LDT: a language definition technique

Adrian Johnstone
Royal Holloway,
University of London
Egham, Surrey,
TW20 0EX, UK
a.johnstone@rhul.ac.uk

Elizabeth Scott
Royal Holloway,
University of London
Egham, Surrey,
TW20 0EX, UK
e.scott@rhul.ac.uk

Mark van den Brand
Technische Universiteit
Eindhoven
Den Dolech 2, Postbus 513,
5600 MB Eindhoven,
Netherlands
M.G.J.v.d.Brand@tue.nl

ABSTRACT
We establish a semantics for building grammars from a modularised specification in which modules are able to delete individual productions from imported nonterminals. Our goal is to allow precise answers to the question: ‘what character level language does this grammar generate’ in the face of difficult issues such as the mutual embedding of languages that have different whitespace and commenting conventions. Our technique is to automatically generate a character level grammar from grammars written at token level in the conventional way; the grammars is constructed from modules each of which may have its own whitespace convention. Modules have import and export lists, and productions may be deleted nonterminals upon import. We conclude with a discussion of a concrete syntax which we use as the input language to our parser generator.

Categories and Subject Descriptors
D.3.1 [Programming languages]: Formal Definitions and Theory; F.4.3 [Formal Languages]: operations on languages

General Terms
Languages, Theory

Keywords
Context Free Grammar, Modularity, Lexer/parser interface

1. INTRODUCTION
The panel discussion at the tenth anniversary LDTA meeting (2010, Paphos) identified a set of challenges for the next wave of grammar based software engineering tools. Amongst these was the need for a unified language description notation and an archive of language descriptions. In a position paper circulated after the meeting we proposed the following.

1. A collaborative effort to create a Language Definition Technique (LDT) with associated facets for the representation of grammar-related specifications such as attribute grammars, rewrite rules, action semantics and so on;
2. The construction and maintenance of an archive of language descriptions extracted from standards documents (and other sources with a reliable provenance) expressed in LDT;
3. The specification of notation and API’s to allow interworking between grammar based tools.

This paper is a contribution to item 1. We present a semantics for our modular grammar scheme called LDT, and a concrete example of a syntax which supports LDT and which is used as input to our ART GLL parser generator [6].

In software engineering, the motivations for modularisation of source code include the following, each of which is also applicable to the engineering of large grammars: (a) the decomposition of a large application into small subsections that can be worked on separately without disturbing the contents, and behaviour, of other modules; (b) the facilitation of incremental builds, in which only the changed module and modules dependent on those changes need to be rebuilt; (c) information hiding (abstraction) that allows internal details to be suppressed as an aid to comprehension; (d) namespace management to allow the same identifier to be used for different purposes in different modules; and (e) reuse of pre-existing fragments.

The primary motivation for this work comes from Klint, Lämmel and Verhoef’s position paper [7], and from discussions with users of SDF’s modularity mechanism [4, 11]. We have also corresponded with colleagues in the Natural Language Parsing community for whom modularity has become a central concern both in terms of managing large grammars and as part of the underlying semantics of natural language [12].

We have limited our scope to grammar specification for conventional programming language translator generation but carefully separated out the formal semantics of our approach from the concrete syntax used in our particular tools in the hope that the general approach will be found broadly applicable. We found that the three most significant issues for traditional tooling were (i) how to allow users to modify existing grammars in a way that removes some elements of the generated language; (ii) how to combine grammar modules in this way without ‘accidentally’ yielding grammars which were not context free (since context free languages...
are not closed under set difference) (iii) how to provide a general mechanism that allows users to build both token level and character level tools in a way that seems natural to users of existing tools.

This paper is just about establishing a semantics, and describing a possible syntax, for the modular composition of grammars. We do not discuss the various techniques for parsing and lexing or the disambiguation of multiple derivations. We shall discuss our approach to these issues in future publications.

2. MODULARISED GRAMMAR SPECIFICATION

In LDT rules are written into one or more modules which together form a modularised grammar specification (MGS). A valid MGS defines an associated CFG formed by composing productions under the control of import expressions and deleters which may be used to suppress productions on import.

In many programming languages it is conventional to allow whitespace to be inserted between some language elements so that strings which vary only in their whitespace are equivalent. It is burdensome to specify these whitespace conventions in full: for instance every keyword instance are equivalent. It is burdensome to specify these whitespaces so that strings which vary only in their whitespace to be inserted between some language elements.

The main goals of LDT are (i) to support modularity and reuse within grammar specifications, (ii) to allow users to write grammars at token level and for the associated character-level grammar (including whitespace and comment handling to be automatically generated, (iii) to specify a general (Shared Packed Parse Forest) SPPF format that natively supports EBNF and (iv) to specify disambiguation semantics. In this paper we discuss our mechanisms to support (i) and (ii): aspects of parsing, derivations and disambiguation will be described in a later paper.

Context Free Grammars

A Context Free Grammar (CFG) is a 4-tuple \((N, T, P, S)\) where \(N\) is a finite set of nonterminal symbols, \(T\) is a finite set of terminal symbols with \(T \cap N = \emptyset\), \(P\) is a finite set of productions of the form \(A ::= \alpha\) where \(\alpha \in (N \cup T)^*\), \(\alpha\) is an alternate of \(A\), and \(S\) is the start symbol.

The relation \(\Rightarrow\) is defined as follows: if \(\alpha\beta\gamma \in (N \cup T)^*\) and \(B ::= \beta \in P\), then \(\alpha\beta\gamma \Rightarrow \alpha\beta\gamma\). We write \(\Rightarrow^+\) for the transitive closure of \(\Rightarrow\), and \(\Rightarrow^*\) for the reflexive and transitive closure of \(\Rightarrow\). We say that \(\alpha\) is a sentential form of the grammar, \(\Gamma\), if \(S \Rightarrow^+ \alpha\) and the language of \(\Gamma\), \(L(\Gamma)\), is defined as \(L(\Gamma) = \{ w \in T^* | S \Rightarrow^* w \}\). We say that \(x \in (N \cup T)\) is reachable if \(x\) appears in some sentential form of \(\Gamma\).

We note that there may be two or more different productions with the same left hand side and right hand side, often referred to as 'repeated rules'. As is conventional, where no confusion results we shall elide this issue and refer to a rule by its left and right hand sides, \(A ::= \alpha\) Formally we assert that each alternate has an associated integer whose sole purpose is to allow the identification of the production. A grammar production is uniquely identified by this number and the nonterminal on its left hand side. Where we are required to distinguish alternates we shall use the notation \(A ::= \alpha, n\) to indicate the number, \(n\), of the alternate. When we come to build grammars from modularised specifications it will turn out to be useful to permit two grammar productions with different left hand sides to have alternates with the same associated integer, that is \(A ::= \alpha, n\) and \(B ::= \alpha, n\). Thus formally a production is identified by the integer and its left hand side, not just the integer.

Modularised Grammar Specifications

A Modularised Grammar Specification (MGS) is a 9-tuple \((M, N, W, T, I, E, D, P, S)\) in which the sets \(M, N, T, I, D,\) and \(P\) are pairwise disjoint and

1. \(M\) is a finite set of module names,
2. \(N\) is a finite set of nonterminal names, expressed as a partition, or pairwise disjoint union, of \(|M|\) subsets, \(\cup\{N_K | K \in M\}\)
3. \(W \subseteq N\) is a set of whitespace conventions, where, for each \(K \in M\), \(W_K = W \cap N_K\) contains at most one element, written \(W_K\) if it exists,
4. \(T\) is a finite set of terminal names,
5. \(E \subseteq N\) is a finite set of export expressions,
6. \(I\) is a finite set of import expressions of the form \(X \leftarrow Y\), where \(X \in N_K\) and \(Y \in E \setminus N_K\), for some \(K\),
7. \(D\) is a finite set of deleters of the form \(X \not\rightarrow \delta\), where \(X \in (N \setminus W)\) and for some \(K \in M\), \(\delta \in ((N_K \setminus W) \cup T)^*\),
8. \(P\) is a finite set of module productions of the form \(X \rightarrow \delta\), where, for some \(K \in M\), \(X \in (N_K \setminus W)\) and \(\delta \in ((N_K \setminus W) \cup T)^*\), \(\delta\) is an alternate of \(X\),
9. \(S \in N\) is the start symbol.

We say that \(A \in N\) is nullable if \(A \rightarrow \epsilon \in P\) or if \(A \rightarrow x_1 \ldots x_n \in P\) where each \(x_i\) is nullable, \(1 \leq i \leq n\).

We note that, as for grammar productions, there may be two or more different module productions with the same left hand and right hand side. Again, formally there is an integer associated with each alternate and a production is identified by this integer and its left hand side nonterminal. However, where no confusion results we shall elide this issue and refer to a rule by its left and right hand sides, \(A \rightarrow \alpha\) rather than as \(A \rightarrow \alpha, n\). A tool implementing LDT semantics can automatically generate these numbers and they need never be visible to the user.

3. THE RAW CFG INDUCED BY AN MGS

In this section we discuss the building of a CFG from an MGS. We call this grammar the raw grammar because it takes no account of whitespace and comment layout considerations, i.e. it ignores the set \(W\). As we discussed above, we do not wish to formally exclude repeated rules and in the
following discussion we shall use the integers associated with
alternates when referring to module and grammar productions.
the only purpose of this is to ensure repeated rules in the
MGS induce corresponding repeated rules in the CFG.
Thus, where repeated rules are not required, the reader can
safely ignore the integers and read \( A \to \alpha \) for \( A \to \alpha, \alpha. \n\)
**Definition** An MGS \( \Sigma = (M, N, W, T, I, E, D, P, S) \),
induces a raw CFG \( T_{\Sigma}(X) = (N, T, Q, S) \) where \( Q \) is defined
as follows.

First define \( Q' \) to be the smallest set of productions for which
1. if \( X \to \alpha, \alpha, \alpha, \alpha \in P \) then \( X \to \alpha, \alpha \in Q' \),
2. if \( Y \to X \in I, X \to \alpha, \alpha, \alpha \in Q' \), and \( Y \to \alpha, \alpha, \alpha \notin D \) then
   \( Y \to \alpha, \alpha, \alpha \in Q' \),
The set \( Q \) is obtained from \( Q' \) by removing all productions of the form
\( Y \to \alpha, \alpha, \alpha \), where \( Y \) is not reachable from \( S \) and \( Y \) is not in the same
partition of \( N \) as \( S \).

The production set \( Q \) may be constructed using the fol-
lowing algorithm.

1. **Initialisation**
   - Set \( Q' \) and \( Q \) to \( \emptyset \).
   - For each element \( Y \to \alpha, \alpha, \alpha \in P \)
     - Add an element \( Y \to \alpha, \alpha \to R \).
2. **Resolution**
   - Repeat
     - Remove \( X \to \alpha, \alpha, \alpha \) from \( R \) and add it to \( Q' \)
     - For each \( Y \to X \in I \)
       - If \( Y \to \alpha, \alpha, \alpha \notin D \) and \( Y \to \alpha, \alpha, \alpha \in Q' \)
         - Add \( Y \to \alpha, \alpha \to R \)
   - Until \( R = \emptyset \).
3. **Reaching**
   - Suppose \( S \in N_L \).
   - For each \( Y \to \alpha, \alpha, \alpha \in Q' \)
     - If \( Y \) is reachable or if \( Y \in N_L \)
       - Add \( Y \to \alpha, \alpha \to Q \)

**4. AUTOMATIC WHITESPACE INSERTION**

Compilers often include an initial lexical phase in which an
input sequence of characters is transformed into a sequence
of grammar terminals. Each terminal \( t \in T \) has an associ-
ated set of strings of characters, its *pattern*. The strings in
the pattern of a terminal are called its *lexemes*. Typically
the character set is a standard character set such as ASCII
or Unicode but in principle it could be, for instance, a collec-
tion of pixels, a sequence of mouse actions, or some output
from a voice recognition system.

The patterns are usually regular languages over the set of
characters and thus are commonly specified using regular
expressions. For some applications this separation between
the character strings and the grammar is too restrictive. The
core problem is the situation in which the patterns of two
different terminals have lexemes in common. For example,
in a language which is the union of two languages such as
COBOL and SQL, a lexeme may belong to a keyword termi-
nal in one part of the grammar and the identifier terminal in
the other. One solution is to do away with the separate lexical
phase, make the character set the terminals of the grammar
and use grammar productions to specify the patterns. The
terminals of the new grammar become nonterminals in
the new grammar, which we shall refer to as a character
level grammar. This is the approach taken in SDF [10] and
a detailed discussion can be found in [11].

When a separate lexical phase is employed, it is used to
resolve lexical ambiguities and to remove layout and com-
ments. A character level grammar thus has to include whites-
pace characters in its terminal set, and the positions in the
grammar productions where whitespace is allowed have to
have an instance of a nonterminal which generates the cor-
responding whitespace language. Adding these instances
by hand is tedious and makes the grammar hard to read.
Our MGS includes an automatic whitespace insertion mech-
anism, specified by the elements of the set \( W \) of whitespace
conventions.

Automatic whitespace insertion is the insertion of non-
terminals \( W_M \in W \) and thus is carried out in exactly the
cases where the whitespace convention \( W_M \) is nonempty.
The assumption is that \( W_M \) derives the desired language of
whitespace elements. The instances of \( W_M \) are left out of the
MGS and automatically inserted into the grammar during
the resolution process. In general we want to allow optional
whitespace, thus, if \( W_M \) is not nullable, we create a new
nonterminal \( W_M' \), whose productions are \( W_M' \to \epsilon | W_M \)
and insert \( W_M' \).

The main problem with a na\"ıve automatic \( W \) insertion
process is that it can generate significant ambiguity in the
grammar. (The examples below use the optional whitespace
but similar examples based on non-optional whitespace can
also be constructed.)

**4.1 Modules with no imports**

First we study the case of a module \( M \) with no imports
except to \( W_M \), and we consider two possibilities:

(a) \( W_M \) is inserted between each grammar symbol, so
\( A ::= x_1 W_M x_2 \ldots W_M x_d \) and a new augmented start
rule \( S' ::= W_M' \) \( S \to W_M \) is created.

(b) \( W_M' \) is inserted after each terminal, e.g. so
\( A ::= a W_M' A b W_M' \) and a new augmented start rule
\( S' ::= W_M' S \) is created.

To analyse these proposals we shall assume that
\( W_M ::= v | Zv, \) where \( Z ::= v | Zv \) and \( v \) denotes a
whitespace token.

Consider the grammar specification \( S ::= b A c, \)
\( A ::= \epsilon | a \) and the input string \( bevc \). Under proposal (a)
the grammar is rewritten to

\[
\begin{align*}
S & ::= b W_M' A W_M' c \quad A ::= \epsilon | a W_M' \\
W_M & ::= v | Zv \\
Z & ::= v | Zv \\
W_M' & ::= \epsilon | W_M
\end{align*}
\]

then we have two distinct left-most derivations

\[
\begin{align*}
S \Rightarrow & b W_M' A W_M' c \Rightarrow b W_M A W_M c \\
& \Rightarrow b v W_M' c \Rightarrow b v c \\
S \Rightarrow & b W_M A W_M c \Rightarrow b W_M c \Rightarrow b W_M c \\
& \Rightarrow b v c
\end{align*}
\]

Under proposal (b) the productions for \( S \) and \( A \) are rewrit-
to

\[
\begin{align*}
S & ::= b W_M' A c W_M' \quad A ::= \epsilon | a W_M' \\
& \text{with only one left-most derivation} \\
S \Rightarrow & b W_M' A c W_M' \Rightarrow b W_M A c W_M' \Rightarrow b v A c W_M' \\
& \Rightarrow b v c W_M' \Rightarrow b v c
\end{align*}
\]

4.2 Extending to imported alternates

We now consider the situation in which a nonterminal \( A \) in \( M \) imports an alternate \( \alpha \) from another module \( K \). In this section we look at the action that needs to be taken with respect to \( W_M \). We consider \( W_K \) in the next section.

If case (a) in the previous section is used then there is no additional action required to deal with \( W_M \).

If case (b) is used then the whitespace, \( W_M \), of \( M \) will not, and should not, be attached to the terminals in \( \alpha \). We consider two possibilities

1. Insert \( W_M' \) after each instance of \( A \).
2. Insert \( W_M' \) at the end of the imported alternate.

The first possibility, (1), can lead to unnecessary ambiguity if \( A \) also has an alternate from \( M \). Consider again the grammar specification \( S ::= b A c, \quad A ::= \epsilon | a \) and suppose that \( d \) is imported from \( K \) as a new alternate for \( A \).

For proposal (b)(1) the productions for \( S \) and \( A \) are rewritten to

\[
S ::= b W_M' A W_M A W_M' \quad A ::= \epsilon | a W_M | d
\]

then for \( b c \) we have two distinct left-most derivations

\[
S \Rightarrow b W_M' A W_M A W_M' \Rightarrow b v W_M c W_M' \Rightarrow b v W_M c W_M \Rightarrow b v c W_M'
\]

\[
S \Rightarrow b W_M' A W_M A W_M' \Rightarrow b A W_M' c W_M' \Rightarrow b v W_M c W_M \Rightarrow b v c W_M'
\]

Proposal (b)(2) can also lead to ambiguities in some cases. Consider the grammar specification \( S ::= b A c, \quad A ::= \epsilon \) suppose that \( B \) is imported from \( K \) as a new alternate for \( A \) and that \( B ::= \epsilon | d \). For proposal (b)(2) the productions for \( S, A \) and \( B \) are rewritten to

\[
S ::= b W_M' A c W_M \quad A ::= a W_M' | B W_M' \quad B ::= \epsilon | d
\]

Again there is more than one left-most derivation of \( b c \).

4.3 Importing from modules with nonempty \( W_K \)

If an alternate \( \beta \) is imported by \( A \in N_M \) from a module \( K \) which has a default whitespace nonterminal \( W_K \) we need to ensure that whitespace from \( K \) can surround all the terminals generated from \( \beta \). For this we need to permit \( K \)-whitespace to the left of instances of \( A \).

If \( W_K \) and \( W_K' \) import the same nonterminal then there is nothing to do. Thus we suppose that the rule for \( W_K \) imports \( G ::= u | G u \) and we shall assume that the module \( K \) has whitespace default convention (b)(2) as this introduces less ambiguity than (b)(1). We consider two possibilities.

(i) Insert \( W_K' \) before each instance of \( A \).

(ii) Insert \( W_K' \) at the start of the imported alternate.

The first possibility can create an error in the case where the nonterminal \( A \) also has an alternate, \( \alpha \) say, from \( M \).

Consider the grammar specification \( S ::= b A c, \quad A ::= a \) suppose that \( B b \) is imported from \( K \) as a new alternate for \( A \) and that \( B ::= \epsilon | d \). Suppose that \( M \) has default whitespace behaviour (b)(2).

Using (i) the productions for \( S, A \) and \( B \) would be rewritten to

\[
S ::= W_K' A c W_M' \quad A ::= a W_M' | B b W_K' W_M'
\]

\[
B ::= \epsilon | d W_K'
\]

For this grammar we have

\[
S \Rightarrow v u a c
\]

hence permitting \( K \)-whitespace in an \( M \) string whose derivation has not involved any of the imported alternates.

Using (ii) the productions for \( S \) and \( A \) would be rewritten to

\[
S ::= a W_M' \quad A ::= a W_M' | W_K' B b W_K' W_M'
\]

which does not permit the above derivation. However, there may still be some whitespace ambiguity. For example, if there were also a rule \( S ::= a \) then the string \( bab \) would have two left-most derivations.

\[
S \Rightarrow A A W_M' \Rightarrow b W_K W_M A W_M' \Rightarrow b W_K W_M' A W_M' \Rightarrow b u W_M' A W_M' \Rightarrow b u b W_K W_M' \Rightarrow b u b W_K
\]

\[
S \Rightarrow A A W_M' \Rightarrow b W_K W_M A W_M' \Rightarrow b W_M' A W_M' \Rightarrow b W_K B b W_K W_M' \Rightarrow b u b W_K W_M' \Rightarrow b u b W_K
\]

Further ambiguity will arise from the rule

\[
A ::= W_K' B b W_K' W_M' \quad \text{if the whitespace conventions for } M \text{ and } K \text{ overlap. For example, if both contain the whitespace character } y,
\]

\[
Z ::= v | y | Z v | Z y \quad G ::= u | y | G u | G y
\]

Then there are two left-most derivations of the string \( byc \).

Of course, none of this ambiguity makes the grammar incorrect, but it does mean that either an on-the-fly or post-parse disambiguator will be needed.

In order to minimise ambiguity, we shall use b(2)(ii) as the basis for the grammar induced by an MGS, defined in the next section.

4.4 The CFG induced by an MGS when \( W \neq \emptyset \)

Because rules will be rewritten with inserted elements from \( W \), cyclic import dependencies will result in the lengths of alternates increasing without limit. Thus we place a non-cyclic condition on the MGS.

We say that \( W_M \) and \( W_K \) are equivalent, written \( W_M \equiv W_K \), if there is an import \( W_M \leftarrow Z \) in \( I \) if and only if there is an import \( W_K \leftarrow Z \) in \( I \). Otherwise \( W_M \not\equiv W_K \).

An MGS, \((M, N, W, T, I, E, D, P, S)\), displays whitespace sensitive cyclic module dependency if there is a sequence of imports \( X_i \leftarrow X_{i+1} \), \( 1 \leq i \leq n \), in \( I \) such that \( X_i \in N_K, K_1 = K_{n+1} \) and for some \( i, W_N K_i \not\equiv W_{N K_{i+1}} \).

For each \( W_M \in W \) if \( W_M \) is not nullable define a nonterminal \( \hat{W}_M \) whose productions are \( \hat{W}_M ::= \epsilon | W_M \) otherwise let \( \hat{W}_M = W_M \). Let \( W' = \{ \hat{W}_M | W_M \in W \} \). For a string \( \rho \in (T \cup N \cup W') \) define \( \overline{\rho} \) to be the string in \( (T \cup N)^* \) obtained by removing all instances of the nonterminals \( W'_M \) from \( \rho \).

For a string \( \rho \in (T \cup N)^* \) and \( K \) such that \( W_K \not\equiv \emptyset \), define \( \rho(L, K) \) to be the string in \( (T \cup N)^* \) obtained by inserting an instance of \( W_K \), after each instance of each terminal in \( \rho \). If \( W_K = \emptyset \), \( \rho(L, K) = \rho \).

Definition A MGS, \( \Sigma = (M, N, W, T, I, E, D, P, S) \), which does not display whitespace sensitive cyclic modular dependency induces a whitespace expanded CFG \( T_{\Sigma'}(X) = (N', T, Q, S') \) where \( N' = N \cup W' \cup \{ S' \} \) and \( Q \) and \( S' \) are defined as follows.

First define \( Q' \) to be the smallest set of productions for which
1. If \( X \to \rho, n \in P \) and \( X \in N_K \), then \( X := \rho_K, n \in Q' \).

2. If \( Y \to X \in I, X := \rho, n \in Q', Y \in N_M, X \in N_K \) and \( Y \to \not\in D \) then
   (a) \( Y := \rho, n \in Q' \) if \( W_M \equiv W_N \).
   (b) \( Y := A \rho B, n \in Q' \) if \( W_m \not\subseteq W_N \), where \( A = \epsilon \)
   if \( W_K = \emptyset \) and \( A = W_K \) otherwise, and \( B = \epsilon \) if
   \( W_M = \emptyset \) and \( B = W_M' \) otherwise.

The set \( Q \) is obtained from \( Q' \) by removing all productions of the form \( Y := \rho, n \in W \) where \( Y \) is not reachable from \( S \) and \( Y \) is not in the same partition, \( N_L \), say, of \( N \) as \( S \), adding \( W_M' := \epsilon | W_M \) to \( Q \). For each non-nullable \( W_M \in W \) and, finally, \( \not\in \emptyset \), adding a new production \( S' := W_L S \), where \( S' \) is a new nonterminal. If \( W_L = \emptyset \) then we define \( S' = S \).

We can give an algorithm for computing \( Q \) as follows.

1. **Initialisation and whitespace after terminals**
   - Set \( Q' = Q = \emptyset \)
   - For each \( W_M \in W \)
     - If \( W_M \) is not nullable
       - Add \( W_M := \epsilon | W_M \) to \( Q \)
   - For each \( Y \to \rho, n \in P \)
     - Add \( Y := \rho, n \) to \( R \), where \( Y \in N_M \)

2. **Resolution**
   - Repeat
     - Remove \( X := \tau, n \) from \( R \) and add it to \( Q' \)
     - For each \( Y \to X \in I \) such that \( Y \to \tau \in D \)
       - Suppose that \( Y \in N_M \) and \( X \in N_K \)
         - If \( W_K \equiv W_M \) and \( Y := \tau, n \not\in Q' \)
           - Add \( Y := \tau, n \) to \( R \)
         - Else if \( W_K = \emptyset \) and \( Y := \tau W_M', n \not\in Q' \)
           - Add \( Y := \tau W_M', n \) to \( R \)
         - Else if \( W_K = \emptyset \) and \( Y := W_K \tau, n \not\in Q' \)
           - Add \( Y := W_K \tau, n \) to \( R \)
         - Else if \( Y := W_K \tau W_M, n \not\in Q' \)
           - Add \( Y := W_K \tau W_M, n \) to \( R \)
   - Until \( R = \emptyset \)

3. **Reaching**
   - Suppose \( S \in N_L \)
   - For each \( Y := \tau, n \in Q' \)
     - If \( Y \) is reachable or \( Y \in N_L \)
       - Add \( Y := \tau, n \) to \( Q \)

4. **Initial whitespace**
   - If \( W_L \not\in \emptyset \)
     - Let \( S' \) be a new nonterminal not in \( N \) or \( W' \cup U \cup M \cup I \cup P \)
     - Add \( S' := W_L S \) to \( Q \)
   - Else set \( S' = S \)

5. **A Concrete Syntax for LDT**

We have established formal mechanisms for composing Context Free Grammars from Modular Grammar Specifications which automatically perform whitespace insertion, but we have not so far spelt out details of the concrete syntax.

In this section we describe the two versions of concrete syntax supported by ART, a parser generator that implements the LDT modularity semantics and outputs parsers in the GLL style [9].

In ART module names are alphanumeric identifiers; each module name must be unique (that is, a module description cannot be split into two parts). Full nonterminal names are pairs, written \( L.X \), where \( L \) is a module name and \( X \) is another alphanumeric identifier. Within a module \( L \) the nonterminals can be referred to by their right hand part, \( X \), alone. This form of nonterminal naming ensures that the nonterminals form a partition, so \( N_L \) is the set of all nonterminals of the form \( L.X \). The representation of terminals is slightly different in the two different concrete syntaxes supported by ART. This will be discussed later, but in both versions backquoted characters of the form `a indicate terminals.

An ART specification is a (possibly) empty sequence of modules. Module headers provide parenthesised import and export lists with an optional whitespace nonterminal, and are then followed by a (possibly empty) list of module productions and deleters.

In detail, in the concrete syntax import expressions of the module \( L \) are written \( \mathcal{K}.k=1 \) to represent an import of the form \( L.k \leftarrow K.k \). As a shorthand, a concrete import expression may be a module name, \( K \), representing the set of all imports of the form \( L.k \leftarrow K.k \) where \( k \) is exported from \( K \). Export expressions are just the names of the exported nonterminals. In the concrete syntax, we use the := symbol to represent \( \rightarrow \) and := to represent \( \not\rightarrow \).

This small example contains two modules. Module \( \text{imp} \) exports nonterminals \( \text{imp}.X \) and \( \text{imp}.Z \). Module \( \text{mod} \) imports only \( \text{imp}.Z \). In this application \( \text{imp}.X \) is not used. We use renaming to add the alternates of \( \text{imp}.X \) to \( \text{mod}.Y \) and we use a deleter to suppress the import of production \( \text{mod}.Y \to \cdot \).

After reaching analysis has been applied, the resulting CFG is as shown on the right.

```
\[
\text{mod}(\text{imp}.Z=Y)(S) ;
\text{mod}.S ::= 'a \mod.A \cdot 'c;
\text{mod}.A ::= # | 'a \mod.Y
\text{mod}.Y ::= 'd | \text{imp}.Z \cdot 'd \mod.g
\text{imp.}(X Z)
\text{X} ::= 'a \mod.Z \cdot 'a Z
\text{Z} ::= 'd \mod.Z \cdot 'd \mod.g
\]
```

5.1 Terminals in ART

There is an uncomfortable relationship between the formal notion of a context free grammar terminal and the terminals defined in typical programming language standards documents because of some conflation of the notions of a terminal and the lexemes in its pattern. It is clear that most of the time programming language designers think in terms of streams of terminals abstracted from the concrete strings of characters that make up the input to a translator, but when developing lexical level processors they must think at the level of individual characters. In particular, the LR(1) and LL(1) grammar properties of, say, ANSI-C are established over the terminals, not the lexemes, of the language, and in the case of the use of *typedef* to define new names for types, some trickery is required to ensure that ambiguities are abstracted away within the lexer-parser interface.

To support both classical terminal level grammars and character level grammars, ART has two versions of its syntax, accessed via a command line switch. In the terminal level version terminals can be denoted by both backquoted characters, `a`, and by singly quoted character strings *aterminal1*. This syntax is straightforward and needs no further discussion.

In the default, character level, version of the syntax termi-
In detail ART interprets 'aterminal' as an abbreviation for the nonterminal artTerm.aterminal; it creates a production of the form

aterminal ::= 'a' 't' 'e' 'r' 'm' 'i' 'n' 'a' 'l'

in artTerm and adds aterminal to its export list. Finally, ART adds an implicit import of artTerm.aterminal to each module that contains an instance of the terminal 'aterminal'. In this way, the exported nonterminals from artTerm form the names of the terminals in the corresponding terminal level syntax.

5.2 The artSystem built-in module

A detailed character level specification of a language is necessarily somewhat system-specific: our basic character code might be ASCII or Unicode or some more arcane code depending on both host architecture and programming language, and text file conventions especially those concerning line ends vary between operating systems. So as to ease portability, each ART implementation provides a built-in module called artSystem which exports nonterminals anyChar and newline.

The built-in rule for anyChar is constructed automatically by ART as an alternation over every character in the host character set, so an instance of anyChar will consume one character from the input (including new lines). The main application is the construction of rules which match comments and strings such as lex.stringTerminal which will be described in next section. The technique is to import anyChar as a universal match-all rule and then use delimiters to locally adjust the match set so as to exclude delimiters.

5.3 A syntax for ART

In the rest of this section we show a simplified syntax for ART\(^1\). There is no reason in principle, of course, why existing tools should not be extended using the LDT modularity semantics in their own syntactic style and that is why we have presented the formal semantics independently of our implementation.

\(^1\)ART supports both EBNF and TIFF formalism described in [5], but for reasons of compactness we have elided those parts here.

We present ART’s syntax via four modules: main, lex which contains rules for lexical elements, ws which contains the whitespace convention used by the main module, and basicChar which defines the alphabetic and numeric characters.

We make use of the artSystem module in the definitions for module lex and ws so as to construct nonterminals that match singly quoted strings and comments. For example, lex imports (most of) the productions for artSystem.anyChar to local nonterminal lex.notQuote, but since the deleter notQuote \(`\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\\
5.4 Language embedding and whitespace

In Section 4 we developed whitespace insertion conventions which allow correct embedding of languages (including their comment and whitespace rules) whilst at the same time reducing the amount of ambiguity in the grammar compared with, say, simply inserting a call to the language's whitespace sub-language after every terminal and nonterminal instance. In this section we give a concrete example of such an embedding.

Consider the embedding of, say, C programs within Pascal programs. Their whitespace conventions are similar, but comments on Pascal are delimited by /* */ and in C by /* . . . */. We shall model this situation by defining a languages p, e, and c. The p language has one string qq with a whitespace convention that includes a single space, a single newline or a comment delimited by braces. The c language is the set \{ d, bb, a \} with a similar whitespace convention to p except that comments are delimited by square brackets. Language e is the set of strings obtained by concatenating strings from p, a subset of c and then p, in that order, with a whitespace convention that has no comment form. The ART concrete MGS is shown here on the left, and the induced CFG on the right. Unreachable productions are parenthesised, and we use a postfix ? operator to represent optional nonterminal instances, that is ws.wp? is shorthand for (ws.wp | #).

\[
\begin{align*}
\text{e}(p \text{ ws } \text{ we}(S)) & \quad \{ \text{ ws } \text{ we} \rightarrow \}' \} \\
S & \quad \{ \text{ ws } \text{ we} \rightarrow \text{artSystem.newLine } \} \\
B & \quad \{ p \} \\
p & \quad \{ \text{ ws } \text{ wp} \rightarrow \text{artSystem.newLine } \} \\
\text{A} & \quad \{ \text{ q } \} \\
c & \quad \{ \text{ ws } \text{ wc}(B) \} \\
B & \quad \{ d \} \\
\text{we} & \quad \{ \text{ artSystem.newLine } \} \\
\text{wc} & \quad \{ 'C' \} \\
\text{ws} & \quad \{ \text{ artSystem.newLine } \} \\
\text{wp} & \quad \{ \text{ artSystem.newLine } \} \\
\text{w} & \quad \{ \text{ b } \} \\
\end{align*}
\]

The module e imports p and c, but deletes the production B := 'a'; hence in the output there are only two productions for e.B. ART automatically inserts the appropriate optional whitespace after string terminals in rules from modules p and c, and wraps imported rules in module e with optional whitespace rules so as to allow resynchronisation of the whitespace conventions. Finally, the production for start symbol e.S is also prefixed with an instance of e.ws.

6. RELATED WORK

In this section we look at the modularity features of some well known grammar based tools.

The LL parser generator ANTLR [8] does not support modularity at the grammar level although if the target language is Java it is possible to import other Java modules; this mechanism is not powerful enough to facilitate grammar modularity. Since LL is not a general parsing technique true modularity would in any case be difficult to sustain.

TXL [2] offers a very primitive grammar composition mechanism based on the standard include mechanism. It is possible to split up the grammar rules over multiple files, but there is only one start nonterminal: program. On the other hand a redefine mechanism allows replacement of the alternates for a specific nonterminal, or extension of the alternates of an existing nonterminal. Automatic whitespace handling in TXL may be disabled via a switch in which case whitespace must be explicitly handled within the grammar.

Rats! [3] is a parser generator that generates Java based on parsing expression grammars (PEGs). PEGs are a recent reintroduction of Aho and Ullman’s TDPL formalism [1, chapter 6]. PEGs are not context free grammars since the rules are treated in order with a greedy matching strategy, and as a result not all context free parses will be obtained with a PEG based parser. In RATS!, PEGs are closed under set difference, which eases the deleter problem, although the need to take care over ordering (since the language of a PEG is not independent of rule ordering) complicates matters. The tool allows adding, overriding and removing of individual alternates of grammar rules. In order to perform these modifications, Rats! provides module parameters. Rats! provides operators to add + a new alternate before or after an existing one, to remove - an alternate, and to override := a specific alternate or an entire production.

SDF [10] does not support mechanisms to allow retraction or replacement of production rules or alternates of production rules. The parameterisation mechanism of SDF allows the renaming of terminals and nonterminals. For example, consider a module to manipulate tables with two parameters, key and value of the type Key and Value respectively. When importing this generic module the type Key can be bound to the nonterminal Identifier and the type Value can be bound to the nonterminal Type, in order to create a type environment. The parameterised modules are mainly employed when SDF is used to described the signature of ASF functions and not for describing the syntax of languages.

SDF [11] distinguishes three different types of syntax sections: lexical syntax, context-free syntax, and core syntax. The lexical syntax and context-free syntax sections are specializations of the latter one. The grammar rules in both sections are translated to core syntax grammar rules when the grammar is processed by the parse table generator. A number of grammar transformations will take place, but in this paper we will discuss only the transformation rules re-
lated to the processing of whitespace, new lines and comments, for the other grammar transformation rules we refer to [11].

In the grammar rules of the core syntax the grammar developer has to indicate explicitly where layout has to be inserted in left hand side of a grammar rules (note that in SDF the left and right hand side are swapped with respect to (E)BNF rules). This has to be done via the non-terminal LAYOUT?. where the ? indicates that layout is optional. There may be layout tokens inserted, but it is not strictly necessary. The grammar transformation that takes place when translating a lexical syntax section into a core syntax section does not insert this LAYOUT? between the members in the left-hand side of the lexical grammar rules. For instance, \([a-z]\)[a-zA-Z0-9]* \(-\) \(\rightarrow\) Id is translated into \([a-z]\)[a-zA-Z0-9]* \(-\) \(\rightarrow\) Id. The grammar transformation that takes place when translating a context-free syntax section into a core syntax section inserts the LAYOUT? nonterminal between all members in the left-hand side of the context-free grammar rules. For instance, "begin" Decls Stats "end" \(-\) \(\rightarrow\) Program is translated into "begin" LAYOUT? Decls LAYOUT? Stats LAYOUT? "end" \(-\) \(\rightarrow\) Program. There is no LAYOUT? nonterminal inserted before the first member or appended after last member, except for the rules that have the predefined nonterminal START in the right hand side, for instance, LAYOUT? Program LAYOUT? \(-\) \(\rightarrow\) START. If a member in the left hand side of a context-free grammar rule is nullable, this may lead to ambiguities. This problem is solved by explicitly restricting the recognition of layout characters after a LAYOUT?. This is for instance achieved in the following way, under the assumption that a whitespace, newline and tabular are the layout characters:

\[
\text{context-free restrictions}
LAYOUT? \(-\) \(\rightarrow\) \{ \ \} \ \{\ \n\} \ {t}
\]

SDF provides, like TXL, only one LAYOUT nonterminal. The layout can be defined by the grammar writer, but holds for the entire grammar definition so in the case of language embedding, the LAYOUT nonterminal would normally be set to the union of the individual whitespace languages. Module-specific whitespace conventions are an active development topic in SDF.

7. CONCLUSIONS

We have described a modularisation technique for context free grammars in which productions are copied into importing modules, with deleters allowing individual productions to be suppressed from the copying process. Since removing a production from a context free grammar yields a context free grammar, this method for removing parts of a context free language cannot accidentally yield a context sensitive language, as would be the case if we used set difference over context free nonterminals. We have also developed a semantics for automatic whitespace insertion which supports the traditional conventions used in programming language standards but which allows accurate embedding of languages with differing whitespace conventions.

We believe that our concrete notation is compact and easy to use. We think it is important that tools are able to display exact character-level grammars for real programming languages, as ART is able to do. Our modularity and whitespace insertion semantics are, however, independent of our particular implementation, and we believe they can be retrofit in a straightforward manner to most grammarware tools.

8. REFERENCES


