# **Analysing the SML97 Definition: Lexicalisation**

Elizabeth Scott ⊠®

Department of Computer Science, Royal Holloway, University of London, UK

Adrian Johnstone  $\square$ 

Department of Computer Science, Royal Holloway, University of London, UK

#### Abstract

The specification of the syntax and semantics for Standard ML have been designed to support the generation of a compiler front end, but actual implementations have required significant modification to the specification. Since the specification was written there have been major advances in the development of language analysis systems that can handle general syntax specifications. We are revisiting the SML specification to consider to what extent, using modern tooling, it can be implemented exactly as originally written. In this short paper we focus on the lexical specification.

2012 ACM Subject Classification Software and its engineering  $\rightarrow$  Software notations and tools

Keywords and phrases SML, language specification, lexicalisation, parsing, ambiguity

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.23

### 1 Introduction

In the Definition of Standard ML [8], the syntax and semantics have been formally specified in a way that is designed to support the generation of a compiler. The lexical syntax is specified in prose and defines a set of lexical tokens. The phrase level syntax is specified using a context free grammar, annotated with some comments, whose terminals are these lexical tokens. We refer to this language as full SML and to the corresponding grammar as the fullSML grammar. The 1997 Definition also includes another grammar, bareSML, specifying a smaller language which is almost, but not quite, a subset of full SML. There is a map from full SML to bare SML defined as a set of rewrite rules which are expressed in terms of the structure given by the fullSML grammar. In fact the rules do not always result in a string in full SML, but this is a technical detail that is not important here. Semantics are defined for bare SML in natural semantics style using SOS inference rules written in terms of the bareSML grammar rules.

The salient point here is that we would like to be able to use the 1997 Definition as written as the specification from which a tool that executes semantic evaluation of SML can be directly generated. However, the specification is not in a form that can be directly implemented using classical deterministic approaches. For example, fullSML is ambiguous and some of the specification is in prose or is qualified by textual comments. Actual implementations have required significant, albeit often well documented [9], modification to the structure in the Definition.

Since the Definition was written there have been major advances in the development of systems that can handle general syntax specifications. There are practical general parsers, and our new multi-lexer approach [11] allows the removal of the fixed separation of lexical and phrase level analysis. Our goal is not to review or investigate the modifications required to implement an efficient SML translator; this has been done extensively and the interested reader is encouraged to read Rossberg's discussions [9] related to the development of HaMLet [10] and Kahrs' original analysis [7]. Rather we are revisiting the SML Definition to consider to what extent modern tooling allows the syntax specification to be implemented exactly as originally written. In this paper we focus on the lexical specification.

The traditional separation of lexical and phrase level specification and analysis, as described for example in [1], significantly restricts the lexical specification. An input string of characters has to be lexicalised to a single token string before parsing, but the lexicalisation is not unique. So the lexer makes disambiguation decisions, typically using heuristics such as longest match and token priority. This is a substantial challenge for the 1997 SML Definition which has several different identifier tokens with the same set of lexemes. One approach is to dispense with the lexer/parser divide and specify the language grammar at character level. This allows the syntax analyser to consider all the lexicalisations, but the corresponding context free grammar is highly ambiguous and creates a substantial disambiguation task. We refer to this approach of using a character level grammar with no supplementary lexical disambiguation techniques as pure character level parsing. We have introduced a new form of generalised parser which efficiently parses multiple input strings concurrently [11]. This allows a flexible divide between lexical and syntax level specification, with partial disambiguation carried out at an initial lexical phase. We refer to this as multi-lexer parsing. At the same time as the SML Definition was being developed, the need for practical character level GLR parsing in SDF2 was addressed by Visser [14]. Visser's thesis introduced lexical filters which specify disambiguation rules that are applied variously to the underlying character level LR parse table and at parse time. This is often referred to as scannerless parsing, but we will use the term SGLR parsing to make the distinction with pure character level parsing clear.

In this paper we discuss some of the SML lexical issues and indicate how they can be addressed using multi-lexer parsing. We also compare multi-lexer SML parsing to pure character level parsing and demonstrate that pure character level parsing is not practical. This shows that the alternatives such as multi-lexer parsing and SGLR parsing are fundamentally required. In future work we intend to compare SGLR parsing with GLL multi-lexer parsing as the two techniques achieve their results in quite different ways.

### 2 Lexicalisation

In general we follow the terminology in [1]. For a given a set of characters,  $\mathcal{A}$ , a lexical token denotes a set of strings of characters, the token's pattern, and a character string in the pattern of a token is called a lexeme of that token. A lexicalisation of a character string q is a string,  $t_0 \dots t_m$ , of tokens with the property that there exist  $q_i \in t_i$ ,  $0 \le i \le m$ , such that  $q = q_0 \dots q_m$ .

A context free grammar (CFG) consists of a set,  $\mathbf{T}$ , of terminals, a set,  $\mathbf{N}$ , of nonterminals disjoint from  $\mathbf{T}$ , a start symbol,  $S \in \mathbf{N}$ , and a set of grammar rules  $X := \alpha_1 \mid \ldots \mid \alpha_t$ , one for each nonterminal  $X \in \mathbf{N}$ , where  $\alpha_k \in (\mathbf{T} \cup \mathbf{N})^*$ . We refer to this as a BNF grammar. An EBNF grammar is defined in the same way except that the alternates are regular expressions over  $\mathbf{T} \cup \mathbf{N}$ . It is typical for the terminals of a grammar to be lexical tokens and the terms terminal and token will be used interchangeably.

Classically, the first step in translation is to lexicalise the input character string. We use an end of string character, \$, to avoid having to write special cases for empty strings. For a character string q\$ the set lex(q\$) of all lexicalisations over a token set  $\mathbf{T}$  can be defined as

$$lex(\$) = \{\$\}$$
  $lex(q\$) = \{t_0t_1 \dots t_m \in \mathbf{T}^* \mid \exists q_0 (q = q_0p, q_0 \in t_0, t_1 \dots t_m \in lex(p\$))\}$ 

We can thus assume that lexicalisations are constructed from the left. In effect, for a character string  $a_1 \dots a_d$  we identify lexemes beginning at each index position in increasing order, but we only consider those positions i for which we have already obtained at least one lexicalisation of  $a_1 \dots a_{i-1}$ . Then we have a lexical ambiguity at position i if, for some j > i,

 $a_i 
ldots a_j$  is the lexeme of two different tokens t, s (intersection ambiguity), or if there is also some h > j such that  $a_i 
ldots a_j$  and  $a_i 
ldots a_k$  are both lexemes (prefix ambiguity). The classical resolution is to specify a priority between t and s, and a longest match condition under which h is chosen over j. We consider both longestWithin, which is only applied to lexemes of the same token, and longestAcross, which is applied even when u and uv are lexemes of different tokens. Of course, priority and longest match strategies do not always allow the required disambiguation to be specified. With a multi-lexer parser [11] more than one lexicalisation can be passed to the parser, allowing the parser to make lexical disambiguation decisions.

### 2.1 Multi-lexer parsing

The key idea is that the lexer returns token with extent (TWE) elements, (t, i, j), in which t is the token and i, j are the left and right positions of the corresponding lexeme in the input character string. Construction from the left means, for each i, all tokens (t, i, j) are constructed before tokens of the form (s, i+1, k). Position i > 0 is only considered if some TWE element (t, l, i) already exists. Longest match and priority disambiguation are applied at each position i by comparing right extents, and tokens if the right extents are the same.

In a classical lexer one TWE element, (t, i, j), is selected at position i and the lexicalisation continues from position j. In a fully general multi-lexer approach all the TWE elements are constructed and a post-lexer disambiguator may apply disambiguation rules to remove elements. This allows the possibility that more than one lexicalisation can be retained and subsequently parsed. The MGLL [11] generalised LL style parser can efficiently parse multiple lexicalisations of a character string. We do not discuss the technique here but we have used it for the investigations reported below. A third possibility is a hybrid approach in which some lexical disambiguations are applied "on-the-fly" as in the classical lexer, and others are done post-lexing. The difference between this and a full lexer is that, although (t, i, j) may still be constructed, if it is not retained then the lexer may not subsequently consider position j. A hybrid approach is used in SGLR [14], see Section 7. Our current MGLL parser carries out full lexicalisation, although it can easily be modified to run in hybrid form.

Multiple derivations of a given lexicalisation are efficiently represented as shared packed parse forest (SPPF) [3] which is obtained by merging all the corresponding derivation trees. To extend this to represent the derivations of more than one lexicalisation we use lexical extents in the node labels. The leaves of the tree are labelled with TWE elements, (t, i, j), and an interior node is labelled (A, i, j) where i is the left extent of it left-most child and j is the right extent of its right-most child. These trees are then combined by sharing nodes with the same label, and allowing for different sets of children by creating "packed" nodes as children of (A, i, j) and making each family of children the children of a packed node. Where the SPPF corresponds to just one lexicalisation, phrase level ambiguity corresponds precisely to the existence of nodes with more than one packed node child.

## 3 Aspects of the fullSML grammar

SML program fragments are structured into "phrase classes" that reflect the intended semantics. The fullSML grammar has a nonterminal for each phrase class and grammar rules that specify the sentences of that phrase class. The grammar we use is the same as that in the 1997 SML Definition [8] except that we have converted EBNF optional constructs to BNF and sequences with ellipses, e.g.  $(exp_1, \ldots, exp_n)$ , to recursively defined constructs. For discussion purposes below, part of our grammar is given in Figure 1.

```
\begin{array}{llll} program ::= topdec \; ; \; | \; topdec \; ; \; program \\ topdec ::= strdec \; | \; sigdec \; | \; fundec \\ strdec ::= dec \; | \; \epsilon \; | \; strdec \; strdec \; ; \; strdec \\ dec ::= val \; tyvarseq \; valbind \; | \; \epsilon \; | \; dec \; dec \; | \; dec \; valbind ::= pat = exp \\ tyvarseq ::= tyvar \; | \; \epsilon \; | \; (\; tyList \; ) \\ tyList ::= tyvar \; | \; tyList \; , \; tyvar \\ exp ::= infexp \; | \; if \; exp \; then \; exp \; else \; exp \; | \; case \; exp \; of \; match \; | \; fn \; match \\ match ::= mrule \; | \; mrule \; | \; match \\ mrule ::= pat => exp \\ infexp ::= appexp \; | \; infexp \; vid \; infexp \\ appexp ::= atexp \; | \; appexp \; atexp \\ atexp ::= scon \; | \; longvid \; | \; \{ \; \} \; | \; ( \; ) \; | \; (\; exp \; ) \end{array}
```

**Figure 1** Part of the fullSML grammar.

The lexical classes are specified in the Definition using textual descriptions. In Figure 2 we give a definition using EBNF which includes set difference,  $\setminus$ , (only used to remove finite sets), Kleene ()\* and positive ()+ closure, and ranges, [0-9], [a-z], [A-Z].

```
vid ::= iden
tyvar ::= ' (letter | digit | ')^*
tycon ::= iden \setminus \{*\}
lab ::= iden \mid [1-9] \ digit^*
strid ::= alphaNum
sigid ::= alpha Num
funid ::= alphaNum
longvid ::= (strid.)^* vid
longtycon ::= (strid.)^* tycon
longstrid ::= (strid .)^* strid
scon ::= int \mid word \mid real \mid char \mid string
d := digit
iden ::= idenFull \setminus Rword
idenFull ::= letter (letter | digit | ' | \_)^* | sym^+
alphaNum ::= alphaNumFull \setminus Rword
alphaNumFull ::= letter \ (\ letter \ | \ digit \ | \ ' \ )^*
sym ::= \  \, ! \  \, | \  \, \% \  \, | \  \, \$ \  \, | \  \, \# \  \, | \  \, + \  \, | \  \, - \  \, | \  \, : \  \, | \  \, | \  \, @
                        ~ | \ | ' | ^ | | | *
letter ::= [a - z] \mid [A - Z]
digit ::= [0 - 9]
```

**Figure 2** SML lexical class specification.

### 3.1 Phrase classes

The phrase class Program is the set of all full SML sentences, and there is a corresponding fullSML start nonterminal *program*. An SML program is a sequence of "top level declarations" whose effects are to modify a top level environment. The nonterminal *topdec* generates the phrase class TopDec whose elements are grouped into subclasses StrDec, SigDec and FunDec

(structure level, signature and functor declarations) with corresponding nonterminals strdec, sigdec and fundec. Phrases in StrDec can be concatenated with or without a separating semicolon. Within StrDec we focus on the phrase class Dec, and within Dec we consider the straightforward value declarations, which in turn invoke the expression phrase class Exp. In Figure 1 we have just shown a subset of the constructs, including if and case statements which are not part of the bare SML language and have to be rewritten using the rewrite rules. The nonterminal infexp generates the atomic expressions in the phrase class AtExp of atomic expressions, and also the infix operator syntax. SML allows binding to "patterns" from the phrase class Pat, which are similar in structure to atomic expressions.

As is well documented [9], the nonterminals strdec and dec are ambiguous: sequencing is not forced to be either left or right associative and sequencing of Decs can be done within Dec or StrDec. These ambiguities, in particular the existence of cycles  $strdec \stackrel{+}{\Rightarrow} strdec$  and  $dec \stackrel{+}{\Rightarrow} dec$ , mean that there are infinitely many derivations of any declaration. We have developed an SML-specific cycle removal algorithm which removes specific packed nodes from the SPPF. The remaining ambiguity can be resolved as described in the Definition by choosing alternates by priority. For example choosing strdec ::= dec over strdec ::= strdec strdec, and "longest match" from the left, so generating as much of the input as possible from the left-most nonterminal in an alternate. We use these disambiguation rules, in the form of SPPF "choosers" see Section 6, to remove phrase level ambiguity, allowing lexical ambiguity to be identified in character level SPPFs<sup>1</sup>.

### 3.2 Lexical classes

The terminals of the fullSML grammar include certain reserved words

abstype and andalso as case datatype do else end exception eqtype fn fun functor handle if in infix infixr include let local nonfix of op open orelse raise rec sharing sig signature struct structure then type val with withtype where while

( ) [ ] { } , : ; ... \_ | => -> # :>

We call this set Rword. There is a second small set,  $Tsym = \{ =, *, \sim \}$ , of terminals that have patterns that are singleton sets containing just themselves, but which are allowed to occur in identifiers. The other terminals have patterns that are specified as lexical classes, VId TyVar TyCon Lab StrId SigId FunId SCon D LongVid LongTyCon LongStrId

All of these except SCon and D are sets of identifiers. In the first six classes these identifiers

are either sequences of letters, digits, primes and underscores or symbolic identifiers. TyVar identifiers must begin with a prime. The last three classes are extensions that allow identifiers to be linked to modules. The terminal names are lower case versions of the lexical class names.

We do not give the formal definition of the SCon phrase class in this paper, but note that it includes integers, hexadecimals, reals, strings etc in a standard way. The lexical class D, used in the specification that an operator has infix syntax, is just the set of the digits.

<sup>&</sup>lt;sup>1</sup> The SPPF priority choosers, which are different to the lexical choosers, play a similar role to the SGLR context-free priority specifications [14]. Longest match SPPF choosers allow the disambiguation specification that in the SGLR approach is done using associativity declarations.

### 3.3 The rewrite rules

The particular syntax specification in the Definition is important. The rewrite rules which convert fullSML to bareSML are associated with the phrase classes, for example expressions in Exp are rewritten to other expressions

```
case exp of match 	o 	o 	ext{(fn } match 	ext{)(} exp 	ext{)}
```

We note that the issue of identifier class overlap cannot simply be resolved by merging the classes into a single identifier class and leaving the resolution to post-parse disambiguation. The rewrite rules require the separation of the various identifier lexical classes. For example, the rule

```
vid atpat = expr \rightarrow vid = fn \ vid_1 \ case \ (vid_1) \ of \ (atpat) => expr rewrites elements of the derived phrase class FvalBind of function-value bindings. It should rewrite a phrase of the form vid \ (longvid : longtycon) = longvid, but not one of the form funid \ (strid : sigid) = longstrid, which is part of a functor binding phrase class. These phrases would become indistinguishable if the lexical identifier classes were merged.
```

### 4 SML lexical disambiguation

With any lexical disambiguation there are two potential problems: (1) a local decision means that ultimately the input cannot be lexicalised or (2) a lexical disambiguation removes all the syntactically correct lexicalisations and the parser rejects the input. The former is not common but perhaps surprisingly the latter does occur, for example longestAcross disambiguation causes the rejection of x++y in C-style languages including Java [11]. LongestWithin disambiguation allows a less restrictive approach in which many lexicalisations are removed but some are passed to the parser for resolution.

We consider the lexical classes for SML as actually described in the Definition; so the terminals of the fullSML grammar are the elements of Rword, Tsym and the twelve lexical classes described in Section 3.2.

The Definition states (Section 2.5) that at each lexical stage the longest next item is taken, i.e. longestAcross disambiguation. But this is not always required. The class D and the singleton pattern elements of Rword and Tsym, do not contain lexemes pairs where one is a proper prefix of another, so there is no point in specifying longestWithin disambiguation for these. LongestWithin is specified for the other lexical classes<sup>2</sup>.

TyVar does not have any lexemes which belong to or are prefixes of lexemes of any other class, so no further disambiguation specification is required. It is also not necessary to specify longestAcross disambiguation between the other core identifier classes VId, StrId, FunId, SigId, and TyCon. For example, if we have  $(\mathtt{strid},i,j)$  and  $(\mathtt{vid},i,k)$  where k>j, and  $q\in \mathsf{StrId},\ qp\in \mathsf{VId}$  are the corresponding lexemes, then  $qp\in \mathsf{StrId}$ . So there is a TWE element  $(\mathtt{strid},i,k)$  and longestWithin on  $\mathtt{StrId}$  removes  $(\mathtt{strid},i,j)$ . However, it is possible to have TWE elements  $(\mathtt{vid},i,j)$  and  $(\mathtt{longvid},i,k)$  where  $k>j,\ q\in \mathsf{VId},\ qp\in \mathsf{LongVId}$  but  $qp\notin \mathsf{VId}$ . So we need to specify longestAcross for VId and LongVId.

A full discussion of the degree to which longestAcross is needed is beyond the scope of this paper. In our examples in Section 6 we use full longestAcross, but our multi-lexer parser can handle SML with just longestWithin and indeed with no lexical disambiguation at all.

For TWE sets, post-lexer longestAcross disambiguation can be done at each position i in turn by removing (t, i, j) if there is a (s, i, k) with k > j. We can ensure that at least one lexicalisation remains by first pruning the set of TWE elements to ensure that every

<sup>&</sup>lt;sup>2</sup> LongestWithin for identifier classes can be specified for SGLR parsers using follow restrictions [14], but see the discussion in Section 7.

remaining element belongs to some complete lexicalisation. We refer to this as TWE *dead* branch pruning. LongestAcross disambiguation can also be done during lexicalisation, but this does not easily permit dead branch pruning and we have not done this.

Intersection ambiguity occurs in SML between the nine main identifier classes, and between the classes SCon, Lab and D. The ambiguity cannot be resolved using priority specifications as there is no single class that can be preferred, and the discussion in Section 3.3 shows that we cannot simply merge all the identifiers into a single Id class which is then returned by the lexer.

A partial solution is the so-called Schrödinger's token approach [2] in which the lexer returns a generic id token and the parser substitutes the required actual token using phrase level context information. Since the rewriting needs the token string it would be necessary for the parser to rewrite the input token string, and in any case the more general multi-lexer parsing approach can just parse directly all the options represented by a Schrödinger token.

In fact, Section 2.4 of the SML Definition gives a disambiguation of the classes VId, StrId, TyCon and Lab in terms of syntactic context. Classically this would be implemented via some sort of symbol table that shares information between the lexer and other phases of the translation process. The observation that these classes can be disambiguated at the phrase level means that the multi-lexer approach deals with them in a very simple way, all token choices are passed to the parser and the ambiguity is fully resolved as only the "correct" lexicalisation parses. This also applies to the FunId class which is part of the module specification, addressing the FvalBind rewrite issue discussed in Section 3.3.

There is also intersection ambiguity between the identifier classes and their corresponding long classes. This is not covered by the discussion in Section 2.4 of the Definition, and the ambiguity cannot always be resolved by the parser. The problem is illustrated with the phrase  $if \ x < 1 \ then \ y \ else \ z$ ;. We expect < to be treated as an infix operator but SML also allows it to be a parameter to an application of a prefix operator x. Thus both lexicalisations below belong to the phrase class Exp.

if longvid vid scon then longvid else longvid;

if longvid longvid scon then longvid else longvid;

In fact is not clear from the Definition whether LongVId is intended to be a lexical or phrase class. It would be possible to turn longvid into a fullSML grammar nonterminal and then the lexer will return vid rather than longvid, avoiding the intersection issue. But this results in a phrase level ambiguity which is equivalent allowing the multi-lexer to return both options, and in either case leaves the parser with an ambiguity to resolve.

We note that formally there is no intersection ambiguity in SML between the keywords and the identifier classes because the 97 Definition does not include keywords in the identifier classes. However, for simplicity of specification our SML identifier lexical classes include the reserved words and the corresponding elements are removed from the TWE set before it is passed to the parser<sup>3</sup>.

### 5 Character level parsing

An alternative approach to handling lexical ambiguity is to define the phrase level grammar at character level. Then lexical classes are singleton sets containing just the corresponding character and lexicalisations are trivial and unique. To facilitate our discussion we shall

<sup>&</sup>lt;sup>3</sup> This corresponds to the SGLR approach of specifying reject productions [14], which are written in the form if→ VId {reject}, for keywords such as if.

assume that for each lexical class there is a nonterminal in the character level grammar which derives that class. For SML the lexical classes can be specified using regular expressions over characters, and our GLL parser can handle these directly. We create a character level grammar from fullSML by treating the terminals, vid etc and the reserved words case etc, as nonterminals and adding the rules given in Figure 2. Of course the original lexical ambiguities just become phrase level ambiguities and the problem has been passed to the parser. The parser can construct an SPPF and ambiguities are easily identifiable as nodes that have more than one packed node child. The parser then needs disambiguation rules to decide which of the packed nodes should be retained.

Phrase level ambiguity can be classified as horizontal, two alternates of a nonterminal derive the same string, or vertical, one alternate derives both u and uv [4]. This suggests ambiguity resolution by ordering alternates and applying longest match. This is easy to implement in terms of SPPF packed node removal, and resonates with lexical level priority and longest match disambiguation discussed above. But they are not identical in effect. An interesting question is whether this phrase level disambiguation always gives the same lexicalisation as the one produced using the lexical longest match and priority disambiguation, but this is beyond the scope of this short paper. In fact, the required disambiguation information may not be close in the SPPF to the position of the corresponding multiple packed nodes. For example, there are two syntactically valid derivations of the string val x = bb, one in which bb is derived from a single longvid and one where each b is derived from a separate longvid. These two derivations are generated from the nonterminal appear, which has two packed node children. We want to choose the packed node that has descendent (longvid, 8, 10) rather than the one that has descendants (longvid, 8, 9) and (longvid, 9, 10), but these nodes are not immediate children of the packed nodes concerned and descendent node searching is required.

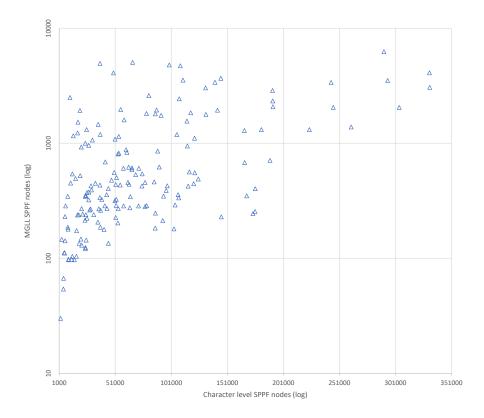
In summary, although, using GLL, character level parsing is worst case cubic, applying lexical disambiguation can be difficult, and the SPPFs produced are much larger than those produced using the multi-lexer parser approach, which permits lexical level disambiguation. In the next section we give some data to support this claim.

### 6 Experimental observations

An ART [6] specification may include EBNF grammar rules, a nominated set of nonterminals called the *paraterminals*<sup>4</sup>, disambiguation declarations called *choosers*, term rewrite rules and attribute equations. Lexical choosers specify TWE set disambiguation as discussed in Section 2.1, and SPPF choosers specify syntax level disambiguation via SPPF packed node removal. The set of paraterminals conceptually splits the specification into an upper phrase-level grammar whose terminals are the paraterminals, and one lower lexical grammar per paraterminal, thus specifying a multi-lexer parser. The specification can also be used to generate a pure character level GLL parser, which we call PCLGLL. The advantage of the multi-lexer approach is that lexical disambiguation can be efficiently applied to the TWE set before parsing. In these experiments we use the paraterminal declarations to generate a multi-lexer parser, PTMGLL, which deploys a GLL recogniser for the paraterminal grammars and constructs longest match and priority disambiguated TWE sets.

<sup>&</sup>lt;sup>4</sup> Paraterminals have a similar role to the SGLR nonterminals A that have an injection of the form  $A-LEX> \rightarrow A-CF>$  [14]

The programming artefacts and full experimental results summarised here are available under art.csle.cs.rhul.ac.uk. Our corpus comprises the SML source files from the MLWorks compiler and interactive environment originally developed by Harlequin, now open sourced github.com/Ravenbrook/mlworks. To avoid the domination of whitespace ambiguity in the character level experiments, the 1,082 files in the flattened source hierarchy were normalised using the !compressWhitespaceSML ART directive which replaces all runs of SML whitespace and comments outside of strings with either a single space or a single newline character if the run encompasses a line end. The resulting corpus is MLWorksSourceFlatCompressed.zip. The ART specification smlFull.art closely follows the Definition, the minor modifications are as mentioned in Section 3, except that we use nonterminals,  $wOp ::= w \mid \epsilon$ , rather than expansion style replacement,  $X ::= \alpha w\beta \mid \alpha\beta$ .



**Figure 3** Relative character and paraterminal SPPF size.

Our first observation is about the practicality of a pure character level parser, in terms of the size of the SPPF, which is independent of any particular parsing technique or implementation style. The scatterplot in Figure 3 shows the number of SPPF nodes using log axes: each point represents one file from our corpus. A linear regression analysis gives a trendline with gradient 0.0109, indicating that paraterminal SPPFs are generally around 100 times smaller than pure character level SPPFs. However this broad trend should be treated with caution as is visually evident from the scatterplot: the  $R^2$  value for the linear trendline is only 0.55, and in fact the observed size ratios in this corpus range from 3.74 to 3733.54. The length of each input file is not a useful guide to the size of resulting character level SPPF since SPPF size to input length ratios range from 4.20 to 465.27. So there is no

straightforward intuition that allows us to judge the size of a character level SPPF from a piece of SML source. One reason for this is that the SML syntax for function application means that in many cases where a keyword or a single longvid is valid, then so is any string of longvids.

In the end, practicality manifests as processing time. Our test system is a DELL XPS 15 9510 laptop with 16GByte of installed memory and an Intel Core i7-11800H eight-core processor running at 2.3GHz under Microsoft Windows 10 Enterprise version 10.0.19042 using Oracle's Java HotSpot(TM) 64-Bit Server VM (build 14.0.2+12-46, mixed mode, sharing). On this high end laptop the character level parser can take many seconds to parse some inputs. A limit of 5 seconds of CPU time for each run results in 631 of the 1081 inputs (58%) timing out under PCLGLL: it is clear that using a pure character level grammar is impractical even on a fast modern laptop. None of the files that were processed in less than five seconds by PCLGLL required more than 20ms for PTMGLL to process them.

It is not straightforward to decide how much of the ambiguity in a character level SPPF is lexical. The phrase level SML grammar is ambiguous, so not all the SPPF ambiguity is lexically based, and as shown in Section 3, our individual specifications of each lexical class are unambiguous. Conversely, as discussed in Section 5, a lexical ambiguity can appear in the SPPF under a syntax level nonterminal. ART phrase level choosers in the PTMGLL specification are used to implement the syntax priorities mentioned in Section 3.1. There are 481 strings in the corpus that also only have a single valid lexicalisation (after lexical disambiguation) and we have specified phrase level choosers that fully disambiguate the derivations of these sentences.

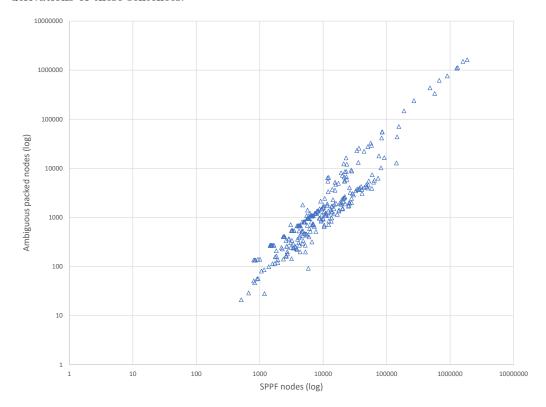


Figure 4 Residual ambiguity.

23:11

Of these 481 strings, 165 are not parsed by the PCLGLL parser in within 5 seconds. For the other 316 strings, applying these choosers in the PCLGLL parser removes all the phrase level ambiguity, and the remaining ambiguity is thus lexical. The scatterplot in Figure 4 shows the number of "ambiguous" packed nodes (i.e. packed nodes with siblings) in the character level SPPF plotted against the total number of nodes. There are 51 strings for which the ambiguous packed nodes comprise greater than 20% of the total SPPF nodes, and in three cases the proportion rises to over 90%.

These observations make concrete the commonly held view that pure character level parsing is not viable; hybrid techniques involving some direct lexical disambiguation are needed.

### 7 Scannerless GLR

At the time of the 1997 SML Definition, tools that could handle general context free specifications were already emerging. ASF+SDF [13] has a GLR [12] based parser, but running this on pure character level grammars in practice triggers the practicality issues illustrated above.

As we have already remarked, Visser addressed the issue in his PhD thesis [14] by adding lexical filters to SDF and developing associated Scannerless GLR parsers. Modifications to SGLR were introduced for use with an RNGLR parser [5] but the approach remains the same. In Visser's thesis the disambiguation heuristics are expressed in terms of derivation selection, but the implementation in an SGLR parser relies heavily on the LR-parsing technique. For example, follow restrictions, associativity and context-free priorities are implemented as modifications to the LR parse table, so comparison with TWE set disambiguation is not straightforward and merits further investigation.

We conclude with the observation that the SML lexical class SCon contains both real numbers such as 1.2 and "words" such as 0w1. So for lexical longest match SCon requires that a real number should not be followed by a period, but a word can be. Thus an SGLR-style follow restriction is not directly available to implement longest match for SCon, the restrictions need to be considered separately for different SCon subclasses. It is likely implementing the SML 97 Definition (as written) using SGLR will generate interesting challenges that Eelco Visser would have relished.

#### - References -

- 1 Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools, 2nd edition*. Addison-Wesley, 2006.
- 2 John Aycock and R. Nigel Horspool. Schrodinger's token. Software: practice and experience, 31:803–814, 2001.
- 3 Sylvie Billot and Bernard Lang. The structure of shared forests in ambiguous parsing. In *Proceedings of the 27th conference on Association for Computational Linguistics*, pages 143–151. Association for Computational Linguistics, 1989.
- 4 Claus Brabrand, Robert Giegerich, and Anders Møller. Analyzing ambiguity of context-free grammars. *Science of Computer Programming*, 75(3):176–191, March 2010. Earlier version in Proc. 12th International Conference on Implementation and Application of Automata, CIAA '07, Springer-Verlag LNCS vol. 4783.
- 5 Giorgios Economopolous, Paul Klint, and Jurgen J. Vinju. Faster scannerless GLR parsering. In *CC'09*, volume 5501 of *Lect. Notes Comput. Sci.*, pages 126–141. Springer, 2009.

### 23:12 Analysing the SML97 Definition: Lexicalisation

- 6 Adrian Johnstone and Elizabeth Scott. Translator generation using ART. In M.van den Brand B.Malloy, S.Staab, editor, *SLE 2010*, volume 6563 of *Lecture Notes in Computer Science*, pages 306–315. Springer-Verlag, 2011.
- 7 Stefan Kahrs. Mistakes and ambiguities in the definition of Standard ML. Technical report, University of Edinburgh, Technical Report ECS-LFCS-93-257, 1994.
- 8 Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, 1997.
- 9 Andreas Rossberg. Defects in the revised definition of Standard ML. https://people.mpi-sws.org/~rossberg/papers/sml-defects-2013-09-18.pdf, 2013.
- Andreas Rossberg. Hamlet; SML reference interpreter. https://people.mpi-sws.org/~rossberg/hamlet/, 2013.
- 11 Elizabeth Scott and Adrian Johnstone. Multiple lexicalisation A Java based case study. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE'19.* ACM, 2019.
- 12 Masaru Tomita. Generalized LR parsing. Kluwer Academic Publishers, 1991.
- M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM TOPLAS*, 24(4):334–368, 2002.
- 14 Eelco Visser. Scannerless generalised-LR parsing. Technical Report P9707, University of Amsterdam, 1997.