}, is expressed by:
  \[
  \texttt{Compound} = [\{\texttt{Instruction “;” \ldots}\}]
  \]

- Predefined entities and routines appear in italic font and start with an uppercase letter. For example, \texttt{Current} and \texttt{Result}. Standard class names and other predefined types appear whole in uppercase. For example, \texttt{INTEGER} and \texttt{CHARACTER}. Capitalized words, however, are reserved by SOOP for denoting syntactical constructs. To fix this, the missing rules need to be added. For example, \texttt{Current} = \texttt{“Current”} and \texttt{INTEGER} = \texttt{“INTEGER”}

There is also a lexical difficulty involved here that is addressed in Section 4.2 below.

- In inheritance rules, where alternatives are separated by vertical bars, SOOP does not allow terminals to appear. For such cases, auxiliary constructs may be added to conform the rule to the syntactical restrictions. For example, the rule \texttt{Sign} = \texttt{‘+’|‘-‘} is written for SOOP as three rules:
  \[
  \texttt{Sign} = \texttt{Plus}\_\texttt{Minus} ; \texttt{Plus} = \texttt{“+”} ; \texttt{Minus} = \texttt{“-”}
  \]
  and the rule \texttt{Boolean\_constant} = \texttt{true|false} is written as the three rules:
  \[
  \texttt{Boolean\_constant} = \texttt{True}|\texttt{False} ; \texttt{True} = \texttt{true} ; \texttt{False} = \texttt{false}
  \]
SOOP could have been altered (perhaps in the future it will be) to accept such rules. However, the underlying difficulty is in deciding whether terminals are meaningful or not. It seems \texttt{‘+‘} is used by the EIFFEL grammar to denote the unary sign and \texttt{“+”} the binary operation. But is this distinction really necessary? The lexical sign is insignificant in a class hierarchy that reveals the operation type anyway.

- In separated lists of the form \{\texttt{Construct \section{\ldots}}\} in the EIFFEL grammar, the separator \section{.} may have several alternatives. For instance,
  \[
  \texttt{Expression\_list} = \{\texttt{Expression Separaror \ldots}\}
  \]
4 Digesting EIFFEL — A Worked Case

In this section we bring a complete example of how SOOP is put to work. In order to exhibit SOOP’s ability to handle different source and target languages, we show in details how SOOP is configured to process EIFFEL code and synthesize classes and objects in C++. The first step is to write a parsing-grammar for EIFFEL. The grammar [17, Appendix C.2] written by EIFFEL’s designer, Bertrand Meyer, was taken as a reference to the syntactical specification of EIFFEL. Although the EIFFEL language has been slightly modified since, this exercise serves well for the purpose of demonstrating SOOP’s capabilities and limitations.

The use of notational simplifications by SOOP (Sect. 3.1, Page 12), made it possible to key in the EIFFEL grammar almost verbatim out of the syntactical specification. Nevertheless, few adjustments were required along the process. The adjustments are categorized below as syntactical adjustments, lexical and grammar adjustments, semantic adjustments, and some other minor adjustments. For each such category, the precise problem is explained, the method taken and difficulties encountered are described, and examples of how rules appearing in the grammar are transformed to meet SOOP requirements are given.
Derivation fails if the designated stream does not exist, or is empty. For example, the
following prints a comma separated list of the elements in the stream vals

```plaintext
List = { Item "," ...}
Item = @vals
```

Each occurrence of a keyword in a rule designates an independent copy of a stream.
For example, the following prints an arithmetic expression for the sum of squares of a
list of elements in the stream vals

```plaintext
Sum = { Pair "*" ...}
Pair = @vals "*" @vals
```

Suppressed

A terminal enclosed in angular-brackets suppresses output of that terminal
(but the terminal is still processed, e.g., in the case of a stream generator enclosed
in angular-brackets, an element is still drawn out of the stream). For example, the
following prints as many '*' as there are elements in the stream the stream vals

```plaintext
Stars = { DrawValue "*" ...}
DrawValue = <@vals>
```

Assignments

An assignment to a keyword gives birth to a new stream whose name is that
of the keyword. The assignment symbol is ‘:=’. The assignment must be enclosed in
angular-brackets and produces no output. For example, derivation of the rule

```plaintext
Class = <c:=@class> <slot:=@c.slots>
```

creates two streams: c, which is an infinite stream of the class record selected out of
class, and slot, which is the stream of the slots field in the last generated record
of the stream class which is the infinite generated element of c. Stream names are
global and could be used anywhere. An assignment of a constant value to a keyword
makes that keyword designate an infinite stream of that constant value, e.g., as in c. It
is also possible to assign an Icon expression or even co-expression to a keyword. The
following will output the octal digits:

```plaintext
Octal = <oct:=(0 to 7)> PrintIt
PrintIt = @oct
```

A stream terminal may fail if the stream is empty, in which case its failure causes the
entire rule to fail. A stream terminal succeeds only if its activation generates a value suitable
to the context it appears in. That is, term would succeed only if it produces a string and
term.name would succeed only if term produces a record which has a field named name, and
that field is of type string.

For example, the grammar in Figure 7 is a synthesizing-grammar that synthesizes the
classes in C++ style.
The class generation routine executes by trying to “derive” the first Construct in the input file. Whenever it encounters a Construct it executes the Derivation associated with it. A Derivation, which is a sequence of items, is executed by executing, in order, the items in it. Backtracking is possible, so if the execution of a later item fails, the effects of execution of all previous items in the list are undone. When several alternatives are possible, as it is in optional items, non-empty lists, or a Derivation which is a list of alternate construct, they are tried in turn until the first one succeeds in the input file.

At this point the reader might wonder how a derivation of a rule can fail at all, and how different outputs can be produced by the same synthesizing grammar. The answer stems from the ability of synthesizing grammars to manipulate streams. Recall that the class generation routine receives as its input a sequence of records, where each record contains a description of a class to be generated. This sequence is generated as a stream, which by convention, is named `class`. The name `class` is assumed by the synthesizing-grammar to be predefined, and used for synthesizing the text code representing the classes. `class` is capable of generating triples \( \langle \text{name}, \text{bases}, \text{slots} \rangle \) where `name` is the class name. `bases` is a list of bases classes from which the class has derived, and `slots` is a list of slots types. The derivation process may, (in fact, should) relate to the `class`.

In order to be able to manipulate streams, we introduce the notion of pseudo-terminals which can take the form of `strings`, `streams`, `suppressed` terminals and `assignments`:

**Strings** Simple terminals can be written as strings enclosed in matching double quotes (or matching single quotes). Each time a rule is successfully applied, all such terminals in it print themselves. Escape sequences allow characters to be included in strings that otherwise would be impossible to include. An escape sequence consists of a backslash followed by one or more characters that are given special meaning. The escape sequences and the characters that they stand for are those defined in ICON, a partial list of which is:

\[
\begin{align*}
\text{	extbackslash f} & \quad \text{formfeed} \\
\text{	extbackslash l} & \quad \text{linefeed} \\
\text{	extbackslash n} & \quad \text{newline} \\
\text{\textbackslash t} & \quad \text{horizontal tab} \\
\text{\textbackslash d\textbackslash d\textbackslash d} & \quad \text{octal code} \\
\text{\textbackslash x\textbackslash d\textbackslash d} & \quad \text{hexadecimal code} \\
\text{\textasciicircum c} & \quad \text{control code}
\end{align*}
\]

For example, the following prints a horizontal tab after processing Head and before processing Tail, and starts a new line right afterwards:

`Example = Head \"\t\" Tail \"\n\"`

**Streams** A keyword preceded by an at sign (@) marks the generation of the next value out of the stream whose name is given by the identifier, and the printing of that value.
The if and the parenthesis in the above rule didn’t receive a slot in the class definition of `Conditional`. This is because these terminals, just as the if and then in an Eiffel conditional statement, are only required for accurate parsing of a program text. Once the text was parsed and a conditional statement was identified as such, all that matters is the structure of this statement: the boolean condition, the statement to execute when this expression is true, and the optional else part. Clearly, only the abstract structure of a program is of concern to a high-level software engineering tool.

To complete the discussion on how a parsing grammar might be translated into a class hierarchy, we need to consider the notation `{ `Construct separator `}` which designates a non-empty list of `Constructs`. The notation is not essential for an expressive power since a non-empty list could be specified without it:

```
Non_Empty_Argument_List = Identifier | Tail
Tail = Identifier\," Non_Empty_Argument_List
```

However, the class definitions which will arise from the above rules are bound to be very cumbersome, and will not capture the deep structure of the language. In contrast to that, the list notation simplifies writing grammars and encourages a style which reveals essential properties of the language. For example, the specification of a Pascal parametrized procedure call can be written as:

```
PPC = Identifier ( { Identifier \," } )
```

This rule translates into a C++ class definition outlined as follows:

```
class PPC {
    public:
        Identifier &slot1;
        PtrList<Identifier> &slot2;
        ...
};
```

### 3.3 The Synthesizing Grammars

The grammars $G_C$ and $G_I$ are called synthesizing grammars since they specify how Soop and the parser it generates should produce output. Except for several modifications, they are very similar in appearance to the parsing grammar with different interpretations given to production rules, as appropriate for producing output instead of parsing a program text. The ensuing description uses the grammar $G_C$ as the main example. The extrapolation to $G_I$ is straightforward.
1. defines that in $S$, the parsing of the $\text{Statement}$ is as either of $\text{Loop}$, $\text{Conditional}$ or $\text{Assignment}$, and

2. states that in the class hierarchy in $T$, the classes $\text{Loop}$, $\text{Condition}$ and $\text{Assignment}$ are inherited from the abstract class $\text{Statement}$.

A grammar which includes the above rule will translate into C++ class definition (for instance) outlined as follows:\textsuperscript{10}

\begin{verbatim}
class Statement { ...};
class Loop: public Statement { ... };
class Conditional: public Statement { ... };
class Assignment: public Statement { ... };
\end{verbatim}

The class associated with a structure symbol of a composition rule will include (at least) a slot for each of the constructs which occur in $\text{Derivation}$. For example, the rule
\begin{verbatim}
Procedure = Header Body
\end{verbatim}

is a composition rule which

1. defines that in $S$, the $\text{Procedure}$ construct is a $\text{Header}$ construct followed by a $\text{Body}$ construct, and

2. states that the class $\text{Procedure}$ in $T$ is an aggregate of two members of type $\text{Head}$ and $\text{Body}$.

The above rule will translate into a C++ class definition outlined as follows:

\begin{verbatim}
class Procedure ...{
    public:
        Header &slot1;
        Body &slot2;
        ...
};
\end{verbatim}

Note that the slots $\text{Head}$ and $\text{Body}$ in the above class are publicly accessible.

An optional item in the $\text{Derivation}$ pattern will translate into a slot which may or may not be occupied. For example, a derivation rule defining a C or C++ conditional statement
\begin{verbatim}
Conditional = if '(' Condition ')' Statement [ ElsePart ]
\end{verbatim}

will translate into a C++ class definition outlined as follows:

\begin{verbatim}
class Conditional ...{
    public:
        Condition &slot1;
\end{verbatim}

\textsuperscript{10}The exact translation is dependent on how $G_C$ was written.
regular expressions of lexical analysis program generators such as lex [14]. They are written as embedded ICON expressions, for example,

\[
\text{DecimalInteger} = \langle \text{many}^3( \&\text{digits}^4) \rangle
\]

A slightly more complex example is the definition of EIFFEL identifiers ("Identifiers are a sequence of characters, all of which must be letters, digits or underscore characters; the first character of an identifier must be a letter")

\[
\text{Identifier} = \langle \text{any}^5( \&\text{letters}^6) \rangle \& \text{many}( \&\text{letters}^6 \&\text{digits}^8 \&\text{'_'}^2 ) \rangle
\]

Terminals in the parsing grammar are either meaningful-terminals or meaningless-terminals. Terminals denoted in the grammar as keywords or as quoted strings are considered meaningless and are silently ignored immediately after parsing. The rationale is that since there is only one possible way of matching keyword and string-terminals against the input, they are merely parsing aids but otherwise carry no information. In contrast to that, a terminal written as an ICON expression enclosed in angular brackets has potentially many different matchings in the program text, and is therefore considered meaningful. If such a terminal occurs in the Derivation sequence of Construct, then the definition of the class corresponding to Construct will have a data slot for that terminal. The value inserted in this slot during instantiation of the class will be the string in the program text that was matched against the ICON expression of that terminal.

Each syntactical construct which occurs in GS, the parsing grammar of the language S, translates into a class definition in T. A program in S is then translated into an object of the class corresponding to the start symbol of the grammar GS. This object represents the top of a composition hierarchy of other objects instantiated from the other syntactical constructs of S. The parsing information is preserved in this composition hierarchy. A writer of a client tool will develop a code to traverse and process the composition hierarchy, exploiting the inheritance hierarchy as an aid in code development.

The class associated with a classification symbol of an inheritance rule will be an abstract one (i.e., no objects will be instantiated from it), and assigned as the superclass of each of the constructs in Derivation. For example, the rule

\[
\text{Statement} = \text{Loop} | \text{Conditional} | \text{Assignment}
\]

is an inheritance rule which

\[\text{many}(c)\) succeeds after scanning the longest initial sequence of characters in the character set \(c\) within the scanned string. It fails if the next character is not in \(c\).
\[\&\text{digits}\] is the character set which includes '0', '1', ..., '9'.
\[\text{any}(c)\] succeeds if the next scanned character is in the character set \(c\). It fails otherwise.
\[\&\text{letters}\] is a character set consisting of the 52 uppercase- and lowercase letters.
\[x1 \& x2\] evaluate \(x1\) and \(x2\) in conjunction, producing \(x2\).
\[x1 \&\& x2\] produces the union of the character sets \(x1\) and \(x2\).
\[\text{If they become necessary for some reason, then they can be readily derived from the hierarchy of classes and objects.}\]
form of the conditional instruction in Eiffel, where nesting conditional instructions in else parts is avoided by using one or more elsif branches

\[
\text{Then_part_list} = \{ \text{Then_part elsif} \ldots \}
\]

\textit{separator} can be either a string or a keyword terminal.

**Optional sequence of constructs** A sequence of zero or more instances are denoted by both notations simultaneously as in

\[
\text{ArgumentList} = [\{ \text{Argument ",", } \ldots \}]
\]

The selection of notational simplifications which SOOP recognizes was inspired by the convention used in [16]. Although the two are not identical, there were enough similarities to contribute to the smooth application of SOOP to the full Eiffel syntax.

The restriction in SOOP to two types of derivation rules was done in order to ensure coherent and meaningful translation composition and inheritance hierarchies. The restrictions adopted are in fact a variation of the \textit{restricted context-free grammars} as defined by Wu and Wang [27, 28, 29], which tried to achieve the same purpose. It is not difficult to see that these restrictions do not lead to any loss of generality. Any language definable by a context free grammar can be also defined by a restricted context free grammar.\(^2\)

Terminals in the input grammars differ in syntax and interpretation due to the different roles they play. They would therefore be described for each grammar type separately.

### 3.2 The Parsing Grammar

In specifying the parsing grammar, terminals can be written as \textit{keywords}, \textit{strings} or \textit{matching expressions}:

**Keywords** All identifiers beginning with a lower-case letter are considered keywords of the language and stand for themselves. The keywords in the following rule

\[
\text{Conditional} = \text{if Condition then Statement else Statement fi}
\]

are if, then, else and fi.

**Strings** Simple terminals which cannot be specified as keywords can be written as strings enclosed in matching double quotes (or matching single quotes), as in

\[
\text{Assignment} = \text{Identifier ":=" Expression ";"}
\]

**Matching-Expressions** Recall that no separate lexical analysis is assumed. Terminals formed by lexical scanning are specified by matching-expressions in ICON enclosed in angular-brackets. Matching-expressions should be thought of as the equivalent of the

\(^2\)To be convinced in that, recall that any grammar can be represented in Chomsky normal form.
3.1 Overall Description of the Input

Each of the input grammars is supplied in a separate file consisting of a list of the derivation rules for that grammar. A derivation rule has the general form

\[ \text{Construct} = \text{Derivation} \]

designating that \text{Construct} is defined by the \text{Derivation} pattern, i.e., \text{Derivation} specifies all possible derivations of \text{Construct}. The \text{Derivation} pattern represents all the grammar’s production rules whose left hand side is \text{Construct}.

\text{Construct} is a \textit{syntactical construct} (a construct for short). It is written as a capitalized identifier, e.g., \text{Procedure}, \text{If Statement}, or \text{Class Definition}. The construct defined by the first derivation rule in the file is the grammar start symbol. A construct cannot be defined by more than one derivation rule. All constructs which occur in a \text{Derivation} pattern must have a definition by a derivation rule.

\textsc{Soop} restricts its input grammars to those including only two types of derivation rules, \textit{inheritance} rules and \textit{composition} rules:

1. \textit{Inheritance} The \text{Derivation} of an inheritance rule is a list of alternative constructs separated by vertical bars, for example

\[ \text{Statement} = \text{Loop} \mid \text{Conditional} \mid \text{Assignment} \]

The \text{Construct} of an inheritance rule, e.g., \text{Statement}, is called a \textit{classification symbol}.

2. \textit{Composition} The \text{Derivation} of a composition rule is a sequence of items (syntactical constructs or other) separated by spaces,

\[ \text{Procedure} = \text{Header} \ \text{Body} \]

The \text{Construct} of a composition rule, e.g., \text{Procedure}, is called a \textit{structure symbol}.

The items allowed in the sequence are constructs, \textit{terminals} of the grammar which play different roles in the different grammars, and \textit{notational simplifications} which provide a concise expression of common derivation idioms.

The notational simplifications allow \textit{optional items}, \textit{sequences of constructs} and \textit{optional sequences of constructs}:

**Optional Item** An item enclosed by square brackets is optional. For example consider the definition of the \textsc{Eiffel} loop instruction:

\[ \text{Loop} = \text{Initialization} \ [\text{Loop invariant}] \ [\text{Loop variant}] \ \text{Exit clause} \ \text{Loop body} \ \text{end} \]

**Sequence of constructs** The notation \{ \text{Construct} \ \text{separator} \ldots \} designates a sequence of one or more occurrences of \text{Construct} separated from each other, if more than one, by \text{separator}. An example for the usage of this feature is given by the multiple branch
3. There is also a possibility for the tool to convert the instance definition into some binary format that can be read and used by an executable form of the tool. Since the instance definition is written in a very small subset $T$, this can be done relatively easily.

3 The Configuring Grammars

This section presents the input grammars and how they are used to configure the working of Soop. The exposition starts with a high level description of how all three input grammars should be prepared. The $G_S$ input grammar and how it affects the generation of class and object hierarchies are described next. The section is terminated with the explanation of how BNF was extended to describe the generation of output, as done by the grammars $G_C$ and $G_I$. 
2.5 The Overall Structure of SOOP

An overall diagram showing the structure SOOP together with that of all its components is presented in Figure 6. The reader is advised to use this figure as a road map.

The core of the overall process is the source code of a compiler-compiler parser-generator library (parserGen.icn) which stands at the same level as the source code of the yacc program. The source code of the parser-generation routine (parserGen.icn) is compiled by the class processor together with the class generation routine (class.icn). The resulting executable program is a parser generator (parserGen) called the source compiler. Take note that the source compiler includes the knowledge of how a grammar GS can be translated into a class definition in T.

The next step is to run the source compiler, supplying as input to it an ASCII file (sGrammar.def) containing the specification of the source grammar GS. In response to this input, the source compiler outputs two files: a parser routine in ICON (parser.icn) and a class definition file in T (class.def). The class definition file is the desired class-type definition, dependent only on S and GC.

The parser routine (parser.icn) is compiled together with the instance generation routine (instance.icn), as supplied by the instance processor. The resulting executable program (parser) is the long awaited SOOP’s parser for S. Clearly, the parser includes the knowledge of how a program of S can be translated into an instance of CT(GS).

Despite the apparent complexity, usage of the tool is quite simple, since execution in the right order of the components is driven by a simple make file script.

2.6 The Working of a Client Tool

By running the parser on a program of S, we get a representation of this program as an Instance Definition in T. This representation, together with the class definition, is at the disposal of any potential client tool written in T. Obviously, the class definition must be compiled together with the tool source code. There are several ways in which the instance definition could be used by the client tool.

1. The client tool may include the instance definition as part of its code, depending on the T’s compiler to translate the instance definition into an actual data object.

2. If the language T supports separate compilation, then the instance definition could be compiled and stored in object code format (.o file in UNIX). A client tool would then only need to link (statically or even dynamically) this object file together with its own, the client object code, in order to process a given program.
2.4 The Language Processor

The language processor uses the class generation executable procedure (parserGen), called the source compiler, as a digestion of the class grammar, and the instance generation routine (instance.icn) as a digestion of the instance grammar. Given as input the source language grammar (sGrammar.def) it produces, the desired parser program (parser), and the class definition file (class.def).

Figure 5: The structure of the language processor.
how SOOP can be used by a client tool.

### 2.2 The Class Processor

A run of SOOP commences with the execution of a class processor. The *Class Compiler* (classGen) reads an ASCII file (cGrammar.def) containing the class-synthesizing grammar specification $G_C$, and translates it into an Icon grammar-driven routine (class.icn). This routine (class.icn), called the *class generation routine*, takes as input an Icon internal stream of records, each representing one of $S$ language constructs (as deduced from the grammar $G_S$), and produces for each such record a class definition in $T$. The class generation routine (class.icn) is then compiled together with an Icon parser-generator source code routine (parserGen.icn) into the language processor’s source compiler (parserGen), which is described in Subsection 2.4.

![Figure 3: The structure of the class processor.](image)

### 2.3 The Instance Processor

Before the language processor can be invoked, the instance processor has to be called in order to prepare other data which the language processor relies on. The *Instance Compiler* (instanceGen) reads an ASCII file (iGrammar.def) containing the instance-synthesizing grammar specification $G_I$, and translates it into an Icon grammar driven routine (instance.icn). This routine (instance.icn), called the *instance generation routine*, takes as input an Icon data structure which represents the parse tree of a program in $S$. The output of the instance generation routine is an initialization sequence in $T$ of an object of type $C_S(G_I)$. The structure of this object preserves the parse information. The instance generation routine (instance.icn) is then passed to the language processor, which will make it part of the parser it generates (parser), as described in Subsection 2.4 below.
Handling each of the input grammars is done by a distinct component of SOOP: the language processor, the class processor and the instance processor. Figure 1 shows the relationships between these components. The class and instance processors produce an internal form representation of $G_C$ and $G_I$. The language processor then uses these representations in digesting $G_S$ to produce the parser of $S$ and the definition of the class $C_T(G_S)$.

![Diagram](image)

**Figure 1:** The structure of SOOP

All three components operate by reading a grammar and translating it into a program. A template of the common structure of all of them is depicted in Figure 2. The heart of a component is a grammar compiler which translates the component’s input grammar into a source code of an ICON routine. This routine is “grammar-driven”—its possible executions have a one-to-one correspondence with the possible derivations specified by the input grammar. The ICON compiler is then used to compile the grammar-driven routine along with other pieces of code (possibly supplied by the other components), into a grammar driven executable procedure.

![Diagram](image)

**Figure 2:** A template for the structure of SOOP’s components

We now turn to briefly describe the specifics of the structure of each component and then
2 The Structure of SOOP

Many traditional compiler-compilers take the approach of making the automatically generated parser a part of the tool that uses it. For example, a user of yacc [12] must intermix in the same file both the grammar and the code for the tool which uses this grammar. In contrast to that, the main goal in our design was to establish a clear separation between the parser, the parsing information, and the tools that use the parsing information. This is because SOOP is intended for use by many independent client tools for processing the same input language.

Let $S$ be the language in which the input programs are written, and let $T$ be the language in which a client tool is written. (It could be that $T = S$, but this is not necessarily so.) Let $G_S$ be a context free grammar for $S$. SOOP takes the approach of viewing a program in $S$ as an instance of $G_S$, just as an object is an instance of a class. Given $G_S$, SOOP generates a definition in $T$ of a class $C_T(G_S)$, such that any input conforming to $G_S$ can be represented as an instance of $C_T(G_S)$. The data-definition of $C_T(G_S)$ is such that it gives easy access to the parsing information of programs of $S$.

In addition to the class definition, SOOP generates a parser which compiles $S$ programs to instances of $C_T(G_S)$. Standardization of the representation of these instances makes the compiled form accessible to all tools which use the class definition $C_T(G_S)$. Clearly, there are many possible formats for the representation of instances of $C_T(G_S)$. In the current implementation of SOOP we selected a representation as a data definition in source form of the language $T$, i.e., the parser translates a program of $S$ into an initialization sequence in $T$ of a variable of the type $C_T(G_S)$. This allows a rapid prototyping, relying on the compiler of $T$ to do the task of translating an external representation of instances into an internal one.

The main penalty of this approach is in the increased structural complexity. The compiler of $T$ has to be invoked for every processed input program. Applying a client tool to an input program is done by compiling together the tool with the translation of the input program into an initialization sequence. Much of the incurred complexity is hidden from the user by a make script which encapsulates the inner mechanisms of SOOP. This script is responsible for maintaining, updating, and regenerating SOOP related programs and files.

2.1 The Principal Components of SOOP

The input to SOOP consists of three grammars: $G_S$, $G_C$ and $G_I$. The grammars $G_C$ and $G_I$ are subsets of a context free grammar for $T$, describing the syntax of class definitions and object instantiations, respectively. The grammar $G_C$ defines how SOOP should produce a data-type definition $C_T(G_S)$ in the language $T$, while $G_I$ sets the format of the output of the parser which SOOP generates.
weight in the industrial world generated many independent attempts [18, 26, 8, 23] of building systems for translating a C++ source code into a more accessible form.

GENO A [4] is a language-independent system for code analysis that can be interfaced to parse trees generated by other compilers. In comparison, SOOP is a complete environment which also encompasses the task of parsing the input language. In that it should be compared to the Grail environment, which is currently under development at the AT&T Bell Laboratories under the leadership of Bjarne Stroustrup. To the best of our knowledge, neither a report on the design of Grail, nor any account of progress in this project have been published.

1.4 Contribution

The idea of viewing production rules as classification and inheritance for compiler-compilers, is not new. Several text books [9, 1] indeed stress that a program could be written which would accept a suitable grammar and produce recursive subroutines, written in some language, for that grammar. We simply took the challenge. The novelty of the tool lies in its:

**Generality:** The tool can be configured for many input languages. Any language for which a grammar is supplied can be parsed. The set of class hierarchy definitions and instances can be generated in any object-oriented target language.

**Simplicity:** Only the language grammars are needed. The grammar is expressed in an easy-to-specify variant of the Backus-Naur form (BNF). For generating a class definition for EIFFEL, for instance, the EIFFEL grammar was copied almost as is from Meyer [16].

**Synthesizing grammar:** Code generation is expressed by using merely a synthesizing-grammar. The synthesizing-grammar applies very similar notations to those used for parsing-grammars.

**Experimenting grounds:** The open architecture approach and high-level code generation (and the choice of ICON [16, 25] for SOOP implementation), facilitates experimenting, program testing, measurements, and reuse.

Outline  The rest of this paper is organized as follows. The following section presents the tool more formally and gives some insight into its design and structure. Section 3 discusses the input grammars and their usage for configuring the tool, and exposes the untraditional synthesizing-grammar. An account of the experiment of configuring SOOP to EIFFEL is given in Section 4. Finally, Section 5 gives the conclusions. A full example of the parsing grammar of a somewhat trivial language, and its translation to C++ and SMALL TALK, is enclosed in Appendix A.
Supplied with these grammars, SOOP produces:

1. A *library* containing a class-type declaration with basic methods.


A software-engineering client tool may use the generated library and parser for producing instances, and process the instances for performing the desired analysis. It is quite possible to use the client tool’s source language as the instance-synthesizing target language. In such a case, the client tool’s compiler could process the generated classes and instances. The generated code could then be included by the client tool, or linked with the client tool, or even dynamically linked at run time. There is also a possibility of writing a supplemental tool that would convert the synthesized code into a binary format. However, this is not currently part of the presented SOOP.

### 1.3 Related Work

Recursive-descent and predictive parsing are widely used in practice. Because of its flexibility, recursive-descent parsing was used in many early compiler-writing systems [1]. A recursive descent top-down parser analyzer has one recursive procedure for each syntactical construct, which parses phrases for that construct. The procedure matches its phrase by comparing the program at the point indicated with right-hand-side parts of rules (for the construct), calling other procedures to recognize subgoals when necessary. A predictive parser is also a common practice in PROLOG [11]. A detailed object-oriented approach which integrates the classification and inheritance concepts into both the specification and generation of compiler and compiling process, are presented by Wu and Wang [30, 28]. In particular, methods for treating grammar symbols as class templates which instantiate nodes of a parse tree for object-oriented semantic analysis, are brought in [27]. The tool we have built employs their terminology and definitions.

There are many existing systems which translate programs of a specific language into an internal representation which admits further processing by specialized tools. These include, for example, DIANA [6] for ADA, Mjølner BETA system [19] for BETA, and the MENTOR project [5] for PASCAL. In addition, many interpreted languages including LISP, PROLOG and to some extent SMALLTALK, allow manipulations of programs as data. In contrast to all these systems, SOOP is not language specific. It is applicable to any programming language whose syntax can be expressed as a context free grammar.

A controversial language, but yet a very important one which SOOP cannot readily support, is C++. This is because C++ syntax is not context free.\(^1\) Its complexity, size and

---

\(^1\)For example, the C++ phrase `a b(c)` is either a function declaration or a definition of a data object depending on the types of the identifiers a, b and c
generating a comprehensive knowledge base of it; other client tools process the source code by scanning the knowledge base. SOOP provides a potential client software engineering tool with:

1. A data-type declaration representing the knowledge base structure.
2. A set of basic methods for handling instantiations of the data-type.
3. A data-structure instance representing a specific analyzed program as a data-type.

An object-oriented approach is taken for representing the knowledge base. With this approach classes correspond to syntactical constructs of the language and objects are nodes in the parse tree [27, 28]. Classes are used as data-type declarations, and instances as data-structures. The entire parsing-grammar is a set of class hierarchy definitions whose instantiation is a parse-tree of a source code program. Production rules are either inheritance classes of the left-hand-side into single alternative symbols of the right-hand-side, or composition classes of the right-hand-side components into the left-hand-side symbol. This is accomplished by restricting the grammar to either inheritance rules or composition rules, respectively.

SOOP is capable of handling different source code languages and different target class-type representations. This versatility is realized by configuring SOOP with three grammars:

1. A parsing-grammar that defines the programming language in which the source code is written.
2. A class-synthesizing grammar that defines the syntax of class declarations in the target language.
3. An instance-synthesizing grammar that defines the form in which the parse-tree of an analyzed program is to be expressed as a data-object instantiation of the class-type.

The three grammars share the following properties:

- They are expressed in an easy-to-use variant of Backus-Naur form.
- No separate lexical information is required. Tokens may express their matching procedures explicitly.
- They apply concepts of success and failure of computation, borrowed from logic goal-directed programming, to drive the control structure.

The two synthesizing grammars take a novel approach of using generators and streams to allow a natural and concise way for phrasing code synthesizing. This is discussed further in Section 3.
1 Introduction

1.1 Motivation

Software productivity is greatly affected by the existence of good tools for source code handling. Indeed, a plethora of such tools exists in almost every programming environment. The UNIX system, for instance, provides, as a standard, the C programmer with a flow chart extractor (cflow), a cross-reference generator (cxlabel), a heuristic checker (lint), code beautifiers (cb and indent), revision control programs such as RCS ([24]), a dependency manager (make [7]), and more. A variety of non-standard tools, like syntax directed editors ([21, 20, 13]), are also available.

Many tools for manipulation and managing source code follow the tradition of Source Code Control System (SCCS best paper awarded [22]) in treating a program as a mere collection of text files. However, the need for complex processing tools, such as tools for test cases generation, tools for software metrics ([3, 2, 15]), tools for typical error patterns search, and many more, require a detailed analysis of the source code. As a result, widely available tools do a compiler’s job and parse the source code in varying levels of thoroughness. In the above UNIX source handling tools, cb and indent perform a relatively higher degree of analysis, indent has a “forgiving” parser, and lint uses the C compiler front-end. In general, repeated re-implementation is a sure recipe for failure. Some tools fail in some situations where a complete analysis is required. It is difficult to maintain the conformity of all tools with different languages, or even with different versions of the same language. For example, indent fails on ANSI C code. Moreover, the difficulty in implementing a parser might prevent bringing theoretical advances in software engineering to use in practice.

Although commercially available environments provide the user with a coherent set of tools that employ a common single complete language parser, most of these environments feature a close architecture which is not amenable to extension by the user or to experimentation by the scientist. This situation is somewhat remedied by the availability of tools which are distributed in source form. For example, the GNU Free Software Foundation compiler gcc is supplied with the source code. Having the code available allows potential reuse of the compiler’s parser. Undoubtedly, a compiler is made for the complicated task of efficient code generation in the target language. However, it is not easy to adopt a compiler to do other jobs. Accessing the code is a cumbersome task. With every new version heavy modifications are required, and generally this does not provide a satisfactory solution.

1.2 The Tool

In this work we present a tool called Soop, that takes one step towards an open-architecture extensionable-system model where a single analyzer is responsible for parsing the code and
SOOP — A Synthesizer of an Object-Oriented Parser

(Preliminary Version)

JOSEPH GIL†  DAVID H. LORENZ‡

The Faculty of Computer Science,
Technion—Israel Institute of Technology,
Technion City, Haifa 32000, ISRAEL;

LPCR Report #9404
April 1994

Abstract

We present a tool for the use of software engineering tool builders. The tool parses programs and synthesizes them into a hierarchy of object-oriented classes and objects. This hierarchy lays the ground for further processing and experimentation with the parsed knowledge. Three grammars are used for configuring the tool: a source parsing grammar, a target class-synthesizing grammar, and a target instance-synthesizing grammar. The parsing grammar defines the programming language in which the parsed program is written. The class-synthesizing grammar defines the language used for generating code describing the structure of classes and their inheritance hierarchy. The instance-synthesizing grammar defines how objects should be generated. Different source languages and target languages may be specified by supplying their grammars. A client tool can use the class-type declarations and the instantiated objects to access the parsed information. Several configurations for existing programming languages as input (such as Eiffel), and existing object-oriented class definitions as output (such as Smalltalk and C++), were implemented. We present an enhancement of the Backus-Naur form that allows a description for synthesizing programs’ text rather than parsing.

†E-mail address: yogi@CS.Technion.AC.IL
‡Research supported in part by the Fund for promotion of research at the Technion
§E-mail address: david@CS.Technion.AC.IL