# Syn: A Single Language for Specifying Abstract Syntax Trees, Lexical Analysis, Parsing and Pretty-Printing*

*Richard J. Boulton*

University of Cambridge Computer Laboratory
New Museums Site, Pembroke Street
Cambridge CB2 3QG, United Kingdom

March 1996

**Abstract**

A language called Syn is described in which all aspects of context-free syntax can be specified without redundancy. The language is essentially an extended BNF grammar. Unusual features include high-level constructs for specifying lexical aspects of a language and specification of precedence by textual order. A system has been implemented for generating lexers, parsers, pretty-printers and abstract syntax tree representations from a Syn specification.

## 1 Introduction

Syn is a language for specifying the syntax of computer languages. Lexical analysis, parsing, pretty-printing, and abstract syntax trees can all be specified in a single compact notation similar to Backus-Naur Form (BNF) grammars. However, the language is not as flexible as individual languages for specifying parsers, etc. The intention is to have a concise specification of syntax with sufficient flexibility and expressive power to be useful for a significant range of realistic languages. The main reason for having a single language is to eliminate redundancy and the consequential difficulty in maintaining consistency.

The most notable features of Syn are:

*Research supported by the Engineering and Physical Sciences Research Council of Great Britain under grant GR/J42236.

- Optional and repeated syntactic elements can be specified directly.
- One notation acts as both a means of specifying options and repetitions, and as a specification of layout for pretty-printing.
- The precedence (binding strength) of terminals is specified implicitly by textual ordering with the aid of dependency analysis on the non-terminals.
- High-level constructs are available to specify lexical features.
- Syn has no reserved words, so there are no ad hoc restrictions on the names that can be used for syntactic categories.

The high level of description in the Syn language allows specifications in more specialised (and usually more expressive) languages to be generated from a Syn specification. It is also straightforward to generate variations within one of these target languages such as a parser that keeps track of source-code line numbers and one that does not, according to the user's requirements, without any need to change the Syn specification.

The design of Syn revolves around an assumption about the form of the abstract syntax trees to be generated, namely that each node of the tree has an $n$-tuple (where $n \geq 0$) of descendants where each descendant may be a single tree, an optional tree, or a list of trees. This assumption was partly motivated by the aims of the wider project in which Syn has been developed: to represent abstract syntax in higher-order logic [Chu40] using the HOL theorem proving system [GM93], so that language texts can be given a formal semantics and reasoned about using mechanised formal proof.

The shape of the abstract syntax trees for a specified language does not have to be given explicitly, but is deduced from the grammar. The notions of parse tree and abstract syntax tree correspond in Syn. This should not be a serious restriction because the language allows quite abstract trees to be specified even under the restrictions imposed by having to be able to generate parsers and pretty-printers. However, if the trees are not sufficiently abstract, one possibility would be to have two Syn specifications, one giving the concrete syntax (and the abstract syntax that follows from it) as usual, and another that gives the abstract syntax that is really required without any concrete syntax. The user would then only have to write a program to translate between the two forms of abstract syntax.

The remainder of this paper contains the following: a Syn specification for a simple imperative language and a brief explanation of the features; a discussion of related work; a detailed description of the Syn language; a presentation of key algorithms used in processing the language; a description of a compiler for Syn implemented in the functional programming language Standard ML; and finally, some conclusions.

## 2 Syn for a Simple Imperative Language

Figure 1 is a Syn specification for Imp, a simple imperative programming language.

The specification begins with the name of the language followed, in parentheses, by the name of the syntactic category that constitutes a full program in the language. The remainder of the specification falls into two parts: lexical declarations and the 'grammar'.

The lexical declarations contain a specification of white-space (spaces and tabs), of the new-line symbol(s) (carriage return), and of comments. A function, 'quote', is used to specify comments as any text enclosed between '(*' and '*)'. Regular expressions are used for the white-space and new-line.

A uniform syntax is used for the remaining declarations but there are actually three distinct forms: non-terminal definitions, sets of fixed terminals, and definitions for data-carrying terminals. The last type is essentially lexical information, though it also controls the data type used for the carried value in the abstract syntax tree (AST). If no type is given, the default used is 'string'.

The non-terminal definitions are essentially BNF augmented with pretty-printing information and the name to be used for the AST node. The latter appears in parentheses at the beginning of each alternative. The AST node name is omitted in the final alternative of each of the integer and Boolean expression categories. This allows parentheses to be consumed without building an AST node.

The remainder of each alternative of the non-terminal definitions consists of nested boxes, i.e. items of the form [<...> ...]. Stripping out the boxes would leave a linear sequence of terminals and non-terminals much as in BNF. The boxes perform two functions: they specify how the language is to be pretty-printed, and they also allow sequences of terminals and non-terminals to be optional or repeated.

There are four kinds of box for pretty-printing purposes, of which three are illustrated in the example. Two items (terminals or non-terminals) that are to appear on the same line when an AST is displayed in the concrete syntax are placed in a *horizontal* box. For example, '<h 1>' specifies that the items are to be separated by one blank character. The '<hv ...>' and '<hov ...>' boxes allow the items to be displayed on the same line or on multiple lines according to how much space is available. The breaking across lines is done inconsistently and consistently, respectively. (See Section 4 for more details on this.)

Optional and repeated boxes are specified using a syntax like that of regular expressions: ? specifies an optional occurrence, * specifies zero or more occurrences, and + specifies one or more occurrences. The conditional statement of Imp illustrates optional boxes. The else branch is optional. The Block construct illustrates repetition. The parentheses around the repeated box and the lone com indicate that this com should be added to the list used for the repetition.

```
Imp (com):

#whitespace = {[\ \t]+};
#newline = {\n};
#comment = #quote("(*","*)");

com  ::= (Skip) "skip"
       | (Assign) [<hv 1,3,0> [<h 1> name ":="] iexp]
       | (If) [<hov 1,0,0> [<h 1> "if" bexp]
                           [<h 1> "then" com]
                           [<h 1> "else" com]?]
       | (While) [<hov 1,0,0> [<h 1> "while" bexp] [<h 1> "do" com]]
       | (Block)
           [<hov 1,3,0> "begin" ([<h 0> com ";"]* com) <1,0,0> "end"];

iexp ::= (Integer) int
       | (IntVar) name
       | (IntUnApp) [<hv 1,3,0> intunop iexp]
       | (IntBinApp) [<hv 1,3,0> [<h 1> iexp intbinop] iexp]
       | (IntCond) [<hv 1,3,0> bexp [<h 1> "=>" iexp] [<h 1> "|" iexp]]
       | () [<h 0> "(" iexp ")"];

bexp ::= (Boolean) bool
       | (BoolUnApp) [<hv 1,3,0> boolunop bexp]
       | (BoolRelApp) [<hv 1,3,0> [<h 1> iexp relop] iexp]
       | (BoolBinApp) [<hv 1,3,0> [<h 1> bexp boolbinop] bexp]
       | () [<h 0> "(" bexp ")"];

intunop ::= "~";

intbinop ::= "+" (l) =|= "-" (l) | "*" (l) =|= "/" (l);

boolunop ::= "not";

boolbinop ::= "or" (l) | "and" (l);

relop ::= "=" =|= "<" =|= ">" =|= "<=" =|= ">=";

int ::= #integer {[0-9]+};

bool ::= #boolean;

name ::= {[A-Za-z][A-Za-z0-9_']*};
```

Figure 1: Syn specification for a simple imperative language

This is referred to as *grouping*.

Precedence is specified implicitly by the ordering of alternatives. Earlier constructs within each non-terminal have a higher precedence (are more tightly binding). Between non-terminals the precedence is based on their dependencies. When two or more non-terminals are mutually dependent the conflict is resolved as before using the textual ordering.

The sets of terminals are used to avoid having to duplicate complex formats for each of a number of terminals. They consist of a list of strings each with an optional (left or right) associativity. If no associativity is given, the terminal is taken to be non-associative. The syntax '`=|=`' is a variant form of alternation. It simply expresses the intention that the two terminals it sits between should have the same precedence rather than the first having a lower precedence than the second.

Hopefully, the example gives the reader a useful impression of the Syn language. The language is described in more detail in Section 4.

# 3 Related Work

There has been a considerable amount of previous research on specification languages for syntax. The lexical analyser generator Lex [Les75] and parser generator Yacc [Joh78] are perhaps the best known. Most of the work has been directed towards parsing, and languages that also allow specification of pretty-printing are rare. They do, however, arise in generators for software engineering environments, e.g., the Ergo Support System (ESS) [LPRS88], the programming system generator PSG [BS86], CENTAUR [BCD+88] and the Synthesizer Generator [RT89].

The syntax specification language of ESS is quite similar to Syn. It is a single language that has iterators, unparsing annotations, and high-level lexical specification. However, the Syn language has more implicit features such as inferring precedence and the form of the ASTs. ESS makes use of attributes and higher-order abstract syntax which the implementation of Syn currently does not. ESS also supports sub-languages which Syn does not.

PSG has one language for specifying all aspects of syntax and both static and dynamic semantics. However, the various aspects of syntax are specified separately within the language, so there is a lot of redundancy. In particular, the tokens used between the lexer and parser have to be named and the construction of ASTs has to be specified explicitly. The form of identifiers, integers, reals and strings are assumed and fixed. Other basic lexical entities cannot be defined, restricting the range of languages to which the system is applicable. The concrete syntax is restricted to LL(1) grammars with elimination of left recursion having to be done by hand. Also, no mention is made of how precedence is dealt with for parsing. Pretty-printing is specified as another grammar with annotations for insertion of new-lines and indentation. The formatting may be conditional on the existence of optional syntactic elements or on the form of sub-ASTs.

CENTAUR provides specification languages for concrete and abstract syntax (Metal [KLMM83] and SDF [HHKR89]), for pretty-printing (PPML [MC86]), and for static and dynamic semantics, together with a powerful user interface. The ASTs in CENTAUR can have attributes and annotations such as comments and formal assertions. The leaves may be ASTs of sub-languages.

A Metal specification gives rise to input for Yacc and so is restricted to LALR(1) grammars. For SDF (the Syntax Definition Formalism) there are no restrictions on the context-free grammar used (though this may lead to ambiguities) and the lexers and parsers are built incrementally.

SDF is a sophisticated language having a lot in common with Syn. The form of the ASTs is inferred, and notation is included for repeated syntactic elements but not for optional ones. However, there are no high-level lexical constructs apart from a special category for white-space, and no means of specifying pretty-printing. The latter is omitted on the grounds that it would make the specifications too difficult to read. The relative precedence of operators and groups of operators can be specified but, unlike in Syn, this is done explicitly. It is also possible to specify meta-variables that can be used in place of pieces of concrete syntax in a language text. These are used for syntax-directed editing and also allow a semantics for the language to be given over the concrete syntax. The form of meta-variables is specified using regular expressions. The SDF Reference Manual [HHKR89] has a good discussion of other syntax-specification formalisms.

PPML (Pretty-Printing Meta-Language) is not based on a grammar for the language but instead features patterns for matching abstract syntax trees and corresponding formats that specify how strings and the text for subtrees are to be grouped and positioned. The pretty-printer specifications in Syn were inspired by PPML. One advantage of having a separate language for specifying pretty-printing is that more than one style of pretty-printing can be used.

The Synthesizer Generator creates a syntax-directed editor for a language from separate specifications of abstract syntax, formatting, concrete input syntax, etc. The specification of syntax includes context-sensitive properties such as type consistency. The specification languages are based on attribute grammars. Indentation and line breaking are specified in an unparsing scheme by means of control characters in the output strings. An unparsing scheme also specifies which abstract syntax tree nodes are selectable and which productions of the grammar are editable as text. Lists of syntactic elements and optional elements are considered.

Rubin [Rub83] describes a language based on context-free grammars that includes lexical and pretty-printing specifications, but no abstract syntax. The pretty-printing annotations take the form of actions to be performed for each symbol of a production, such as starting a new line and indenting by a specified number of spaces. These annotations are similar to the control characters in the Synthesizer Generator. Rubin's work is motivated by the desire to build syntax-directed editors for software engineering environments.

Rose and Welsh [RW81] describe grammars extended with formatting operations that manipulate a stack of margin values. One of their aims is to move specification of formatting out of the realm of language implementors and users into that of the language designer. They also discuss formatting of comments, a difficult issue that is rarely addressed.

In contrast to Syn, which has been developed to support formal reasoning about computer languages, software engineering environments emphasize pretty-printing for syntax-directed editing. For such structural editing it is important that the output on a screen can be changed incrementally since full recomputation of a text of several hundred lines is too slow to be performed at every key stroke.

# 4 The Syn Language

A Syn specification for the Syn language is given in Appendix A[1]. This section describes the features of the language in detail.

A Syn specification consists of a header, lexical definitions, and definitions for the syntactic categories of the language. The header specifies the name of the language being defined, and optionally, the name of the category that constitutes a full 'program' in the language. If no category is specified, the first category in textual order that has abstract syntax tree (AST) node names is used.

## 4.1 Lexicals

Each lexical definition begins with '`#`' and ends with '`;`'. An identifier follows the sharp sign, then optionally an equals sign and either a regular expression or a function call. Currently, the valid identifiers are `whitespace`, `newline`, `comment`, and `case_insensitive`. The lexicals `whitespace` and `newline` should be specified by regular expressions and can appear at most once. If not present they default to:

```
#whitespace = {[\ \t]+};
#newline = {\n};
```

Any number of comments (zero or more) may be specified. A function rather than a regular expression is expected as each comment specification. The character sequences specified by the comment functions will be ignored by the lexer.

The `case_insensitive` lexical, if present, instructs the lexer to allow alphabetic terminals (terminals consisting of letters only) to be given in any mixture of

[1] The escape sequences allowed in the category `sconst` are more extensive than the Syn specification would imply. This illustrates a limitation of Syn. Also, the tilde '`~`' used in the category `iconst` represents the unary minus sign.

upper and lower case. For pretty-printing, the forms given in the Syn specification are used.

A function application begins with a '`#`' followed by an identifier and arguments in parentheses. There can be zero or more arguments separated by commas but there must be the correct number for the function. Each argument is a string constant. The use of the sharp sign ('`#`') distinguishes lexical names and function names from user identifiers. The intention is to avoid taking names as reserved words (keywords) of the Syn language that the user might wish to use for the language being defined.

## 4.2 Functions

The functions currently available are:

```
#quote(l,r)
#escape_quote(l,e,r)
#doubleup_quote(l,r)
#nested_quote(l,r)
#quote_in_line(l,r)
#escape_quote_in_line(l,e,r)
#doubleup_quote_in_line(l,r)
#nested_quote_in_line(l,r)
```

All these functions specify quotations, with left ($l$) and right ($r$) delimiters. The function `escape_quote` also has an argument specifying the character ($e$) to be used to introduce an escape sequence. The function `quote` specifies a simple quotation; the right delimiter cannot appear inside the quotation. `escape_quote` allows the right delimiter to appear in the quotation by preceding it with the escape character. The escape character can be included by preceding it with itself. In fact, the escape followed by any character is read as the character only. The function `doubleup_quote` allows the right delimiter to appear by writing it twice. `nested_quote` allows quotations to appear inside quotations to arbitrary depth. A new-line character is allowed to occur inside these quotations. The '`in_line`' versions prohibit this.

## 4.3 The Syntactic Categories

There are four kinds of syntactic category definition: for data-carrying terminals, for sets of fixed terminals, for non-terminals, and for aliases. All four are of the form:

```
<identifier> "::=" <body> ";"
```

It is in the body that they differ. They are described in the following subsections.

### 4.3.1 Data-Carrying Terminals

The form of each data-carrying terminal of the language being defined is specified by a regular expression. An example of such a terminal is one for identifiers. The data in this case is the name of the identifier. The regular expression is used by the lexer. As for the lexicals, a function (see Section 4.2) can be used instead of a regular expression. For example, string constants would normally be specified as a quotation using one of the functions.

By default, the data carried by the terminal is taken to be a string. However, the user may specify some other suitable type for the terminal. The current options are: `string`, `character`, `boolean`, `natural`, `integer`, `rational`, and `real`, but only some of these may be supported for any given implementation language. Each type has a default regular expression or function associated with it. The user may override these defaults but the compiled code may not then correctly interpret the user's specification in producing the data to be carried by the terminal. The defaults are:

```
string    : escape_quote("\"","\\","\"")
character : {'.'}
boolean   : {true|false}
natural   : {[0-9]+}
integer   : {-?[0-9]+}
rational  : {-?[0-9]+/[1-9][0-9]*}
real      : {-?[0-9]+(\.[0-9]+)?(E-?[0-9]+)?}
```

Terminal types are identifiers preceded by '`#`'.

A data-carrying terminal may be made left or right associative by postfixing the specification with '`(l)`' or '`(r)`'. This is useful for user-defined infix operators in a language.

### 4.3.2 Sets of Terminals

Languages often have a fixed set of operators. It is inconvenient to have to give a production for each operator when these are the same except for the operator itself. For this reason, fixed terminals (constant strings) can be grouped together to form sets of terminals. A syntactic category is defined for each set of terminals and only one production has to be used bearing the name of the category.

The terminals in a set are string constants separated by one of the alternation symbols, '`|`' or '`=|=`'. The second form specifies that the terminals to its left and right are to have the same precedence. Otherwise, the earlier a terminal is in the text, the lower is its precedence (binding strength). Each terminal may be specified as left or right associative by postfixing the string with '`(l)`' or '`(r)`', respectively.

### 4.3.3 Non-Terminals

A non-terminal category is specified as a collection of alternative productions. Each alternative consists of an optional AST node name in parentheses followed by a format. The Syn compiler deduces the branches of the AST from the non-terminals in the format. If no AST node name is given then there must be precisely one non-terminal in the format and it must be the non-terminal category being defined. This restriction is necessary because no AST node will be built for the format. The facility is intended to allow parentheses to occur around an expression without a corresponding AST node being constructed.

An AST node name may be used more than once within the same category but not in different categories. Additionally, the formats must be consistent for each occurrence, that is they must give rise to the same AST node. When a node name is used more than once, exactly one of the uses must be followed by an asterisk (‘`*`’) to indicate that the corresponding format is the one to be used for pretty-printing.

A format plays the role of both a grammar extended with iteration constructs and a specification of how to lay out the concrete syntax when generated from the abstract syntax tree. Iteration constructs specify that a sequence of terminals and non-terminals can be repeated. The same effect can be achieved using recursive syntactic categories but there are two limitations of recursion. First, it leads to collections of abstract syntax that naturally belong together occurring as nested AST branches. Second, it is difficult to specify a good layout for pretty-printing.

A format, then, is either a terminal (a string constant possibly annotated by an associativity), a non-terminal (an identifier), or a box (denoted by square brackets). A box may be empty indicating an empty string of characters in the concrete syntax, or it may contain a list of objects (nested formats to a first approximation). There may also be a specification of the layout, and a box may be followed by a modifier to indicate that it is optional (‘`?`’) or can be repeated zero or more times (‘`*`’) or one or more times (‘`+`’).

Layout specifications for a whole box take one of the following forms:

<h $h$>
<v $i$,$v$>
<hv $h$,$i$,$v$>
<hov $h$,$i$,$v$>

The first form specifies that the objects of the box are to be positioned on the same line separated by $h$ spaces. The second form specifies a vertical layout in which each object is placed on a separate line (separated by $v$ blank lines) and the second and later objects are indented by $i$ characters. Normally the indentation is relative to the first object but if the integer value used is preceded by ‘`++`’ then the indentation is relative to the previous object. The third form (horizontal-vertical) specifies that as many objects as possible should be placed on one line, and a new line begun when there is no more space, with indentation of $i$. The final form (horizontal-or-vertical) specifies that all the objects should go on one line or they should be formatted vertically. The last two forms are often referred to as *inconsistent* and *consistent* breaking, respectively.

As an example, consider the specification `<hv 1,3,0>`. With a wide output field this might give rise to the text:

```
<object1> <object2> <object3> <object4>
```

With less space the text could be:

```
<object1> <object2>
   <object3>
   <object4>
```

As another example, the specification `<hov 1,++3,1>` could give rise to:

```
<object1> <object2> <object3> <object4>
```

or to:

```
<object1>

   <object2>

      <object3>

         <object4>
```

If no layout specification is given in a box it is assumed to be a horizontal box with separation of 0. Layout information need not be given in a Syn specification. However, a printer generated from such a specification will produce a continuous stream of output with no line breaks.

The syntax and semantics of these specifications is based on the Pretty-Printing Meta-Language (PPML) [MC86] of the Centaur system [BCD$^+$88].

Each object in a format is either itself a format (with an optional layout specification) or a group of objects. A layout specification before an object is a list of separation values with commas between them and enclosed in angle brackets. The number and form of the separation values should correspond to the layout specification for the surrounding box. A layout specification before an object is used to override the separation values of the box with new values for that object only. An example of this can be found in the specification in Section 2 where the `end` of a block is aligned with the `begin`.

Objects may be grouped using parentheses to indicate that the non-terminals they contain should be made into a single branch of the AST node. This is only possible when the non-terminals are the same category and precisely one is inside a repetition. A list is then formed as the branch of the AST[2] by concatenating the non-terminals before the repetition to the list generated for it and concatenating the non-terminals after the repetition to the tail of the list. The non-repeated non-terminals in the group cannot be optional. Grouping is important not only for constructing sensible ASTs but also for generating the parser. A repeated non-terminal followed by the non-terminal alone may give rise to a parsing conflict if grouping is not used.

The syntax of formats allows specifications to be made that are inconsistent with the intention to have each AST node being a tuple where each component is a simple value, an optional simple value, or a list of simple values. (A 'simple value' is either an application of an AST node or an element of one of the basic types given in Section 4.3.1.) Inconsistent specifications should be rejected by any Syn compiler. Some formats that should not be used include: nested repetitions because they would require a list of lists of simple values; repetitions or options containing more than one non-terminal because they would require a list of or optional tuples or something similar; repetitions containing options. Nested options are allowed provided only one non-terminal is present as they reduce to a single option. When printing nested options the entire outer option is either printed or omitted depending on the value in the AST; the inner options are treated as simple boxes. Repetitions can be nested inside options since if the part of the format containing the repetition is not present an empty list can be used, while if it is present the list comes from the repetition as normal.

Repetitions and options that are free of non-terminals are allowed and can be nested arbitrarily since the AST is not dependent on them. However, for pretty-printing the absence of non-terminals implies that there is no way of knowing whether to print an option or not, or how many times to print a repetition. The behaviour chosen is to not print the contents of an option and to print the contents of a repetition the minimum number of times. Optional terminals are sometimes used as flags, in which case the presence or absence of the terminal needs to be recorded in the AST. To achieve this in the Syn language it is necessary to introduce a new non-terminal N which has the terminal as its only alternative, and replace the original optional terminal by an optional N.

### 4.3.4 Aliases

For an alias, the body of the syntactic category definition is simply an identifier. This defines the syntactic category to be an alias for the category named by the identifier, which may itself be an alias. This can be useful for a language which has identifiers for several different kinds of object, e.g. for integer variables and

[2]Conceptually, a list is more than one branch, but in the Syn scheme for ASTs, lists of a non-terminal and optional non-terminals can be elements of the tuple used to represent the branches.

for procedure names. If they are lexically distinct there is no problem. However, suppose they have the same regular expression, e.g.:

```
intvar   ::= {[a-z][a-z0-9]*};
procname ::= {[a-z][a-z0-9]*};
```

The lexer will not then be able to distinguish the two categories. The solution is to make them both an alias for one category:

```
intvar     ::= identifier;
procname   ::= identifier;
identifier ::= {[a-z][a-z0-9]*};
```

Whether an `identifier` is an `intvar` or a `procname` can be determined after the parse from the context. It cannot be done during the parse because the Syn language is restricted to context-free grammars.

## 4.4 Precedence

As in sets of terminals, the precedences of terminals within a non-terminal category are determined by textual order. Within an alternative the precedences of the terminals increase from left to right[3]. There is a decreasing order of precedence between the alternatives. When the category name of a set of terminals appears in an alternative the order of precedence for the set of terminals is inserted into the order for the category at that point.

Precedence between non-terminal categories is determined from their dependencies. If a category A is mentioned in the definition of category B then a relation is established stating that A (and all of its terminals) has a higher precedence than B (and all of its terminals). Mutually dependent categories give rise to a cycle in the relations. In such a case, the cycle is broken by ignoring one of the dependencies that violates the textual ordering of the category definitions.

When a terminal appears in more than one place, it may be given more than one precedence. The duplicates are removed by choosing the lowest precedence.

# 5 Key Algorithms

An implementation of the Syn language requires algorithms to extract various information from the parse tree for a Syn specification, and algorithms to generate code for input to lexer, parser, and pretty-printer generators, etc. The latter are tool-specific and so are not discussed in this paper (though see Section 6). The former are mostly straightforward, though optional and repetitive syntactic elements, when nested in particular, cause complications. However, a few of the algorithms used are far from obvious. These are described below.

[3]This ordering promotes the acceptance of optional syntactic elements by a left to right parse since a 'shift' of an optional terminal will be favoured over a 'reduce'.

## 5.1 Precedence Analysis

Precedences in Syn are inferred from the textual ordering. The algorithms to achieve this are described in the following sections. They should be viewed as experimental. In particular, the approach described is only one possible way of guessing precedences for the terminals of a language.

### 5.1.1 Left and Right Symbol Sets

The algorithm for inferring precedences for the terminals of a language requires that for each symbol (terminal or non-terminal) $s_{p,i}$ in a production $p$, two sets $L_{p,i}$ and $R_{p,i}$ be known[4]. $L_{p,i}$ is the set of symbols that may occur immediately to the left of $s_{p,i}$ and $R_{p,i}$ is the analogous set for symbols to the right. In a simple production these two sets are obvious. For example, consider the following alternative of the `com` category from Figure 1:

```
(While) [<hov 1,0,0> [<h 1> "while" bexp] [<h 1> "do" com]]
```

When stripped of formatting information it yields the following production:

```
com ::= "while" bexp "do" com
```

The sets $L$ and $R$ for the first symbol (`"while"`) are $\{\}$ and {`bexp`} respectively. For the second symbol they are {`"while"`} and {`"do"`}.

For productions containing optional and/or repeated symbols, computing $L$ and $R$ is not so straightforward. For example, the alternative

```
(Block) [<hov 1,3,0> "begin" ([<h 0> com ";"]* com) <1,0,0> "end"]
```

is essentially the following production:

```
com ::= "begin" [com ";"]* com "end"
```

For the second symbol (`com`), $L$ is {`"begin"`, `";"`}, the `";"` being included because of the repetition. For the fourth symbol (also a `com`), $L$ is the same set, but this time it is the inclusion of `"begin"` that may be unexpected. It is included because the repetition may occur zero times.

To compute $L$, the sequence of symbols is traversed from left to right with a set of immediately preceding symbols $S$ being maintained. The value of $S$ is saved on entry to an optional or repeated box. Let $S_b$ be its value on exiting the box. Then, the new value $S'$ to be used for further processing is as follows, depending on the type of the box:

**Optional** $S' = S \cup S_b$

[4]Actually, for the algorithm that follows it is sufficient to compute the sets for terminals only.

**Zero or more repetitions** $S' = S \cup S_b$

**One or more repetitions** $S' = S_b$

Multiple traversals are required to compute the $L$ sets for symbols inside a repeated box because $S \cup S_b$ has to be used on entry rather than $S$, and the former is not known until the box has been traversed.

The set $R$ can be computed using a similar technique applied from right to left.

### 5.1.2 Relations Between Non-Terminal Categories

Let the predicate $\mathsf{Term}$ be true of a symbol $s$ if and only if $s$ is a terminal or the name of a category that is either a data-carrying terminal or a set of terminals. Let $\mathsf{NonTerm}$ be true of a symbol $s$ if and only if $s$ is the name of a non-terminal category, i.e. a collection of alternative productions. Now, consider a non-terminal category $c$ with the following production $p$:

$$c \ \texttt{::=}\ s_{p,1} \ \ldots \ s_{p,n_p}$$

The adjacency set $A_{p,i}$ for each symbol $s_{p,i}$ is computed to be:

$$A_{p,i} = \begin{cases} \{s \in L_{p,i} \cup R_{p,i} \mid \mathsf{NonTerm}(s)\} & \text{if } \mathsf{Term}(s_{p,i}) \\ \{\} & \text{otherwise} \end{cases}$$

For a terminal symbol, $A_{p,i}$ is the set of non-terminals that may appear immediately to its left or right. For a non-terminal, $A_{p,i}$ is empty.

Now, for each category $c'$ in $A_{p,i}$ a relation $c < c'$ is established. The intuition behind this is that an operator in $c$ should have a lower precedence than operators in categories that form its arguments. This is no more than a heuristic. In particular, it treats all terminals as operators. The approach might be refined to treat categories to the left of an operator differently to categories to the right, based on the associativity (if any) of the operator. The $L$ and $R$ sets have been kept separate for this reason. Formally, the relation established between non-terminal categories is denoted by $<_{c,p}$:

$$<_{c,p} = \bigcup_{i=1}^{n_p} \{(c, c') \mid c' \in A_{p,i} \text{ and } c' \neq c\}$$

Note that no relation is established between $c$ and itself. Denoting the set of alternative productions for category $c$ as $P_c$ and the set of non-terminal categories as $C$, the full set of relations between such categories is:

$$< \, = \bigcup_{c \in C} \bigcup_{p \in P_c} <_{c,p}$$

| | |
|---|---|
| $<_{\texttt{com},\texttt{Skip}}$ | $\{\}$ |
| $<_{\texttt{com},\texttt{Assign}}$ | $\{(\texttt{com}, \texttt{iexp})\}$ |
| $<_{\texttt{com},\texttt{If}}$ | $\{(\texttt{com}, \texttt{bexp})\}$ |
| $<_{\texttt{com},\texttt{While}}$ | $\{(\texttt{com}, \texttt{bexp})\}$ |
| $<_{\texttt{com},\texttt{Block}}$ | $\{\}$ |
| $<_{\texttt{iexp},\texttt{Integer}}$ | $\{\}$ |
| $<_{\texttt{iexp},\texttt{IntVar}}$ | $\{\}$ |
| $<_{\texttt{iexp},\texttt{IntUnApp}}$ | $\{\}$ |
| $<_{\texttt{iexp},\texttt{IntBinApp}}$ | $\{\}$ |
| $<_{\texttt{iexp},\texttt{IntCond}}$ | $\{(\texttt{iexp}, \texttt{bexp})\}$ |
| $<_{\texttt{bexp},\texttt{Boolean}}$ | $\{\}$ |
| $<_{\texttt{bexp},\texttt{BoolUnApp}}$ | $\{\}$ |
| $<_{\texttt{bexp},\texttt{BoolRelApp}}$ | $\{(\texttt{bexp}, \texttt{iexp})\}$ |
| $<_{\texttt{bexp},\texttt{BoolBinApp}}$ | $\{\}$ |

Table 1: Relations between non-terminals in the Imp language

As an example, consider again the Imp language in Figure 1. The relations established for each production are given in Table 1. The overall relation is thus:

$$\{(\texttt{com}, \texttt{iexp}), (\texttt{com}, \texttt{bexp}), (\texttt{iexp}, \texttt{bexp}), (\texttt{bexp}, \texttt{iexp})\}$$

This illustrates a problem: the '<' relation is not necessarily a partial order. In this case, both `iexp` $<$ `bexp` and `bexp` $<$ `iexp`. The conflict is resolved by appealing to the textual ordering of the categories, as described in the following section. Earlier categories are given higher precedence, so `iexp` is given a higher precedence than `bexp`. The conflict arises in the example due to the presence of the conditional integer expression.

### 5.1.3 Ordering of Non-Terminals

The textual ordering relation $T$ for the non-terminal categories is such that $cTc'$ if $c$ appears before $c'$ in the Syn description. To resolve conflicts in the relation $<$, a directed graph is constructed with vertices $C$ (the set of non-terminal categories) and edges defined by $<$. A set of edges is then computed such that removal of the edges breaks all cycles in the graph and the edges removed all satisfy the relation $T$ (i.e. their precedences are in conflict with the textual ordering). The resulting acyclic graph is then topologically sorted to produce a total ordering[5] for the categories.

A number of simple and efficient approaches to breaking cycles in the graph are available. One is to remove all cycle edges that satisfy $T$. Another is to

[5] There may be more than one.

```
function break_cycles(P,Y);
var S, B;

   procedure best_break(S,B');
   begin
      if S = {}
      then begin if |B'| < |B| then B := B' end
      else begin
              pick an s ∈ S;
              for each edge e in s do best_break(S − {s},B' ∪ {e})
           end
   end;

begin
   S := ⋃_{y∈Y} {{e ∈ y | P(e)}};
   B := ⋃_{s∈S} s;
   best_break(S,{});
   return B
end;
```

Figure 2: Function `break_cycles`

break each cycle before the vertex (category) that appears earliest in the textual ordering. The disadvantage with these approaches is that they may remove more edges than necessary, in the latter case because two or more cycles may share an edge. A rather computationally intensive algorithm that minimises the number of edges removed is illustrated in Figure 2.

In the function `break_cycles`, $P$ is a predicate on an edge and $Y$ is the set of cycles. Each cycle is represented as a set of edges. $S$ is the result of removing from each cycle the edges that do not satisfy $P$. ($P$ is instantiated to $T$.) The set of edges $B$ to remove from the graph is initialised to all the edges in $S$. The recursive procedure `best_break` is then used to look for a smaller set of edges that break all the cycles. It tests each of the combinations of edges, taking one from each cycle, looking for a set smaller than the best found so far. When the new combination is the same size as the current best the one more consistent with the textual ordering could be retained, but such a refinement is not used here. The algorithm is optimised for space efficiency by testing each combination of edges as it is generated.

The topological sort used to obtain a total ordering of the non-terminal categories is a standard algorithm. The vertices of the graph that have no incoming edges are called the *initial vertices*. A total ordering can be obtained by repeatedly deleting the initial vertices of the graph, appending the vertices removed at each step to a sequence. A refinement is to remove vertices one at a time and, when there is a choice of initial vertices, to select the one that appears latest in the textual ordering.

### 5.1.4 Ordering of Terminals

Within each non-terminal category the precedence of the terminals is determined by textual ordering as specified in Sections 4.3.2 and 4.4. The result is an ordered sequence (i.e. a list) of sets. Each set contains terminals of the same precedence.

It is possible for a terminal to appear more than once in a Syn specification, giving rise to more than one possible precedence value. The lowest precedence is chosen. The other occurrences in the list of sets are removed. This may cause some of the sets to become empty in which case they are removed from the list. Choosing the lowest precedence is an arbitrary decision. More research is required to determine what should really be done, if in fact it is even possible to make a good choice in all situations.

Further difficulties may arise when committing to particular parser and pretty-printer generators. For example, they may not allow the same precedence to be given to more than one terminal, especially if the terminals have different associativities. In such cases, an approximation has to be used.

## 5.2 Grouping and Ordering of Categories

Dependencies exist between the syntactic categories of a language and some of these dependencies may be mutual, i.e. cyclic. When implementing abstract syntax, parsers, etc., the simplest approach is to treat all the categories as mutually dependent on each other. This covers any mutual dependencies that do exist and the ordering of the categories does not then matter. However, this approach can cause efficiency problems. It is often worthwhile avoiding mutual dependencies where possible. To achieve this it is necessary to group categories that are mutually dependent and order these groups so that no category is dependent on one occurring in a later group. The following algorithm achieves this.

The dependencies $D$ of the non-terminal categories are computed, i.e. each production $c$ `::=` $s_1 \; \dots \; s_n$ yields dependencies $(c, s_i)$ for the $s_i$ that are names of categories (rather than explicit terminals). A directed graph $G$ is then constructed with vertices $\{c\}$ for each category $c$, and edges corresponding to the dependencies $D$. So, each vertex is a set of category names, initially of one element. The procedure `reduce_graph` (Figure 3) is applied to $G$ eliminating cycles by merging the vertices of the cycle into a single vertex. Any trivial cycles (from a vertex to itself) are then removed and a topological sort (see Section 5.1.3) is performed on the resulting acyclic graph. This yields a list of sets of category names such that the categories in each set are mutually dependent.

The procedure `reduce_graph` has a local variable $C$ for storing a set of cycles, and a local function and procedure. The local function `non_trivial_cycles`

```
procedure reduce_graph(var G);
var C;

   function non_trivial_cycles(V);
   begin
      if V = {}
      then return {}
      else begin
              choose a vertex v in V;
              if there is a cycle in G of length > 1 containing v
              then return all cycles in G containing v
              else return non_trivial_cycles(V − {v})
           end
   end;

   procedure merge_vertices(V);
   begin
      add a new vertex m = ⋃_{v∈V} v to G;
      for each vertex v in V do
         begin
            for each edge e of the form (u, v) do
               begin add an edge (u, m); delete e end;
            for each edge e of the form (v, w) do
               begin add an edge (m, w); delete e end;
            delete v
         end
   end;

begin
   C := non_trivial_cycles(vertices in G);
   while C ≠ {} do
      begin
         merge_vertices(vertices in C);
         C := non_trivial_cycles(vertices in G)
      end
end;
```

Figure 3: Procedure reduce_graph

looks for non-trivial cycles with a common vertex. It does not return a set of cycles in which some are disjoint.

# 6 An Implementation

A compiler for the Syn language, called ML-Syn, has been implemented in Standard ML [MTH90, Pau91], a higher-order functional programming language that uses strict evaluation. The compiler also generates code in Standard ML. The Standard ML of New Jersey implementation is used because it provides the lexer and parser generating tools ML-Lex and ML-Yacc. These tools generate Standard ML code in much the same way as the Lex and Yacc tools do for the C programming language. In addition, an analogous tool for generating pretty-printers (ML-Pretty) has been implemented. Input for these lower-level tools is generated from the Syn specification and they in turn generate ML.

The ML-Syn implementation is modular. A parser for the Syn language produces a parse tree. There are then several programs for extracting the different aspects of syntax from the Syn specification: one for the abstract syntax, one for the lexical aspects, one for grammars, and one for pretty-printing information. The result in each case is represented as ML data types. These are designed to be free of extraneous information and independent of target tools. The intention is for back-ends for different tools to be implementable, taking these data types as their input. The back-ends generate abstract syntax trees for the target languages and then use a pretty-printer to generate code.

The normal mode of operation is for files to be generated as input to ML-Lex, ML-Yacc, etc. However, the modularity allows files to be bypassed in some cases. For example, the abstract syntax tree generated for ML-Pretty can be passed directly to its internals, bypassing the pretty-printing and reparsing phases.

The use of pretty-printing for code generation allows the output files to be read easily, or even edited if the user so wishes. This suggests two ways of using ML-Syn. One is to only ever edit the Syn specification and have the entire syntactic support generated from it at the cost of working within the constraints of the Syn language. The other approach is to write a rough Syn specification of the language, generate files for ML-Lex, etc., and then use these as source files as if they had been hand-written. The original Syn specification is then discarded. The latter approach allows syntactic support to be rapidly generated while retaining the full expressiveness of the lower-level tools. If the language to be supported is still under development the Syn specification could be used during prototyping and then, once the language has settled, the lower-level specifications could be modified to fine-tune the implementation.

The use of ML-Lex and ML-Yacc places some constraints on the Syn specification. Although Syn supports full context-free syntax, ML-Yacc only supports grammars that are LALR(1). ML-Lex is restrictive because it does not implement lookahead. Users need to bear these constraints in mind when writing a Syn specification, and they may result in ML-Lex and ML-Yacc giving errors on the input provided by ML-Syn. Users therefore need to be familiar with the lower-level tools. Relating an error message back to the original Syn specification is generally not too difficult because ML-Syn uses the same (or similar) names in its output as those appearing in the Syn specification. However, it can sometimes be difficult to see what changes need to be made to the Syn specification in order to make ML-Yacc happy. The automatic inference of precedence is often the culprit.

A particular difficulty with using the ML-Lex/ML-Yacc combination is that token names must be kept consistent between the two. ML-Syn avoids this problem by generating the token names automatically.

ML-Yacc does not directly support optional and repeated syntactic elements. Options are implemented as alternative productions with and without the optional symbols. Repetitions are implemented by introducing new (recursive) syntactic categories that build a list of ASTs. Although this may sound straightforward, the code is quite complex because there may be several options or repetitions in an ML-Syn production or they may even be nested.

# 7 Discussion

The Syn language is unusual in allowing all aspects of concrete syntax to be specified in a single non-redundant formalism. The implicit specification of precedence is especially novel. The cost of this simplicity is loss of expressive power. However, Syn has been used successfully to generate parsers and pretty-printers for a number of realistic computer languages.

The main limitation of Syn seems to be in its inability to handle context-sensitive features such as user-defined infix operators. Nor is it suitable for languages that use formatting information such as indentation to determine block structure and the like. Tools like Yacc are not designed to support such features either, but the availability of a full programming language within the specifications they take as input usually allows such features to be hacked.

In addition to providing some support for context-sensitive features, a future version of Syn might allow references to categories that are not defined in the Syn specification. It would then be possible to specify a language that has some other language(s) as a sub-language.

Inferring precedence from textual ordering is by far the most experimental feature of the Syn language. Although there is positive evidence to support its inclusion, a number of issues remain unresolved. In particular, when a terminal appears more than once in a specification, which occurrence should take priority when assigning a precedence to the terminal?

## Acknowledgements

Thanks are due to Matthias Mutz (Passau), Michael Norrish (Cambridge), Ralf Reetz (Karlsruhe), and Daryl Stewart (Cambridge) for feedback on Syn.

## References

[BCD+88] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. In Henderson [Hen88], pages 14–24.

[BS86] R. Bahlke and G. Snelting. The PSG system: From formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, October 1986.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[Hen88] P. Henderson, editor. *ACM SIGSOFT'88: Third Symposium on Software Development Environments* (*ACM SIGSOFT Software Engineering Notes*, 13(5), and, *ACM SIGPLAN Notices*, 24(2)), Boston, Massachusetts, November 1988.

[HHKR89] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — reference manual —. *ACM SIGPLAN Notices*, 24(11):43–75, November 1989.

[Joh78] S. C. Johnson. YACC — yet another compiler-compiler. Technical report, Bell Laboratories, Murray Hill, NJ, U.S.A., July 1978.

[KLMM83] G. Kahn, B. Lang, B. Mélèse, and E. Morcos. Metal: A formalism to specify formalisms. *Science of Computer Programming*, 3(2):151–188, 1983.

[Les75] M. E. Lesk. Lex — a lexical analyzer generator. Computer Science Technical Report 39, Bell Laboratories, Murray Hill, NJ, U.S.A., October 1975.

[LPRS88] P. Lee, F. Pfenning, G. Rollins, and W. Scherlis. The Ergo support system: An integrated set of tools for prototyping integrated environments. In Henderson [Hen88], pages 25–34.

[MC86] E. Morcos-Chounet and A. Conchon. PPML: A general formalism to specify pretty-printing. In H.-J. Kugler, editor, *Information Processing 86 (Proceedings of IFIP Congress)*, pages 583–590, Dublin, 1986. IFIP, Elsevier Science Publishers B.V. (North-Holland).

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Pau91] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[RT89] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[Rub83] L. F. Rubin. Syntax-directed pretty printing — a first step towards a syntax-directed editor. *IEEE Transactions on Software Engineering*, SE-9(2):119–127, March 1983.

[RW81] G. A. Rose and J. Welsh. Formatted programming languages. *Software — Practice and Experience*, 11:651–669, 1981.

# A A Syn Specification of the Syn Language

```
Syn (specification):

#whitespace = {[\ \t]+};
#newline = {\n};
#comment = #nested_quote("(*","*)");

specification ::=
   (Spec) [<v 0,1> [<h 0> identifier <1> ["(" identifier ")"]? ":"]
                   [<v 0,0> [lexical]*]
                   [category]+];

lexical ::=
   (FlagLex) ["#" identifier ";"]
 | (RegExpLex) [<hv 1,3,0> ["#" identifier <1> "="] [regexp ";"]]
 | (FunctionLex) [<hv 1,3,0> ["#" identifier <1> "="] [function ";"]];

function ::=
   (Function)
      [<hv 0,3,0> ["#" identifier]
                  ["(" [<hv 0,0,0> ([sconst ","]* sconst)]? ")"]];

category ::=
   (Category) [<hv 1,3,0> [<h 1> identifier "::="] [body ";"]];

body ::= (RegExpTermBody) [<h 1> regexpterm [associativity]?]
       | (StringTermsBody) [<hv 1,0,0> stringterm [or_stringterm]*]
       | (AlternativesBody)
           [<hov 1,~2,0> (alternative [<h 1> "|" alternative]* )]
       | (AliasBody) identifier;

regexpterm ::= (RegExpTerm) regexp
             | (FunctionTerm) function
             | (TypedTerm) ["#" terminaltype]
             | (TypedRegExpTerm) [<hv 1,3,0> ["#" terminaltype] regexp]
             | (TypedFunctionTerm)
                 [<hv 1,3,0> ["#" terminaltype] function];

terminaltype ::= identifier;

associativity ::= (LeftAssoc) "(l)" | (RightAssoc) "(r)";

or_stringterm ::= (LessPrec) [<h 1> "|" stringterm]
                | (EqualPrec) [<h 1> "=|=" stringterm];
```

```
stringterm ::= (StringTerm) [<h 1> sconst [associativity]?];

alternative ::=
   (Alternative)
      [<hv 1,3,0> ["(" [identifier]? ")" [preferred]?] format];

preferred ::= (Preferred) "*";

format ::= (StringTermFormat) stringterm
         | (NonTermFormat) identifier
         | (EmptyFormat) ["[" "]"]
         | (BoxFormat)
              ["["
                [<hv 1,3,0> [box_spec]? [<hov 1,0,0> [object]+]]
                "]"
                [modifier]?];

object ::= (Object) [<h 1> [box_params]? format]
         | (ObjectGroup) [<h 1> "(" [<hov 1,0,0> [object]+] ")"];

box_spec ::= (HBoxSpec) ["<h" <1> iconst ">"]
           | (VBoxSpec) ["<v" <1> indent "," iconst ">"]
           | (HVBoxSpec) ["<hv" <1> iconst "," indent "," iconst ">"]
           | (HoVBoxSpec)
                ["<hov" <1> iconst "," indent "," iconst ">"];

box_params ::= (BoxParams) ["<" ([indent ","]* indent) ">"];

indent ::= (AbsoluteIndent) iconst
         | (RelativeIndent) ["++" iconst];

modifier ::= "?" | "*" | "+";

sconst ::= #escape_quote_in_line("\"","\\","\"");

iconst ::= #integer {~?[0-9]+};

identifier ::= {[A-Za-z][A-Za-z0-9_']*};

regexp ::= #nested_quote_in_line("{","}");
```