Dynamic grammars and semantic analysis

Pierre Boullier

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Dynamic grammars & semantic analysis

Pierre Boullier

N° 2322

Août 1994

PROGRAMME 2

Calcul symbolique,

programmation

et génie logiciel

ISSN 0249-6399

Dynamic grammars & semantic analysis

Pierre Boullier*

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet ChLoE

Rapport de recherche n°2322 — Août 1994 — 51 pages

Abstract: We define a *dynamic grammar* as a device which may generate an unbounded set of context-free grammars, each grammar is produced, while parsing a source text, by the recognition of some construct. It is shown that dynamic grammars have the formal power of Turing machines. For a given source text, a dynamic grammar, when non ambiguous, may be seen as a sequence of usual context-free grammars specialized by this source text: an initial grammar is modified, little by little, while the program is parsed and is used to continue the parsing process. An experimental system which implements a non ambiguous *dynamic parser* is sketched and applications of this system for the resolution of some semantic analysis problems are shown. Some of these examples are non-trivial (overloading resolution, derived types, polymorphism, ...) and indicate that this method may partly compete with other well-known techniques used in type-checking.

Key-words: parsing, incremental, extensible, LR, unbounded look-ahead, context-free, context-sensitive, ambiguity, type-checking.

(Résumé : tsvp)

*Email: Pierre.Boullier@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Grammaires dynamiques et analyse sémantique

Résumé : Nous définissons une *grammaire dynamique* comme un dispositif qui peut engendrer un nombre non borné de grammaires indépendantes du contexte, chacune de ces grammaires étant produite, au cours de l'analyse d'un texte source, par la reconnaissance de constructions particulières. On montre que les grammaires dynamiques ont la puissance formelle des machines de Turing. Pour un texte source donné, une grammaire dynamique, lorsqu'elle est non ambiguë, peut être vue comme une séquence de grammaires usuelles, spécialisées par l'analyse de ce texte : une grammaire initiale est modifiée au fur et à mesure de l'analyse du programme et est utilisée pour en poursuivre l'analyse. Un système expérimental, qui implante un *analyseur dynamique* non ambigu est esquissé et des applications de ce système à la résolution de quelques problèmes d'analyse sémantique sont proposés. Certains de ces exemples sont non triviaux (résolution de surcharge, types dérivés, polymorphisme, ...) et indiquent que cette méthode peut partiellement concurrencer les techniques bien connues utilisées en contrôle de type.

Mots-clé : analyse, incrémental, extensible, LR, pré-vision non bornée, indépendant du contexte, dépendant du contexte, ambiguité, contrôle de type.

1 Introduction

In the sequel we assume that the reader is familiar with the LR-method and terminology (see [1] for example).

This paper is divided into three main parts:

1. the concept of dynamic grammars;
2. an implementation into an experimental system;
3. its usage in type checking.

In the first part, we give a formal definition of the notion of dynamic grammars. Intuitively, we first may thought of a dynamic grammar as a set of usual context-free grammars, but these grammars are not statically known (except the first one which is called the initial grammar), but are rather computed while a given source text is analysed. A bottom-up dynamic grammar, which is the concept defined in this paper, used a kind of Shift-Reduce parsing algorithm: as usual, terminal symbols are shifted within the current grammar (a grammar in the set), while the reduction process works as follow. The recognition of a production (handle) is performed on a (dynamic) right-sentential form defined in the current grammar, this recognition induces the creation of a new grammar, and the replacement of the right-hand side of the handle by its left-hand side non-terminal leads to a new right-sentential form whose viable prefixes must be defined in the new grammar. In fact, a dynamic grammar is not the set of CF-grammars but the device which allows the generation of these grammars during the analysis of source texts. It is shown that dynamic grammars have the formal power of Turing machines.

In the second part, an experimental system which implements a dynamic parser (i.e. a device which can analyse texts written in a language defined by a dynamic grammar) is sketched. In fact, the domains where dynamic grammars can be used, seems rather extensive. In this paper we will only concentrate on context-sensitive features (i.e. type checking) of programming languages and therefore we restrict our system to non-ambiguous dynamic grammars. Our implementation is based upon LR(0)-parsers whose automata are lazily generated, when needed, while an input text is parsed. Since the analysis of this text may affect the grammar itself, the corresponding modifications are

themselves incrementally taking in account. The parsing actions of these automata are deterministically performed by using a *look-ahead oracle* which is responsible for the choice of a unique action, choice which may necessitate an unbounded look-ahead process.

The third part is devoted to an application of dynamic grammars in the semantic analysis phase of translators. This section is based upon a language which is (partly) defined by a dynamic grammar which expresses, besides the usual concrete syntax, its type constraints. This sample language collects a medley of caracteristic problems with which the user is usually confronted when dealing with type checking. Note that each of these problems has been (successfuly) processed by our experimental dynamic parser. Some of the presented examples (overloading resolution, subtyping, polymorphism, ...), are not trivial and demonstrate the power of the method and system. In particular, the (unique) identification of overloaded names is performed without any resolution algorithm.

This paper ends by a brief overview of some other researches in this field and by a concluded section.

2 Dynamic Grammars: A Formal Definition

In context-free grammars, rightmost derivations are defined which lead from the start symbol to sentences through right-sentential forms. These derivations always refer to a single grammar. We are going to define a similar notion which relies, not on a single grammar, but on a set of grammars, each derivation being performed using a grammar in this set.

Let $\mathcal{G} = \{G_i \mid G_i = (N_i, T_i, P_i, S)\}$ be an enumerable set of context-free grammars where:

- G_0 denotes an *initial grammar*.
- $N = \bigcup_i N_i$ is a non-empty finite or infinite set of *non-terminal* symbols.
- $\forall i, S \in N_i$ is the (common) *start symbol*.
- $T = \bigcup_i T_i$ is a non-empty finite or infinite set of *terminal* symbols. N and T are disjoint.

- $V_i = N_i \cup T_i$ and $V = \bigcup_i V_i$ is the *vocabulary*.
- $P_i \subseteq N_i \times V_i^*$ is the set of rule (for G_i). Each rule is denoted by $A \rightarrow \alpha$.
- $P = \bigcup_i P_i$.

We define, on each G_i, two binary relations. The first one named *extended rightmost derivation* and denoted $\underset{G_i,rm}{\Rightarrow}$ as being:

$$\beta A x \underset{G_i,rm}{\Rightarrow} \beta \alpha x$$

where $A \rightarrow \alpha \in P_i$, $\beta \in V_i^*$ and $x \in T^*$. This relation is closed to the traditional one, except that the suffix x is defined on the terminal vocabulary T and not only on the local terminal vocabulary T_i.

The second one named *prefixed rightmost derivation* and denoted $\underset{G_i,pr}{\Rightarrow}$ is defined by:

$$\beta A x \underset{G_i,pr}{\Rightarrow} \beta \alpha \quad (1)$$

$$\beta A x \underset{G_i,pr}{\Rightarrow} \beta \quad (2)$$

where $\alpha, \beta \in V_i^*$, $A \in N_i$ and $x \in T_i^*$. Case (1) stands iff $\exists A \rightarrow \alpha \in P_i$ while case (2) occurs iff $\nexists A \rightarrow \alpha \in P_i$.

If $S \overset{*}{\underset{G_i,pr}{\Rightarrow}} \sigma$, any prefix of the string σ is called a *dynamic viable prefix* for G_i.

If there is a sequence of grammars G_n, G_{n-1}, ..., G_{p+1}, G_p, ..., G_i such that:

$$\begin{array}{lccccc}
S & \underset{G_n,rm}{\Rightarrow} & \gamma_n & = & \alpha_n & = & \beta_{n-1}A_{n-1}x_{n-1} \\
 & \underset{G_{n-1},rm}{\Rightarrow} & \gamma_{n-1} & = & \beta_{n-1}\alpha_{n-1}x_{n-1} & = & \beta_{n-2}A_{n-2}x_{n-2} \\
 & \vdots & & & & & \\
 & \underset{G_{p+1},rm}{\Rightarrow} & \gamma_{p+1} & = & \beta_{p+1}\alpha_{p+1}x_{p+1} & = & \beta_pA_px_p \\
 & \underset{G_p,rm}{\Rightarrow} & \gamma_p & = & \beta_p\alpha_px_p & = & \beta_{p-1}A_{p-1}x_{p-1} \\
 & \vdots & & & & & \\
 & \underset{G_i,rm}{\Rightarrow} & \gamma_i & = & \beta_i\alpha_ix_i &
\end{array}$$

where, for each string γ_p, its prefix $\beta_p\alpha_p$ is a dynamic viable prefix for G_p, the string γ_i is called a *dynamic right-sentential form*.

When a dynamic right-sentential form x is an element of T^* and when it last has been produced by a extended rightmost derivation in the initial grammar (i.e. $S \underset{G_n,rm}{\Rightarrow} \gamma_n \ldots \underset{G_0,rm}{\Rightarrow} \gamma_0 = x$), it is called an *dynamic right sentence*.

If $\gamma_{p+1} = \beta_pA_px_p$ is a dynamic right-sentential form, $\gamma_{p+1} \underset{G_p,rm}{\Rightarrow} \gamma_p = \beta_p\alpha_px_p$, and γ_p is also a dynamic right-sentential form then the rule $A_p \rightarrow \alpha_p \in P_p$ at that position, is call a *dynamic handle*. Though γ_p is not a right-sentential form for G_p since x_p may be a string which is not in T_p^*, we may however speak of (dynamic) handles and hence rewrite a string by performing reductions within different grammars.

In fact, two consecutive grammars G_{p+1} and G_p, in a given sequence, should be related. Assume that we have $\beta_pA_px_p \underset{G_p,rm}{\Rightarrow} \beta_p\alpha_px_p$, we know that β_pA_p is a dynamic viable prefix for G_{p+1} and $\beta_p\alpha_p$ is a dynamic viable prefix for G_p and hence that β_p is a dynamic viable prefix for both grammars.

The language defined by $\mathcal{G}$ is the set of all its dynamic right sentences. However, since the number of grammars in $\mathcal{G}$ may be unbounded, a dynamic grammar will be defined, not as $\mathcal{G}$, but rather as being a device which can generate this set of context-free grammars.

Let us define more notions and notations.

- (G_i, π, α, x): where π is an element of P^*, α is an element of V_i^* and x an element of T^* is a *dynamic configuration*.

- (G_i, π, α, x): is a dynamic *viable* configuration when α is a dynamic viable prefix for G_i.
- $c_0 = (G_0, \epsilon, \epsilon, x)$: is an *initial* dynamic viable configuration.
- $c_f = (G_f, \pi, S, \epsilon)$: is a *final* dynamic viable configuration.
- $(G_i, \pi, \alpha\beta, x)$: is a dynamic viable configuration *reduceable* by $B \to \beta$ when $\exists B \to \beta \in P_i$ ($B \to \beta$ is a dynamic handle).

Let $\mathcal{M}$ be a turing machine called the *rule engine*. This machine takes as input a dynamic viable configuration $c_i = (G_i, \pi, \alpha\beta, x)$ reduceable by $B \to \beta$ and generates, upon acceptance, a context-free grammar G_{i+1} (i.e. $G_{i+1} = \mathcal{M}(c_i, B \to \beta)$).

On dynamic viable configurations we define two binary relations:

name	*scan*	*emit*
denotation	$\mapsto$	$\underset{\mathcal{M}}{\rightsquigarrow}$

$$\mapsto \;=\; \{(c, c') \mid c = (G_i, \pi, \alpha, tx) \wedge c' = (G_i, \pi, \alpha t, x)\}$$

This scan is the analog of a shift action in a Shift-Reduce parser. As a consequence, the scan relation may only be applied if the symbol t is an element of T_i.

$$\underset{\mathcal{M}}{\rightsquigarrow} \;=\; \{(c_i, c_{i+1}) \mid c_i = (G_i, \pi, \alpha\beta, x) \wedge c_{i+1} = (G_{i+1}, \pi B \to \beta, \alpha B, x)\}$$

If c_i is reduceable by $B \to \beta$ and $G_{i+1} = \mathcal{M}(c_i, B \to \beta)$ then we have $c_i \underset{\mathcal{M}}{\rightsquigarrow} c_{i+1}$. Emit is analog to a reduce action in a Shift-Reduce parser where the right-hand side of a dynamic handle is recognized within grammar G_i and the transition upon the left-hand side non-terminal is performed within grammar G_{i+1}.

By composing $\mapsto$ and $\underset{\mathcal{M}}{\rightsquigarrow}$, we define a binary relation denoted by $\underset{\mathcal{M}}{\Rightarrow}$ and called *bottom-up dynamic reduce*:

$$\underset{\mathcal{M}}{\Rightarrow} \;=\; \overset{*}{\mapsto} \underset{\mathcal{M}}{\rightsquigarrow}$$

Intuitively the scan relation performs reading in the source text (for a given grammar) while emit captures both the notion of reduction and the generation of a (new) grammar: a (dynamic) handle is chosen, the rule engine performs the generation of a new grammar and then the reduction using this handle is done.

A dynamic viable configuration c_i is *valid* iff we have $c_0 \overset{*}{\underset{\mathcal{M}}{\Rightarrow}} \overset{*}{\mapsto} c_i$ where c_0 is an initial configuration. In the sequel only valid dynamic viable configurations are considered and simply called configurations.

Definition 1 *We call context-free bottom-up dynamic grammar (dynamic grammar for short)*[1] *the couple* $(\mathcal{M}, G_0)$.

The language defined by a dynamic grammar $(\mathcal{M}, G_0)$ *is:*

$$\mathcal{L}(\mathcal{M}, G_0) = \{x \mid (G_0, \epsilon, \epsilon, x) \overset{*}{\underset{\mathcal{M}}{\Rightarrow}} c_f\}$$

Where $c_f = (G_f, \pi, S, \epsilon)$ is a final configuration and π is a *dynamic right parse* for x w.r.t. $(\mathcal{M}, G_0)$[2]. We note that the ith element (from left to right) in π is a rule in grammar G_{i-1}.

If $c_0 = (G_0, \epsilon, \epsilon, x)$, $c_0 \overset{*}{\underset{\mathcal{M}}{\Rightarrow}} c_f^{'} = (G_f^{'}, \pi^{'}, S, \epsilon)$ and $c_0 \overset{*}{\underset{\mathcal{M}}{\Rightarrow}} c_f^{''} = (G_f^{''}, \pi^{''}, S, \epsilon)$ such that $\pi^{'} \neq \pi^{''}$, the string x is *ambiguous* w.r.t $(\mathcal{M}, G_0)$ and the dynamic grammar is also said to be ambiguous.

We see that if the rule engine $\mathcal{M}$ is such that we have $G_0 = \ldots = G_i = \ldots = G_f$ then $\mathcal{L}(\mathcal{M}, G_0)$ is the context-free language $\mathcal{L}(G_0)$ and π is a right parse for x in the usual sense.

The following theorem shows that dynamic grammars have the same formal power as Turing machines.

Theorem 1 *Let L be a recursively enumerable set. There exists a dynamic grammar $(\mathcal{M}, G_0)$ such that $L = \mathcal{L}(\mathcal{M}, G_0)$.*

[1] This paper defined dynamic grammars as a device oriented toward bottom-up parsing; of course, a similar notion of context-free top-down dynamic grammar could be defined.

[2] In fact, this definition captures both the notion of language (as a set of strings) and the way a string could be (bottom-up) parsed (or recognized when the second field of a configuration, which contains a partial dynamic right parse, is omitted).

Proof: Consider T the alphabet of L, $N = \{S\} \cup \{A_i \mid i = 0, 1, 2, \ldots\}$ and the (infinite) regular grammar whose rule set is:

$$P_i \left\{ \begin{array}{l} P_2 \left\{ \begin{array}{l} P_1 \left\{ \begin{array}{rcl} S & \rightarrow & A_0 x_1 \\ A_0 & \rightarrow & \epsilon \\ A_1 & \rightarrow & A_0 \end{array} \right. \\ S \rightarrow A_1 x_2 \\ A_2 \rightarrow A_1 \end{array} \right. \\ \vdots \\ S \rightarrow A_{i-1} x_i \\ A_i \rightarrow A_{i-1} \\ \vdots \end{array} \right.$$

Where the x_i's are the elements of L. We see that this infinite device describes L. Since L is a recursively enumerable set, there exists a Turing machine which enumerates each word x_i in L exactly once. The rule set P_0 of the initial grammar G_0 is such that $P_0 = P_1$ and x_1, in the first rule, is the first word of L which has been enumerated by a first call to the Turing machine. Let x be a word in L. The initial configuration c_0 is $(G_0, \epsilon, \epsilon, x)$. The first step of this parsing process is the following: c_0, the reduceable initial configuration by $A_0 \rightarrow \epsilon$ is given to $\mathcal{M}$ which produces $G_1 = G_0$ and the emit machine generates the configuration $c_1 = (G_1, \pi_1 = A_0 \rightarrow \epsilon, A_0, x)$.

After this initialization step, the rule engine $\mathcal{M}$ will work as follows: it accepts c_i any reduceable configuration by some rule as input, calls the Turing machine which enumerates the (next) word x_{i+1} and emits the grammar G_{i+1} whose rule set is P_{i+1}.

Let $c_i = (G_i, \pi_i, A_{i-1}, x)$ be the configuration produced after the i^{th} step by the emit machine.

Then two cases can occur:

1. if $x = x_i$ then the scan machine produces from c_i the configuration $c'_i = (G_i, \pi_i, A_{i-1} x_i, \epsilon)$, c'_i is given to the emit machine which returns a final configuration $c_{i+1} = (G_{i+1} = G_i, \pi_i S \rightarrow A_{i-1} x_i, S, \epsilon)$. The parser may then stop.

2. if $x \neq x_i$ then c_i is given to the emit machine which calls $\mathcal{M}(c_i, A_i \rightarrow A_{i-1})$, the turing machine is called, the word x_{i+1} is enumerated, the grammar G_{i+1} is generated and the configuration $c_{i+1} = (G_{i+1}, \pi_{i+1} = \pi_i A_i \rightarrow A_{i-1}, A_i, x)$ is finally returned.

We see that the previous process can be iterated until the Turing machine enumerates x.

Conversely, if $x \in \mathcal{L}(\mathcal{M}, G_0)$, by definition there exists $c_0 = (G_0, \epsilon, \epsilon, x) \overset{*}{\underset{\mathcal{M}}{\Rightarrow}} c_f = c_n = (G_n, \pi_n, S, \epsilon)$ which shows that x is the n^{th} enumerated word of L.

$\diamond$

We can observe that the grammars $G_0, G_1, \ldots, G_i, \ldots$ used in the proof of theorem 1 are such that their rule set P_i form an increasing sequence $P_0 = P_1 \subset \ldots \subset P_i \subset \ldots$. This shows that the power of Turing machines may be reached by a regular grammar whose rule set (and right-hand side lengths) are not bounded. It also shows that the power of dynamic grammars may be reached with a rule engine which (only) adds rule to its input grammar without any deletion. However, for practical purposes, our rule engines will be able to both add and delete rules.

In this proof, the idea of having a Turing machine, the rule engine, which uses another one seems artificial, but a similar method could be used in practical cases. An example could be found in paragraph 4.8 where the problem of forward references in goto statements are treated in a similar way: since the rule engine knows the various grammars seen so far $G_0, G_1, \ldots G_i$, the various prefixes, partial parses, etc ..., it can, for taking its current decision, reactivate another dynamic parser and, by this means, perform several (partial) passes over the source text.

3 A Dynamic Parser

We call *dynamic parser* a device which, from a dynamic grammar and a string x is capable of computing configurations. This parser (among other things) uses scan/emit machines which computes the scan/emit relations and the rule

engine of a dynamic grammar. It is allowed to halt on reaching a final configuration.

As in usual context-free parsing, for efficiency reasons, dynamic parsers may only handle subclasses of dynamic grammars. In this section we sketch a dynamic parser implementation which works with (almost) non ambiguous dynamic grammars. More precisely, even with an ambiguous dynamic grammar, for any initial configuration it will produce at most a single dynamic right parse. The main reason behind this restriction choice, stems from the fact that our system is intended to demonstrate its power during the semantic analysis phase of programming languages (see section 4), which, by nature, should lead to unambiguous parse of sentences. Note that an implementation using a graph-structured stack, as in ([14]), [15] or [13]), should lead to an unrestricted dynamic grammar implementation. This means that the whole process should work deterministically and that at most a single dynamic right parse can be built for any source string. Therefore, every grammar produced by the rule engine should be unambiguous or, to be more accurate, if an ambiguity, in a given grammar, is exhibited during an analysis, the system should be able to choose a single action among all these possibilities.

We define a bottom-up parsing process in which the choice of a dynamic handle must be done without any look-ahead since x, the last component of a configuration, is a string which is not necessarily an element in T_i^*. Hence an implementation based upon an LR(0) parser seems a good choice. However, the various G_i are dynamically generated and the constraint for each one to be LR(0) is (in practice) far too strong. It even seems that, ideally, this parsing process should not be constrained by the class of grammar produced by the rule engine, i.e. G_i could be a (general) context-free grammar and therefore the underlying parsing algorithm must be able to handle it. In order to conciliate these opposite points of view we proposed the following schema: an LR(0) based parser which is equiped with a *look-ahead oracle*. Each time the LR(0) parser falls into a non-LR(0) state (i.e. it exists a Shift/Reduce or a Reduce/Reduce conflict), the oracle is called and is responsible for the choice of at most a single parsing action. If the oracle returns no action, a syntactic error is detected and the whole process stops (or an error recovery mecanism is invoked). If the response of the oracle is a Shift action, the scan machine is called, if the action

is Reduce by some production (the oracle chooses a handle), the emit machine is called with that production.

To perform this task, the look-ahead oracle must be able to resolve all the situations and therefore it must use a general context-free parsing algorithm. Moreover the parsing of the look-ahead string may rely upon various grammars which must themselves be produced on each reduce action of this look-ahead process. This means that the emit machine (and hence the rule engine) could be called from this oracle. Since the answer of the oracle to its callee must be unique, this process as a whole must be unambiguous even if the oracle must be able to handle ambiguous situations and to choose among them a unique action.

It is well known that, in some pathological cases, a non deterministic parser based upon a Shift/Reduce algorithm may loop without consuming any input symbol. This could happen in two ways:

- the parse-stack takes back a previous value,
- the parse-stack grows infinitely.

The first case occurs with cyclic grammars (i.e. when $A \stackrel{+}{\Rightarrow} A$) and involves an unbounded ambiguity. The second case occurs when $A \underset{rm}{\stackrel{+}{\Rightarrow}} \alpha A x$ and $\alpha \stackrel{+}{\Rightarrow} \epsilon$. If $x = \epsilon$, we fall back to the previous case. If $x \neq \epsilon$, this derivation does not imply any ambiguity but the grammar at hand is not LR(k) for any k (it is not even LR(π)— see [9] for a definition). The unbounded growth of the parse-stack is due to the existence of a cycle (which spells α) in the underlying automaton. In the framework of a single grammar such situations may be solved by implementing the parse-stack as a graph with cycles that express its potentially unbounded growth. But, with dynamic grammars, this type of solution cannot resolve all problems: consider a current grammar where $A \stackrel{+}{\Rightarrow} A$ and say, to simplify, that it contains the rule $A \rightarrow A$ which has been chosen as dynamic handle by the emit machine. Since the rule engine may or may not delete this rule (and by the fact suppress the problem), the parser does not know whether the process is going to loop or not. To avoid this disagreement, we decide to put a (minor) restriction upon the rule engine and to require a more deterministic behavior. Three attitudes are possible:

1. assume that the rule engine will not endlessly return a grammar which presents this problem,

2. assume that the rule engine will never return such grammars,

3. or, on the contrary, assume that the same grammar will always be produced, when the rule engine is placed in similar situations.

Choice (1), from the parser's point of view is the most comfortable but do not seem to be very reasonable since no test in the parser can check its conformity. In our experimental system, choice (2) has been taken and the conformity with this restriction of each output grammar is performed. If this restriction seems too heavy for practical purposes we intend to move to the next choice. Choice (3) assumes that, when the rule engine is called on reductions performed while traversing empty cycles, it will return identical grammars. It should be noticed that this restriction do not prohibit the rule engine to generate those kinds of grammars, but only assumes that it will generate the same grammar in similar situations. As already mentioned, the handling of choice (3) will necessitate the usage of a graph-structured parse-stack.

In our current experimental system the look-ahead oracle works as follow. We first try to solve the LR(0) conflict with k symbols of look-ahead, k being a parameter which can be adjusted dynamically. Then the set of all paths that use these k symbols is explored depth-first. Each time an error is detected the current path is given up. If only paths corresponding to a single parsing action have been found, this action is returned. This mecanism is in fact a dynamic LR(k) method using different grammars for checking the look-ahead string. If no path is found, no action is returned. If paths associated with different parsing actions are valid, the LR(0) conflict cannot be resolved with k tokens of look-ahead. By extension we say that the dynamic grammar is not LR(k). Conversely, if a source text is successfully analysed by this dynamic parser we can only say that no LR(k) conflict has been detected. Notice that nothing can be said about the class of the dynamic grammar since another source text might well bring out an LR problem.

In fact there is (theoretically) no reason to stop after k tokens, this process may be continued on the whole suffix x. In such a case, if the oracle used a general context-free algorithm, real ambiguities could be detected, for (local)

non-determinism which led to error would be automatically rejected by this mecanism when it fell into a dead-end path. In practice, this costly method may be modified, without loosing too much power, in the following way. Languages usually possess what is called *key*-terminals which are high level sentences separators (or terminators), such as —typically enough— a semi-colon. These key-terminals are often used in error recovery methods to implement what is called *panic mode*. We have used a similar notion for our purpose. Let $x = ytz$ be the current suffix and t the first (from left to right) key-terminal in x. Our supposition is that, for a given language, a conflict which cannot be resolved in using yt as lookahead could not be resolved either in using ytz. For a given suffix x, the value $k = |yt|$ (or $k = |x|$ if there is no key-terminal in x), will be the current parameter of our look-ahead parser. If the conflict is not resolved within this length, the source text (and hence the dynamic grammar) is said to be *quasi*-ambiguous. This statically unbounded but dynamically constrained property seems a good trade-off. On the one hand, it keeps the process computationally tractable while, on the other hand it is still powerful enough. For example, it allows to solve (see 4.4), the non trivial problem of identifying overloaded symbols.

If a text is quasi-ambiguous, (a special entry of) the rule engine may itself participate to its resolution. Some brute force methods such as the disambiguation rules of YACC can be used when priorities and associativity kinds have been assigned to rules and symbols and/or some other specific methods depending upon the language and the conflict. If the rule engine does not solve the current conflict, some parser built-in methods (e.g. priority is given to the Shift action or else to the Reduce action by the longest right-hand side rule or ...) are performed and ultimately a single action is selected.

We have seen that each grammar G_i is only used for a while (until the next reduction) and hence a partial lazy generation of the LR(0) automaton (as opposed to a full generation) seems adequate: only the part (states and transitions) of the automaton which is used during this local analysis will be constructed.

We note $M_i^0 = (Q_i, V_i, q_i^0, \delta_i, F_i)$ the LR(0) automaton associated with G_i. When a new configuration $c_i = (G_i, \pi B \rightarrow \beta, \alpha B, x)$ is returned by the emit machine, we only compute the part of M_i^0 whose path accounts for αB. We first compute its initial state q_i^0 and if $\alpha = X_1 \dots X_p$, for each $j \in [1..p]$ we compute

the states q_i^j and the part of the transition function δ s.t. $q_i^j = \delta(q_i^{j-1}, X_j)$. We then compute $q = \delta(q_i^p, B)$. The current parse-stack of the current LR(0) parser is initialized with $q_i^0 q_i^1 \ldots q_i^p q$. If this construction is not possible (i.e. αB is not a dynamic viable prefix), it means that the rule engine $\mathcal{M}$ is erroneous and therefore $(\mathcal{M}, G_0)$ is not a dynamic grammar.

The previous construction is not always necessary. This is trivially the case when two successive grammars G_i and G_{i+1} are identical, the part of M_i^0 that was constructed and the corresponding parse-stack can be reused as such for M_{i+1}^0. The idea to reuse (part of) the construction work already performed may be extended. Without loss in generality let us assume that grammar G_{i+1} is a *modification* of G_i, i.e. G_{i+1} is obtained from G_i by deleting and adding rules. This view of modification of the input grammar is not a restriction since it is always possible to get G_{i+1} from G_i by first deleting (the rule set of) G_i and then adding (the rule set of) G_{i+1}. It is possible to incrementally modify the part of M_i^0 being built so far, each time a rule is deleted or added in order to get the corresponding part of M_{i+1}^0. Moreover the previous process must be invertible when it is called from the look-ahead oracle since, when returning from level $i+1$ to level i, the environment must be restored. (In fact the only thing which is needed is the linear part of the LR(0) automaton which spells αB and the corresponding parse-stack). Several approaches ranging from complete storing of a given environment at level i to a complete [re]computing of this level from level $i-1$ or even level $i+1$ may be thought of.

In practice the view of grammars evolving by modifications and the corresponding view of a single LR(0) automaton (and parse-stack) being modified each time a rule is added or deleted seems reasonable for several reasons:

- In practice most reductions are not associated with grammar modifications.
- It is often the case that when a modification occurs, it does not alter the part of the automaton corresponding to the parse-stack. For example we will see in section 4 that the handling of the declaration part in our sample language essentially modifies the statement part processing.
- It is well known that in programming languages, most grammatical constructions are LR(1). In our framework this means that a look-ahead

of one (or few) token(s) in the oracle is often sufficient to resolve a conflict. If during this look-ahead check the modifications of the grammar associated with the reductions involved do not alter the immediate look-ahead string, it is not necessary to call the rule engine and hence to do (and undo upon return) the automaton modification. In the examples of section 4, the only time the oracle needs to call the rule engine in order to get new grammars to parse the look-ahead is when a block is entered and local labels and goto statements are processed.

3.1 Incremental Modifications of the LR(0) Automaton

Assume that the rule engine $\mathcal{M}$ is called from the emit machine with configuration $c_i = (G_i, \pi, \alpha\beta, x)$ and rule $B \rightarrow \beta$ and that it returns the grammar G_{i+1} which is a modification of the grammar G_i such that the rule set of G_{i+1} is $P_{i+1} = (P_i \setminus P_i^d) \cup P_i^a$ where P_i^d is the set of rules to be deleted and P_i^a is the set of rules to be added. Before returning the configuration $c_{i+1} = (G_{i+1}, \pi B \rightarrow \beta, \alpha B, x)$ to its caller, the emit machine performs some updates on $M_i^0 = (Q_i, V_i, q_i^0, \delta, F_i)$, the part of the LR(0) automaton which is partially constructed so far, in order to get M_{i+1}^0, a new automaton in which the relevant modifications of P_i^d and P_i^a have been taken into acount. Remember that $\alpha\beta$ spells a path in M_i^0 and that αB should spell a path in M_{i+1}^0. The lazy incremental LR(0) parsing algorithm guarantees that, for each $c_i = (G_i, \pi, \alpha, x)$ the corresponding automaton M_i^0 contains (at least), a (non empty) initial state q_i^0, a transition function δ which leads from this initial state to non empty states by transitions over any prefix of α (i.e. if $\alpha = X_1 \cdots X_p$, $\forall i \in [1..p]$, $\delta(q_i^0, X_1 \cdots X_i) \neq \Phi$), and that the corresponding *parse-stack* is $q_i^0 q_i^1 \cdots q_i^i q_i^{i+1} \cdots q_i^p$.

3.1.1 Add or Delete Rule $A \rightarrow \alpha$

The idea of the following algorithm is to minimize the influence of each individual rule update (addition or deletion) during a grammatical modification over the part of the LR(0) automaton being built so far and to postpone the

reconstruction of states and transitions until the moment when they are needed.

When a rule defining the non-terminal symbol A (an A-production) is added or deleted we first look for the states whose set of items would be modified by such an update. Each (already computed) state q with an item of the form $[B \rightarrow \beta \bullet A\gamma]$ is selected. Afterwards, each such q is not (immediately) recomputed but only isolated from its predecessors (if any) by "cutting" all its in-transitions. This means that every q becomes unreachable from the initial state and hence its successors are also unreachable from the initial state by a path going through q. Note that a state q which has been isolated may be "reactivated" later on, thus allowing access to its successors (if they have not been swept out by the garbage collector!). This isolation of a state from its predecessors is simply performed by modifying the transition function δ.

Each rule in $P_i^a \cup P_i^d$ is considered in turn and is passed as an argument to the procedure *update* which is sketched in Table 1.

```
        procedure cut-in-transitions (q)
            for each (q', X) ∈ {(q', X) | δ(q', X) = q} do
(1)             δ(q', X) := Φ
            end do
        end procedure

        procedure update (A → α)
            for each state q ∈ Q_i do
                if [B → β•Aγ] ∈ q then
(2)                 cut-in-transitions (q)
                end if
            end do
        end procedure
```

Table 1: Algorithm for incremental addition/deletion of rules.

(1): The in-transitions for a particular state q are erased from the transition function δ.

(2): Every q which has an item ($[B \rightarrow \beta \bullet A \gamma]$) with an LR-marker before A has its in-transitions erased.

3.1.2 Viable Prefix Checking

When the new grammar G_{i+1} has been built, we must check that αB is a dynamic viable prefix for G_{i+1}, recompute (if necessary) the transition function δ and the corresponding states over αB, and build the new *parse-stack*. This work is performed by the function *check-viable-prefix* in Table 2 which is called with αB as an argument.

(1): When a transition from state q over symbol X leads to an empty state, this goal is (re)computed by advancing the LR(0) mark past X and by calling the *closure* function on the resulting subset. If *item-set* is a set of LR(0) items, then *closure* (*item-set*) is the set of items constructed from *item-set* by the two rules:

1. *closure* (*item-set*) is initialized with *item-set*.
2. If $[B \rightarrow \beta \bullet Y \gamma]$ is in *closure* (*item-set*) and $Y \rightarrow \rho$ is a production, then add the item $[Y \rightarrow \bullet \rho]$ to *closure* (*item-set*), if it is not already there. This rule is applied until no more new item can be added to *closure* (*item-set*).

(2): The new initial state q^0_{i+1} is (re)computed from S, the start-symbol,

(3): q^0_{i+1} should not be empty,

(4): The new *parse-stack* is initialized with the new initial state.

(5): Each state is (re)computed from its predecessor by transition over the corresponding symbol of the viable prefix, and the transition function is updated consequently (if needed).

(6): An empty state indicates that the parameter is not a viable prefix for the current grammar.

```
        function next-state (q, X) return state
            q' := δ(q, X)
            if q' = Φ then
(1)             δ(q, X) := q' := closure ({[A → αX•β] | [A → α•Xβ] ∈ q})
            end if
            return q'
        end function

        function check-viable-prefix (X_1 ... X_p) return boolean
(2)         q^0_{i+1} := closure ({[S → •α] | S → α ∈ P_{i+1}})
(3)         if q^0_{i+1} = Φ then return false end if
(4)         parse-stack := q^0_{i+1}
            for j := 1 to p do
(5)             q^j_{i+1} := next-state (q^{j-1}_{i+1}, X_j)
(6)             if q^j_{i+1} = Φ then return false end if
(7)             parse-stack := parse-stack || q^j_{i+1}
            end do
            return true
        end function
```

Table 2: Algorithm for checking the viable prefix.

(7): The *parse-stack* grows concurrently.

As already stated, these general algorithms could be greatly improved in order not to recompute things which do not need to have to. At some places in the look-ahead oracle, if it is known that a call to the rule engine will not change its result (i.e. the LR(0) conflict will be solved within a certain length which will not modify the grammar), the checking of the look-ahead could be performed within the same grammar. Hence the call to the rule engine could be avoided, the LR(0) automaton does not need to be modified neither when moving forward nor when returning backward.

4 Type Checking

This section aims at showing the usage of dynamic grammars in performing (part of) the semantic analysis phase in translators. Usually the syntax of (programming) languages is context-sensitive. The simplest example is given by identifiers which can be used only when they have been declared. It is known that this simple constraint cannot be captured by a context-free grammar. Traditionally, this inherently context-sensitive process is split in two parts: the first part is specified by a (subclass of a) context-free grammar which describes a super-set of our language (i.e. the CF grammar allows any identifier to occur in an expression without being declared) and involves some well known parsing method (LL, LR, ...); and a second part, called semantic analysis, checks that the source text is really an element of our language. This second part is traditionally described and implemented within some non syntactic formalism (semantic actions, attribute grammars, ...). This section will show, through examples, that the dynamic grammar paradigm may be used to solve, within a unified framework, some typical and non trivial semantic analysis problems.

In our system, the calls to the rule engine are an integral part of the way production rules are written. The rule engine may be invoked by the so-called *syntactic predicates* and *syntactic actions*. The code of these predicates and actions are under the control of the dynamic grammar implementor who can use the parser environment data structures and service routines. A syntactic predicate is a boolean function whose name, denoted by a `&` followed by an integer number, may follow any symbol in the right-hand side of a grammar rule. It means that during syntactic analysis the grammar symbol with which it is associated is recognized if and only if it is the symbol at hand and the boolean function answers `true`. Such a symbol with a predicate is said to be *extended*. A syntactic action is a procedure whose name, denoted by a `@` followed by an integer number, may occur at any place in the right-hand side of a grammar rule. This syntactic action is implemented as a non-terminal symbol whose production definition has an empty right-hand side, the real call to the rule engine being associated with this empty reduction. The couples action/predicate usually work as follows: actions perform (if any) the grammar modifications and may set global data structures which (later on) are examined

by predicates which, in their turn, can influence the course of the parsing process.

Syntactic actions are differentiated from the usual semantic actions called on each reduce in order to allow our system to be coupled with an arbitrary (further) syntax directed semantic processor.

In the current version, our experimental system is not linked with a "dynamic scanner" and, for a given language, only a (statically) bounded number of tokens are known. Therefore, it seems that the rule engine cannot generate rules with new terminal symbols. In fact syntactic predicates may be used to (partly) get round this restriction. Assume for example that a particular identifier, say "`foo`", has been recognized by the scanner as being an element of the generic token class "`ident`" and that we want to use this name as a terminal symbol in a new rule (say $A \rightarrow$ "`foo`") with the meaning that only the identifier whose name is "`foo`" will match this terminal. Since each occurrence of "`foo`" will be recognized by the scanner as being an "`ident`" this production will never be recognized. One solution is to generate the rule $A \rightarrow$ "`ident`"`&1` where the predicate `&1` answers `true` iff the name of the identifier is "`foo`". In fact, in order to facilitate the understanding of the generated rules we will write $A \rightarrow$ "`foo`" though this situation is handled via predicates. Nevertheless, new symbols like "`foo`" may themselves be extended with predicates which are supposed to handle some other situations.

In the sequel we do not want to advocate solving semantic analysis problems in the way it is done but rather to show than they can be solved within this framework. In fact, for each problem there exists a large variety of ways (in the very same sense that many grammars may describe the same language) of solving it.

Our example is a block-structured language. Each block, bracketed by "{" and "}" is an optional list of declarations followed by an optional list of statements. In the first initial grammar only integer and boolean scalar variables may be declared. Besides block, empty statement and if statement our language allows integer and boolean assignments. This mean that the left-hand side of an integer assignment may only be an integer scalar and its right-hand side an integer expression. Integer and boolean expressions are (separately) described with (some of) their usual operators. The only difference with a usual description is that the only integer literals are the integer numbers and the boolean

literals are `true` and `false`. This means that, for the time being, in this initial grammar, the notion of applied occurrence of, say, integer variable is not specified. This notion occurs in the right-hand side of some productions (describing integer expressions) but there is no rule with this notion as left-hand side.

4.1 Scalar Variables

Our first example is shown Table 3.

```
(1)   int a;
(2)   bool b;
(3)   {
(4)      int a, b;
(5)      a = b+1;
(6)   }
(7)   a = b;
```

Table 3: Scalar Variables.

(1): Until the end of this declaration, the rule engine, via syntactic action calls has accumulated information about the text seen so far, especially the fact that a variable whose name is "`a`", of type integer, has been declared in block level #0. On ";" a last call to the rule engine adds a rule defining an integer variable as being "`a`":

$$\textit{int-lhs} \quad \rightarrow \quad \text{"a"} \tag{3}$$

The fact that the type of this variable is integer is captured in the name of the non-terminal symbol in the left-hand side of rule (3).

Up till now, according to the current grammar, the only variable name which may appear in an integer expression or as left-hand side of an integer assignment is "`a`".

(2): On ";" the rule engine adds rule:

$$\textit{bool-lhs} \rightarrow \text{"b"} \tag{4}$$

Tests for the redeclaration (at the same block level) of variables are performed in the rule engine.

(3): The block level #1 is entered.

(4): On "," the rule engine generates the rule:

$$\textit{int-lhs} \rightarrow \text{"a"} \tag{5}$$

At this point the two rules (3) and (5) define "a" as an integer variable and are identical. Our language says that "a" in block level #1 hides the "a" in block level #0. The "a" declared at line (1) will only be visible again at block exit after line (6). Here, two possibilities are conceivable:

- the rule (5) is not added by the rule engine and therefore will not be deleted at block exit line (6),
- the rule (3) is first deleted and the (same) rule (5) is then added.

For the dynamic parser point of view, the first solution is simpler since no grammar modification is implied. However the second solution has been retained because it allows more systematic processing (the rules which define hidden variables are deleted) and above all these rules do not define the same variable and the semantic processing (though not tackled here), could need this distinction.

Therefore, On "," the rule engine deletes rule (3) and adds rule (5).

On ";" the rule engine deletes rule (4) and adds rule:

$$\textit{int-lhs} \rightarrow \text{"b"} \tag{6}$$

From now on and until the end of the current block, "b" is an integer variable.

(5): Here, the identifier "`a`" is recognized by rule (5) as an integer variable. Since the left-hand side of the assignment operator is *int-lhs*, the right-hand side must be a phrase of *int-expr* which means that syntactically the only operands which are allowed are integer literals. Hence the identifier "`b`" is recognized by rule (6).

(6): We quit block #1, and return back in block level #0. Variables which have been declared inside that block become inaccessible so the rule engine deletes rules (5) and (6) and the variables which have been hidden should be accessible again, so the rule engine adds rule (3) and (4).

(7): Here, "`a`" is recognized as being the integer variable that was declared at line (1), so the right-hand side of this assignment statement must be an *int-expr* whose operands should be integer literals. The token in hand "`b`" do not match any integer variable. So a syntax error is detected on the identifier "`b`".

4.2 Arrays & Functions

The notions of array and function is now added to our language. The elements of arrays are integer or boolean, the size of an index must be an integer expression and the number of their indexes are not bounded. Any reference to an array element must agree with its declaration (i.e. element type, indexes type which must be integer and number of indexes). Functions may return integer or boolean values and may have parameters whose type are integer or boolean. A reference to a function must agree with its declaration, it cannot be used in the left-hand side of an assignment statement.

The names of arrays and functions follow (for the time being) the same scoping rules as the ones for scalar variables.

The next example is shown Table 4.

(1): On "`,`" the rule engine adds rule:

$$\textit{int-lhs} \rightarrow \text{"}\texttt{i}\text{"} \tag{7}$$

On "`;`" the rule engine adds rule:

$$\textit{int-lhs} \rightarrow \text{"}\texttt{j}\text{"} \tag{8}$$

```
(1)   int i, j;
(2)   int a [4, 5];
(3)   int f (int, bool);
(4)   a [i][j] = f (1, i!=j);
(5)   i = f (i+j);
```

Table 4: Arrays & Functions.

(2): Until the end of this declaration, the rule engine has accumulated information about the text seen so far, especially the fact that an array variable whose name is "`a`", of type integer, with two indexes has been declared in block level #0. On "`;`" the rule engine adds rule:

$$\textit{int-lhs} \;\rightarrow\; \texttt{"a"}\ \texttt{"["}\ \textit{int-expr}\ \texttt{"]"}\ \texttt{"["}\ \textit{int-expr}\ \texttt{"]"} \tag{9}$$

After this declaration the only occurrence of an applied array identifier is inside an integer expression or as left-hand side of an integer assignment statement. The name of this array could only be "`a`", and it must have exactly two indexes of type integer.

(3): Here, the information gathered during the analysis of this declaration via various syntactic action calls indicates that a function variable whose name is "`f`", of type integer, with two parameters has been declared in block level #0. Moreover, the first parameter has type integer and the second has type boolean. On "`;`" a last call to the rule engine adds rule:

$$\textit{int-fun} \;\rightarrow\; \texttt{"f"}\ \texttt{"("}\ \textit{int-expr}\ \texttt{","}\ \textit{bool-expr}\ \texttt{")"} \tag{10}$$

Note that the left-hand side non-terminal name is *int-fun* and not *int-lhs* which prevents the use of functions in left-hand side of assignments. Once again, after this declaration the only occurrence of an applied function identifier is inside an integer expression, the name of this function could only be "`f`", and it must have exactly two parameters, the first one is of type integer while the second is of type boolean.

(4): Here, with the symbol "`a`" in hand, the only production to be entered is rule (9), "`i`" and "`j`" are both *int-expr* (see rules (7) and (8)), and the left-hand side of this assignment statement is recognized as being an *int-lhs* by rule (9). Hence the right-hand side should be an *int-expr*. Since "`f`" is a integer variable, rule (10) is entered. Its first parameter "`1`" is an *int-expr* while the second parameter "`i!=j`" is recognized as a *bool-expr*. The identifications of the various variables, number of parameters, and types lead to the validation of rule (10). Finally, line (4) is parsed by the rule in the initial grammar which defined an integer assignment statement.

(5): Here "`i`" is an integer variable (rule (7)) so the integer variable "`f`" could occur there if the suffix of rule (10) is validated. Its first parameter should be an integer expression, so "`i+j`" is valid, but the lack of second parameter causes a syntax error to be detected on "`)`".

4.3 Extensible Language

A language is said to be extensible when it is able to increase its own syntax. Of course, dynamic grammars seem very well suited to process such languages. The following feature is borrowed from PROLOG. We added to our language the possibility for a function with one or two parameters to be (also) used as an operator. Such functions are declared with three extra attributes. The first is an integer which gives its absolute precedence level (priority), the second tells wether it can be used as a prefix, infix or postfix operator and the third specifies its type of associativity (see line (2) of Table 5 for an example).

```
(1)   int a [4];
(2)   int f (int, bool)(150, infix, left);
(3)   a [1] = f (3 f true + 1, false);
(4)   a [3 f (1, true)] = 2;
```

Table 5: Extensible Language.

(1): On ";" the rule engine adds rule:

$$\textit{int-lhs} \;\rightarrow\; \texttt{"a"}\ \texttt{"["}\ \textit{int-expr}\ \texttt{"]"} \tag{11}$$

(2): As in paragraph 4.2, after the first ")", the rule engine adds rule:

$$\textit{int-fun} \;\rightarrow\; \texttt{"f"}\ \texttt{"("}\ \textit{int-expr}\ \texttt{","}\ \textit{bool-expr}\ \texttt{")"} \tag{12}$$

The last part of this function declaration specifies that "f" may also be used as an infix operator which is left associative and whose priority is "150". Since (by construction) in our language the additive operators have priority "100" while the multiplicative ones have priority "200", we see that "*" has priority over "f" which itself has priority over "+". The rule engine adds rule:

$$\textit{int-expr} \;\rightarrow\; \textit{int-expr}\ \texttt{"f"}\ \textit{bool-expr} \tag{13}$$

We note that "f" has been used in both rules (12) and (13). The priority value (here "150") and the left-associativity are coded into tables which are used in the look-ahead oracle to resolve ambiguities (see section 3). Rule (13) has in fact exactly the same form as the ones used in the initial grammar to specify the predefined integer and boolean operators. Because such rules may be inherently ambiguous (when they are both left and right recursive), they are (statically) associated with the same (but predefined) attributes (i.e. priority and associativity) as the dynamic operators. The rules, wether they come from the original grammar or are added by the rule engine, are all processed in a uniform manner.

It should be noted that our language becomes ambiguous. Imagine a function "u" with a single (say integer) parameter which can also be denoted as a prefix unary operator. The string "u(1)" denotes both the functional and the operator form. Though, from a semantic point of view, the choice does not matter, the look-ahead oracle (via the special entry of the rule engine already mentioned in section 3) must be able to choose among these possibilities.

(3): After "=", the right-hand side of this assignment must be an *int-expr*. The first "f" can only be an occurrence of the functional form described by

rule (12). Therefore the first parameter must be an *int-expr* and the second a *bool-expr*. Then "`3`" is an integer operand and the second "`f`" could only be the infix form. The remaining problem is wether "`3 f true + 1`" must be seen as "`(3 f true) + 1`" or as "`3 f (true + 1)`". The first form is chosen, due to the fact that "`f`" has higher priority than "`+`". In fact the second form would also have been rejected since the first operand of the operator "`+`" must be an *int-expr*.

(4): Here, the occurrence of "`f`" is identified as the infix operator. Its right operand must be boolean and a syntax error is then detected on "`,`" which is not a boolean operator.

4.4 Overloading of Symbols

The meaning of an *overloaded* symbol depends on its context. Overloading *resolution* consists in determining a unique meaning for that symbol. Such a resolution can involve a non trivial process. In ADA, for an expression, the resolution with an attribute grammar can be implemented by making two depth-first traversals (in fact one and a half) of a syntax tree. Within the formalism of dynamic grammars, the resolution of the overloading of symbols is for free. When a single meaning cannot be determined, the process results in an ambiguity detection.

To demonstrate this, we extend our language with the possibility to overload array and function variables. A function never hides an array with the same name (and vice versa). A function hides another function with the same name if and only if it has the same profile and is declared in an inner block. It is an error when it is declared at the same block level. An array hides another array with the same name if and only if the type of their elements and the number of their indexes are the same and the first one is declared in an inner block. It is an error when both arrays are declared at the same block level. Arrays or functions and scalar variables cannot overload. It is an error when they are declared at the same block level. At different block levels, the inner declaration hides the outer one.

So several functions and arrays with the same name may (co)exist at the same point, the purpose of the resolution mecanism being to choose among them at most one meaning.

An example is shown in Table 6.

```
(1)   int f [5], f (int), f (bool), f (int, bool);
(2)   bool f [5], f (int, int);
(3)   f [1] = f (f [2], f (4));
(4)   f [1] = f (f [2], f [4]);
```

Table 6: Overloading of Symbols.

(1): On ";" the rule engine has add rules:

$$\textit{int-lhs} \rightarrow \texttt{"f"}\ \texttt{"[}\ \textit{int-expr}\ \texttt{"]"} \tag{14}$$
$$\textit{int-fun} \rightarrow \texttt{"f"}\ \texttt{"(}\ \textit{int-expr}\ \texttt{")"} \tag{15}$$
$$\textit{int-fun} \rightarrow \texttt{"f"}\ \texttt{"(}\ \textit{bool-expr}\ \texttt{")"} \tag{16}$$
$$\textit{int-fun} \rightarrow \texttt{"f"}\ \texttt{"(}\ \textit{int-expr}\ \texttt{","}\ \textit{bool-expr}\ \texttt{")"} \tag{17}$$

(2): On ";" the rule engine has add rules:

$$\textit{bool-lhs} \rightarrow \texttt{"f"}\ \texttt{"[}\ \textit{int-expr}\ \texttt{"]"} \tag{18}$$
$$\textit{bool-fun} \rightarrow \texttt{"f"}\ \texttt{"(}\ \textit{int-expr}\ \texttt{","}\ \textit{int-expr}\ \texttt{")"} \tag{19}$$

The overloading resolution will be performed on a context basis and not (only) by the scoping rule mecanism.

(3): When the parser sees the first "f", rules (14)—(19) are entered, the first "[", reduces this set of possibilities to rules (14) and (18) which mean that the previous "f" can only be an array name whose elements type are either integer or boolean. On the LR(0) automaton, the transition on "]" leads to a state where a Reduce/Reduce conflict between rules (14) and (18) exists. Therefore, the look-ahead oracle is called. Assume that the current value of k, the number of look-ahead symbols, is 1. Since the current conflict cannot be resolved within that length, the look-ahead oracle enters its unbounded mode (i.e. it tries to resolve the conflict in using the

look-ahead context until it founds a key terminal — here the "`;`" at the end of line (3)). In fact it will check whether "`= f (f [2], f (4));`" is the suffix of an integer or boolean assignment statement. No answer will result in a syntax error, a single answer in the conflict resolution and both answers in a quasi-ambiguity detection. Though integer and boolean assignments are tried successively (due to the depth-first strategy), the following explanations are given as if all the possibilities were processed in parallel. Once again, the right-hand side of this assignment starts with an "`f`" and we enter rules (14)—(19). After seeing the first "`(`", this number is restricted to the four overloaded functions. Their first parameter is integer or boolean and, therefore, the third "`f`" is not restricted until seeing the second "`[`", which indicates an array reference. The types of its elements (integer or boolean) does not constrain the functions denoted by the second occurrence of "`f`". The "`,`" indicates that only functions with (at least) two parameters are allowed. Since the type of their first parameter is integer, this completely identifies the third "`f`" as being the one defined by rule (14). The type of the second parameter may still be integer or boolean. The second "`(`", indicates that the last "`f`" is one of the functions. The integer literal "`4`" reduces the number of possible functions to three while the first "`)`" reduces this number to one since there is a single function with a single integer parameter. The type of this function being integer, there is a single function with two integer parameters, its type is boolean. At the end, since the type of the right-hand side of the assignment statement is boolean, its left-hand side should also be boolean and the look-ahead oracle successfully announces that rule (18) has been chosen. The right-hand side of this assignment statement is then parsed again as a *bool-expr*.

(4): Here, the difference with the previous line occurs after the last "`f`", the "`[`" indicates an array with boolean or integer elements so "`f [4]`" is an *int-lhs* by rule (14) or a *bool-lhs* by rule (18). Therefore, "`f (f [2], f [4])`" is recognized by both rules (17) and (19). Assignments between integers or booleans being equally valid, the look-ahead oracle detects a quasi-ambiguity on "`;`".

We note that the overloading resolution is (naturally) performed by the unbounded mecanism of the look-ahead oracle.

4.5 Derived Types an Inheritance

Once again, we extend our language with the notion of *subtypes* which are new types derived from a (predefined) type or another previously defined subtype. For example we may defined `short` as being a subtype of `int`. The semantics, though irrelevant here, being that any value of type `short` is also of type `int`. Each subtype inherits the properties of its father type: arithmetic and comparison operators, assignment operator, constants, ..., which therefore induces a form of overloading. Wherever an element is valid, it could be replaced by an element of one of its subtypes. Conversely, when a subtype is declared, it allows for an explicit conversion from one of its ancestor type to its subtype (i.e. "`(short) i`", where "`i`" is a scalar variable of type "`int`", designates a short integer value). Note that in our language, the spaces of subtype and variable names are disjoints so it is possible to declare "`short short;`".

An example is shown in Table 7.

(1): On ";" the rule engine adds rules:

$$\begin{aligned}
\textit{basic-type-name} &\rightarrow \text{"}\texttt{short}\text{"} && (20)\\
\textit{short-var} &\rightarrow \textit{short-lhs} && (21)\\
\textit{short-var} &\rightarrow \textit{short-fun} && (22)\\
\textit{int-var} &\rightarrow \textit{short-var} && (23)\\
\textit{stmt} &\rightarrow \textit{short-lhs}\ \text{"}\texttt{=}\text{"}\ \textit{short-expr} && (24)\\
\textit{bool-expr} &\rightarrow \textit{short-expr}\ \text{"}\texttt{comp-op}\text{"}\ \textit{short-expr} && (25)\\
\textit{short-expr} &\rightarrow \textit{short-expr}\ \text{"}\texttt{add-op}\text{"}\ \textit{short-expr} && (26)\\
\textit{short-expr} &\rightarrow \textit{short-expr}\ \text{"}\texttt{mul-op}\text{"}\ \textit{short-expr} && (27)\\
\textit{short-expr} &\rightarrow \text{"}\texttt{add-op}\text{"}\ \textit{short-expr} && (28)\\
\textit{short-expr} &\rightarrow \textit{short-primary} && (29)\\
\textit{short-primary} &\rightarrow \textit{int-const} && (30)\\
\textit{short-primary} &\rightarrow \textit{short-var} && (31)
\end{aligned}$$

```
(1)   subtype int short;
(2)   subtype short char;

(3)   int i;
(4)   short short;
(5)   char char, f (short);

(6)   i = char+short;
(7)   short = char+short;
(8)   short = 1+2+3;

(9)   if char!=short
(10)     then char = f ((char)(short)(i)+(char)+char);
(11)     else short = f ((char)i);

(12)  char = i;
```

Table 7: Derived Types an Inheritance.

$$\textit{short-primary} \rightarrow \text{"("}\ \textit{short-expr}\ \text{")"} \tag{32}$$
$$\textit{short-primary} \rightarrow \text{"("}\ \text{"}\texttt{short}\text{"}\ \text{")"}\ \textit{int-primary} \tag{33}$$
$$\textit{int-primary} \rightarrow \text{"("}\ \text{"}\texttt{short}\text{"}\ \text{")"}\ \textit{int-primary} \tag{34}$$

Rule (20) specifies that the type named "`short`" may now be used to declare variables or to be the father type of some other new subtype. Rules (21) and (22) prepare for the declaration of future functions or variables of type "`short`". For the time being, there is no definition for *short-lhs* or *short-fun* since no variable of type `short` has been declared so far. Rule (23) allows for any short variable to occur where an integer variable is valid. Rule (24) defines a new assignment statement between short type. Rule (25) defines comparisons between short types as being boolean expressions. Rules (26) to (29) inherit the arithmetic operators of the father type. Though not shown here, the priority and as-

sociativity of these operators are also inherited. Rule (30) indicates that integer numbers are overloaded and are elements of both types "`int`" and "`short`". Rules (31) and (32) complete the definition of short primaries. Rule (33) defines the mandatory explicit conversion from a type to its subtype whereas rule (34) says that even after an explicit conversion to a short type, the result may be used as an integer primary.

We can remark that the languages of *int-expr* and *short-expr* are not disjoint and, therefore, our language is ambiguous since these two notions may be seeked at several places. For example the overloading of the comparison operator "`!=`" and the overloading of the integer literal constants between short and integer types imply that the boolean expression "`1 != 2`" is ambiguous: is the operator "`!=`" an integer or a short comparator, and if it is an integer comparator, are "`1`" and "`2`" integer or short constants? Of course, the same problem arises with overloaded functions whose profiles only differ between a type and one of its subtype. This type of expressions will result in a quasi-ambiguity detection which is handled by the rule engine (see the processing of line (9)).

(2): On "`;`" the rule engine adds rules:

$$\textit{basic-type-name} \rightarrow \text{"}\texttt{char}\text{"} \quad (35)$$
$$\textit{char-var} \rightarrow \textit{char-lhs} \quad (36)$$
$$\textit{char-var} \rightarrow \textit{char-fun} \quad (37)$$
$$\textit{short-var} \rightarrow \textit{char-var} \quad (38)$$
$$\textit{stmt} \rightarrow \textit{char-lhs}\ \text{"}\texttt{=}\text{"}\ \textit{char-expr} \quad (39)$$
$$\textit{bool-expr} \rightarrow \textit{char-expr}\ \text{"}\texttt{comp-op}\text{"}\ \textit{char-expr} \quad (40)$$
$$\textit{char-expr} \rightarrow \textit{char-expr}\ \text{"}\texttt{add-op}\text{"}\ \textit{char-expr} \quad (41)$$
$$\textit{char-expr} \rightarrow \textit{char-expr}\ \text{"}\texttt{mul-op}\text{"}\ \textit{char-expr} \quad (42)$$
$$\textit{char-expr} \rightarrow \text{"}\texttt{add-op}\text{"}\ \textit{char-expr} \quad (43)$$
$$\textit{char-expr} \rightarrow \textit{char-primary} \quad (44)$$
$$\textit{char-primary} \rightarrow \textit{int-const} \quad (45)$$
$$\textit{char-primary} \rightarrow \textit{char-var} \quad (46)$$
$$\textit{char-primary} \rightarrow \text{"}\texttt{(}\text{"}\ \textit{char-expr}\ \text{"}\texttt{)}\text{"} \quad (47)$$

$$\textit{char-primary} \rightarrow \text{"("}\ \text{"}\texttt{char}\text{"}\ \text{")"}\ \textit{int-primary} \quad (48)$$
$$\textit{int-primary} \rightarrow \text{"("}\ \text{"}\texttt{char}\text{"}\ \text{")"}\ \textit{int-primary} \quad (49)$$
$$\textit{short-primary} \rightarrow \text{"("}\ \text{"}\texttt{char}\text{"}\ \text{")"}\ \textit{int-primary} \quad (50)$$

Rule (38) is the complement of rule (23) for a subsubtype and allows for any char variable to occur where a short variable is valid. Rules (49) and (50) say that even after an explicit conversion to a char type, the result may be used as an integer or short primary. We see that the number of rules to be added each time a subtype is declared is $n + 13$ where n is the distance of the subtype to its base type.

(3):(4):(5): During these declarations the rule engine adds rules:

$$\textit{int-lhs} \rightarrow \text{"}\texttt{i}\text{"} \quad (51)$$
$$\textit{short-lhs} \rightarrow \text{"}\texttt{short}\text{"} \quad (52)$$
$$\textit{char-lhs} \rightarrow \text{"}\texttt{char}\text{"} \quad (53)$$
$$\textit{char-fun} \rightarrow \text{"}\texttt{f}\text{"}\ \text{"("}\ \textit{short-expr}\ \text{")"} \quad (54)$$

Remember that variable and type names are managed in different spaces.

(6): Since "`i`" is an integer variable, the right-hand side of this assignment must be an *int-expr*. The token "`char`" matches the right-hand side of rule (53) and then rules (36), (38) and (23) apply and allow a char variable to be an operand of the integer operator "`+`". An analoguous process happens for "`short`".

(7): Very similar to **(6)** except that "`+`" operates on short types since the left-hand side is "`short`".

(8): Constants "`1`", "`2`" and "`3`" are short primaries (see rule (30)) and the ambiguities in the order of evaluation is resolved by the fact that rule (26) inherits the predefined left associativity of the integer operator "`+`".

(9): "`char`" is initially recognized as a "*char-var*" by the rules (53) and (36). Is this "*char-var*" a "*char-primary*" (rule (46)) or a "*short-var*" (rule (38))?

If "`char`" is a "*short-var*" is it a "*short-primary*" (rule (31)) or an "*int-var*" (rule (23))? A look-ahead of one symbol is not sufficient since the comparison operator "`!=`" is overloaded. Another symbol of look-ahead (i.e. "`short`") only eliminates the possibility for "`char`" to be a "*char-primary*". The next look-ahead symbol which is "`then`", defined in our language as being a key-terminal, allows the detection of a quasi-ambiguity. This non unique identification is reported by the rule engine and the corresponding ambiguity is uniquely resolved by the look-ahead oracle.

(10): Rule (53) is first recognized and a *char-expr* (see rule (39)) is guessed as right-hand side. Rule (54) is then entered and a *short-expr* parameter is expected. The occurrences of "`char`" between parentheses may both designate the subtype or the variable. A look-ahead of two symbols is needed to discover that in the first case it means subtype while in the second it is the variable. For the same reason, "`short`" is found to be a subtype.

(11): In "`(char)i`", the conversion from "`int`" to "`char`" is explicit while the conversion to "`short`", the parameter type, is implicit. In the same way the result type of the function "`f`" is "`char`" which is implicitly converted to "`short`", the type of the left-hand side.

(12): Since there is no automatic conversion from "`int`" to "`char`" (if there was, the grammar would be cyclic) a syntax error is detected on "`i`".

4.6 Polymorphism

The purpose of this paragraph is to show that the notion of dynamic grammars may also be used to process some simple cases of polymorphism. Our language is extended with a *type constructor* named `listof` which defines lists and applies to any (constructed) type, and with two predefined polymorphic functions `cons` and `tail`:

$$\begin{aligned} \texttt{cons} &: \forall\alpha.\ \alpha \times \texttt{listof}(\alpha) \ \mapsto\ \texttt{listof}(\alpha) \\ \texttt{tail} &: \forall\alpha.\ \texttt{listof}(\alpha) \ \mapsto\ \texttt{listof}(\alpha) \end{aligned}$$

An example is given in Table 8.

```
(1)   listof int l;
(2)   listof listof int f2 (listof int), l2;
(3)   listof listof listof int a3 [3];
(4)   int i;

(5)   l = cons (i, ());

(6)   if a3 [1] == cons (f2 (cons (i, ())), ())
(7)      then a3 [1] = cons (cons (l, f2 (l)), a3 [2]);
(8)      else l2 = tail (f2 (tail ((()))));

(9)   a3 [1] = cons (i, l);
```

Table 8: Polymorphism.

(1): On ";" the rule engine adds:

$$\textit{listof}^1\textit{-int} \rightarrow \text{"("}\ \text{")"} \quad (55)$$

$$\textit{listof}^1\textit{-int} \rightarrow \text{"cons"}\ \text{"("}\ \textit{int-expr}\ \text{","}\ \textit{listof}^1\textit{-int}\ \text{")"} \quad (56)$$

$$\textit{listof}^1\textit{-int} \rightarrow \text{"tail"}\ \text{"("}\ \textit{listof}^1\textit{-int}\ \text{")"} \quad (57)$$

$$\textit{listof}^1\textit{-int} \rightarrow \textit{listof}^1\textit{-int-var} \quad (58)$$

$$\textit{listof}^1\textit{-int-var} \rightarrow \textit{listof}^1\textit{-int-lhs} \quad (59)$$

$$\textit{listof}^1\textit{-int-var} \rightarrow \textit{listof}^1\textit{-int-fun} \quad (60)$$

$$\textit{stmt} \rightarrow \textit{listof}^1\textit{-int-lhs}\ \text{"="}\ \textit{listof}^1\textit{-int}\ \text{";"} \quad (61)$$

$$\textit{bool-expr} \rightarrow \textit{listof}^1\textit{-int}\ \text{"comp-op"}\ \textit{listof}^1\textit{-int} \quad (62)$$

$$\textit{listof}^1\textit{-int-lhs} \rightarrow \text{"l"} \quad (63)$$

Rule (55)—(62) are only generated the first time a "`listof` *type*", for a given "*type*"", is found. Rule (55) defines the empty list of integers. Rules (56) and (57) define the fonctions "`cons`" and "`tail`" on "`listof`

`int`". Rules (61) and (62) define possible assignments and comparisons and rule (63) that "`l`" is a scalar variable of that type.

(2): On ";" the rule engine adds:

$$\begin{aligned}
\textit{listof}^2\textit{-int} &\rightarrow \text{"(" ")"} && (64)\\
\textit{listof}^2\textit{-int} &\rightarrow \text{"}\texttt{cons}\text{" "(" } \textit{listof}^1\textit{-int} \text{ "," } \textit{listof}^2\textit{-int} \text{ ")"} && (65)\\
\textit{listof}^2\textit{-int} &\rightarrow \text{"}\texttt{tail}\text{" "(" } \textit{listof}^2\textit{-int} \text{ ")"} && (66)\\
\textit{listof}^2\textit{-int} &\rightarrow \textit{listof}^2\textit{-int-var} && (67)\\
\textit{listof}^2\textit{-int-var} &\rightarrow \textit{listof}^2\textit{-int-lhs} && (68)\\
\textit{listof}^2\textit{-int-var} &\rightarrow \textit{listof}^2\textit{-int-fun} && (69)\\
\textit{stmt} &\rightarrow \textit{listof}^2\textit{-int-lhs} \text{ "=" } \textit{listof}^2\textit{-int} \text{ ";"} && (70)\\
\textit{bool-expr} &\rightarrow \textit{listof}^2\textit{-int} \text{ "}\texttt{comp-op}\text{" } \textit{listof}^2\textit{-int} && (71)\\
\textit{listof}^2\textit{-int-fun} &\rightarrow \text{"}\texttt{f2}\text{" "(" } \textit{listof}^1\textit{-int} \text{ ")"} && (72)\\
\textit{listof}^2\textit{-int-lhs} &\rightarrow \text{"}\texttt{l2}\text{"} && (73)
\end{aligned}$$

Rules (55), (64) and (74) show that empty lists have the same representation. The identification of the type of an empty list may then fail when no outside context is available and when expressions involve only (functions operating on) constants. For example in "`if () == cons ((), ()) then ...`" the operands of the comparison operator "`==`" cannot be uniquely typed and the syntax analysis will result in a quasi-ambiguity detection.

(3): On ";" the rule engine adds:

$$\begin{aligned}
\textit{listof}^3\textit{-int} &\rightarrow \text{"(" ")"} && (74)\\
\textit{listof}^3\textit{-int} &\rightarrow \text{"}\texttt{cons}\text{" "(" } \textit{listof}^2\textit{-int} \text{ "," } \textit{listof}^3\textit{-int} \text{ ")"} && (75)\\
\textit{listof}^3\textit{-int} &\rightarrow \text{"}\texttt{tail}\text{" "(" } \textit{listof}^3\textit{-int} \text{ ")"} && (76)\\
\textit{listof}^3\textit{-int} &\rightarrow \textit{listof}^3\textit{-int-var} && (77)\\
\textit{listof}^3\textit{-int-var} &\rightarrow \textit{listof}^3\textit{-int-lhs} && (78)\\
\textit{listof}^3\textit{-int-var} &\rightarrow \textit{listof}^3\textit{-int-fun} && (79)\\
\textit{stmt} &\rightarrow \textit{listof}^3\textit{-int-lhs} \text{ "=" } \textit{listof}^3\textit{-int} \text{ ";"} && (80)\\
\textit{bool-expr} &\rightarrow \textit{listof}^3\textit{-int} \text{ "}\texttt{comp-op}\text{" } \textit{listof}^3\textit{-int} && (81)
\end{aligned}$$

$$\textit{listof}^3\textit{-int-lhs} \rightarrow \text{"}\texttt{a3}\text{"}\ \text{"}\texttt{[}\text{"}\ \textit{int-expr}\ \text{"}\texttt{]}\text{"} \tag{82}$$

(4): On "`;`" the rule engine adds:

$$\textit{int-lhs} \rightarrow \text{"}\texttt{i}\text{"} \tag{83}$$

(5): Since "`l`" is identified by rule (63), the right-hand side of this assignment is an $\textit{listof}^1\textit{-int}$ (see rule (61)) and hence, the first parameter of "`cons`" must be an *int-expr* while the second must be a $\textit{listof}^1\textit{-int}$. Therefore, the empty list "`()`" is uniquely typed.

(6): Here, elements of "`a3`" are of type $\textit{listof}^3\textit{-int}$ so the first parameter of "`cons`" must be a $\textit{listof}^2\textit{-int}$ and the second a $\textit{listof}^3\textit{-int}$. "`f2`" and "`()`" agree with this typing, and so does the single parameter of "`f2`".

(7):(8): Though theses expressions are not simple, the type checking causes no problem.

(9): The first parameter of "`cons`" should be of type $\textit{listof}^2\textit{-int}$, this is not the case and a syntax error is detected on "`i`".

4.7 Record Types

In Christiansen's paper ([7]), several issues are quoted as being difficult to resolved in the framework of adaptable grammars (its own view of dynamic grammars). One is the problem raised by the "with statement" which gives access to record declared somewhere else without spelling the complete path leading to a field. The following example shows how it is processed with our system. First, our language is extended with the declaration of record types and record variables. A record type may contain scalar, array and record type variables (see lines (1) to (7) in Table 9). The access to a given field is performed by a dot notation (see line (9)) or via a with statement. A with statement is any statement which is prefixed by the header "`with` *rec-name* `do`" where *rec-name* designates a record variable name and which allows to access directly any field of *rec-name*.

The way this problem is processed shows a case where a generated rule (see rule (88)) is itself capable, while it is parsed, to call the rule engine (via parsing actions), and therefore to modify the grammar.

An example is shown Table 9.

```
(1)   record fields {
(2)      int i, j;
(3)   };

(4)   record struct {
(5)      int i, j;
(6)      fields f;
(7)   };

(8)   struct s, a [4], f (fields);

(9)   a [3].f.i = f (s.f).f.j;

(10)  with a [1].f do
(11)     i = j;

(12)  with s do
(13)     with f do
(14)        i = f.j;
```

Table 9: Record Types.

(1): The rule engine adds:

$$\textit{basic-type-name} \rightarrow \texttt{"fields"} \tag{84}$$

$$\textit{fields-var} \rightarrow \textit{fields-lhs} \tag{85}$$

$$\textit{fields-var} \rightarrow \textit{fields-fun} \tag{86}$$

$$\textit{stmt} \rightarrow \textit{fields-lhs}\ \texttt{"="}\ \textit{fields-var} \tag{87}$$

stmt → "`with`" *fields-lhs* "`do`" *@1 stmt @2* (88)

bool-expr → *fields-var* "`comp-op`" *fields-var* (89)

Rule (84) allows for the declaration of record variables and functions whose type name is "`fields`" (see line (6)). In rules (85) and (86) *fields-lhs* and *fields-fun* refer to notions which will (eventually) be defined later on when a variable of that type will be declared (see rule (100)). Rules (87) and (89) allow the assignment and comparisons between "*fields*" objects. Rule (88) defines a with statement. The syntactic actions *@1* and *@2* designate explicit calls to the rule engine. The *@1* call will generate rules wich allow the access to the fields declared in the record body without using a dot notation. These rules are in fact a copy of the rules generated for a record body but only with the last suffix symbol (see for example rules (90), (91) and rules (104), (105)). When the scope of the with statement is exited, the *@2* call deletes the rules generated by the corresponding *@1* call.

(2): We have:

int-lhs → *fields-var* "`.`" "`i`" (90)

int-lhs → *fields-var* "`.`" "`j`" (91)

Rules (90) and (91) allow the access to fields "`i`" and "`j`" of the record type "`fields`".

(4): (5): Are very similar to **(1)** and **(2)**:

basic-type-name → "`struct`" (92)

struct-var → *struct-lhs* (93)

struct-var → *struct-fun* (94)

stmt → *struct-lhs* "`=`" *struct-var* (95)

stmt → "`with`" *struct-lhs* "`do`" *@1 stmt @2* (96)

bool-expr → *struct-var* "`comp-op`" *struct-var* (97)

int-lhs → *struct-var* "`.`" "`i`" (98)

int-lhs → *struct-var* "`.`" "`j`" (99)

(6): We have:

$$\textit{fields-lhs} \rightarrow \textit{struct-var}\ \text{"}\texttt{.}\text{"}\ \text{"}\texttt{f}\text{"} \tag{100}$$

The rule (100) allows the access to the field named "`f`" in a variable whose type name is "`struct`", this field being itself of record type "`fields`" and can be used in the left-hand side of an assignment statement.

(8): The rule engine adds:

$$\textit{struct-lhs} \rightarrow \text{"}\texttt{s}\text{"} \tag{101}$$
$$\textit{struct-lhs} \rightarrow \text{"}\texttt{a}\text{"}\ \text{"}\texttt{[}\text{"}\ \textit{int-expr}\ \text{"}\texttt{]}\text{"} \tag{102}$$
$$\textit{struct-fun} \rightarrow \text{"}\texttt{f}\text{"}\ \text{"}\texttt{(}\text{"}\ \textit{fields-var}\ \text{"}\texttt{)}\text{"} \tag{103}$$

Remark that rule (103) prohibits the usage of a record object returned by a function call to be used as left-hand side of an assignment statement or as header of a with statement.

(9): Here "`a [3]`" is recognized by rule (102), then rule (93) is used, the first "`.`" is the middle terminal symbol in rules (98), (99) or (100), the "`f`" selects rule (100). *fields-lhs* is reduced to *fields-var* by rule (85) and finally "`.i`" is recognized by the suffix of rule (90). Therefore the right-hand side of this assignment statement must be an *int-expr*. "`s.f`" is recognized as a *fields-var* by the sequence of reductions: (101), (93), (100) and (85). Rules (103) and (94) indicate that "`f (s.f)`" is a *struct-var* and finally "`f (s.f).f.j`" is recognized as an *int-lhs* by the reduction sequence (100), (85) and (91).

(10): Here "`a [1].f`" is recognized as a *fields-lhs* by rule (100) and "`with a [1].f do`", matches the beginning of rule (88). The rules engine, called via the syntactic action *@1*, adds the rules:

$$\textit{int-lhs} \rightarrow \text{"}\texttt{i}\text{"} \tag{104}$$
$$\textit{int-lhs} \rightarrow \text{"}\texttt{j}\text{"} \tag{105}$$

which are similar with rules (90) and (91) where the field access mecanism has been erased.

(11): Here "`i`" and "`j`" are recognized by rules (104) and (105). When the with statement is exited, rules (104) and (105) are deleted by the call of the syntactic action *@2*.

(12): The "`s`" is recognized as a *struct-lhs* by rule (101) and the call of *@1* in rule (96) adds:

$$\textit{int-lhs} \rightarrow \text{"}\texttt{i}\text{"} \quad (106)$$
$$\textit{int-lhs} \rightarrow \text{"}\texttt{j}\text{"} \quad (107)$$
$$\textit{fields-lhs} \rightarrow \text{"}\texttt{f}\text{"} \quad (108)$$

(13): The "`f`" is recognized as a *fields-lhs* by rule (108) and the call of *@1* in rule (88), as already explained in paragraph 4.1, deletes the rules (106) and (107), and adds:

$$\textit{int-lhs} \rightarrow \text{"}\texttt{i}\text{"} \quad (109)$$
$$\textit{int-lhs} \rightarrow \text{"}\texttt{j}\text{"} \quad (110)$$

(14): Here "`i`" designates "`s.f.i`" and "`f.j`" stands for "`s.f.j`". Though the inner with statement gives access to the field names of "`s.f`", these names can still be denoted by a dot mecanism. In fact, at that position, "`s.f.j`", "`f.j`" and "`j`" designate the same variable.

When the inner with statement is exited, the call of *@2* in rule (88) deletes the rules (109) and (110), and adds the rules (106) and (107). When the outer with statement is exited, rules (106), (107) and (108) are deleted by the call of *@2* in rule (96).

4.8 Forward References

This example illustrates the possibility to have forward information which influence the current parsing. Labels and "`goto`" statements are added to our language with the following meaning: `goto`'s cannot enter inner blocks. Labels are block-structured with at most one label name at a given level. It means that if a label in an inner block hides an outer label, "`goto`"'s in the inner block to that name refer to the inner label, even when this label occurs in the

source text after the "`goto`". Such a case is processed in using the multi-pass possibility of our implementation. A first pass is performed over the source text with a first (here static) grammar and the rule engine collects the information about the fact that a given label occurs in a given block. The information about labels, defined on each block, will be used during the second pass to generate the rules for labeled and goto statements. In fact, during this second pass, the dynamic parser reacts as if the predeclaration of labels at each block level were mandatory.

An example is shown Table 10.

```
(1)    {
(2)       goto A;

(3)       {
(4)          goto A;
(5)          A: goto B;
(6)       }

(7)       A:;
(8)    }

(9)    B:goto A;
```

Table 10: Forward References.

(1): At block level #0, the rule engine adds:

$$\textit{label} \rightarrow \text{"B" ":"} \tag{111}$$

$$\textit{stmt} \rightarrow \text{"goto" "B" ";"} \tag{112}$$

where "`B`" designates the label at line (9). Note that goto statements are not predefined in the initial grammar, they are generated (with references

to good labels) only where the are allowed to be used. The initial grammar only recorded the fact that statements may be labelled, it contains the rules:

$$\textit{label-stmt} \rightarrow \textit{label @3 label-stmt} \quad (113)$$

$$\textit{label-stmt} \rightarrow \textit{stmt} \quad (114)$$

where *@3* will delete the rule which has just produced the non-terminal *label* occurring just before *@3*. (For an example see the processing of line (5).)

When entering the first block, the rule engine adds:

$$\textit{label} \rightarrow \texttt{"A" ":"} \quad (115)$$

$$\textit{stmt} \rightarrow \texttt{"goto" "A" ";"} \quad (116)$$

where "A" designates the label at line (7).

(2): This goto statement is recognized by rule (116).

(3): When entering the second block, the rule engine deletes rules (115) and (116) since a new label with the same name "A" will be defined at that block level, and adds:

$$\textit{label} \rightarrow \texttt{"A" ":"} \quad (117)$$

$$\textit{stmt} \rightarrow \texttt{"goto" "A" ";"} \quad (118)$$

where "A" designates the label at line (5).

(4): This goto statement is recognized by rule (118).

(5): The label "A" is the one define by rule (117), which, when recognized, is immediatly deleted by the rule engine via a call to *@3*. This prevents another use of the same label in the same block. This goto statement is recognized by rule (112).

(6): At block exit, rule (118) is deleted and rules (115) and (116) are added. Note that rule (117) has already been erased at line (5) since the same label name cannot occur twice at the same block level.

(7): The label "`A`" is the one define by rule (115). This rule is immediately deleted by the rule engine via a call to *@3*.

(8): At block exit, rule (116) is deleted.

(9): The label "`B`" is the one define by rule (111) which is deleted by the rule engine. But the only goto statement which is (still) valid is defined by rule (112) and therefore a syntax error is detected on "`A`".

4.9 Permutation Phrases

Though not in the domain of semantic analysis, permutation phrases are constructions frequently used in languages and which are not suitably processed by context-free grammars. A *permutation phrase* refer to any permutation of a set of constituent elements. Among others, such free-order constructs can be found in, attribute specifiers within C or C++ declarations, named parameter associations in ADA subprogram calls or generic instantiations, various COBOL constructs, citation fields in BibTeX bibliographies, *and-group* in the SGML language for document representation, and command-line parameters or options of various Unix programs. This free-word-order phenomenon also occurs prominently in natural languages processing. A permutation phrase of n constituent elements necessitates $n!$ context-free rules to be described. We shall see that this problem is solved by dynamic grammars with a linear complexity. To be more realistic we will assume that some elements should occur in any given permutation while some others are optional. Let M and O be a partition of E, the set of constituent elements, where M stands for mandatory and O for optional, and $p = |M|$, $q = |O|$ and $n = |E|$ be their cardinals ($p + q = n$). A permutation phrase PP (among others) may be described by the following (initial) grammar:

$$PP \rightarrow PP^p \tag{119}$$

$$PP^p \rightarrow PP^{p-1}\ M \tag{120}$$

$$PP^p \rightarrow PP^p\ O \tag{121}$$

$$\vdots$$

$$
\begin{aligned}
PP^{i} &\rightarrow PP^{i-1}\ M && (122)\\
PP^{i} &\rightarrow PP^{i}\ O && (123)\\
&\vdots\\
PP^{1} &\rightarrow PP^{0}\ M && (124)\\
PP^{1} &\rightarrow PP^{1}\ O && (125)\\
\\
PP^{0} &\rightarrow PP^{0}\ O && (126)\\
PP^{0} &\rightarrow \epsilon && (127)\\
\\
M &\rightarrow m_{p} \mid \ldots \mid m_{j} \mid \ldots \mid m_{1} && (128)\\
\\
O &\rightarrow o_{q} \mid \ldots \mid o_{k} \mid \ldots \mid o_{1} && (129)
\end{aligned}
$$

Rule (119) indicates that any permutation contains the p mandatory elements. The rules such (120), ..., (122), ..., (124) define a permutation of i elements as being formed by a permutation of $i-1$ elements followed by a mandatory element, while rules (121), ..., (123), ..., (125) say that such a permutation may also be composed by a permutation of i elements followed by any number of optional elements. Rules (126) and (127) define (initial) permutations only formed by optional elements. Rules (128) and (129) respectively define the mandatory and optional elements. But, as such, this grammar does not describe a permutation phrase since a given element may occur several times in a given phrase. The dynamic grammar mecanism can handle this difficulty. Each time the rule engine is called after the recognition of an element (rule (128) or (129)), this rule is deleted and therefore prohibits any further recognition of the same element within the same permutation. The only thing to do, on rule (119), is to restore the rules which have been deleted in order to allow the parsing of other permutation phrases. We see that the resolution of permutation phrases is easily performed, without any restriction, within the framework of dynamic grammars. This should be contrasted with systems (see [6]), specially dedicated to that purpose, and which impose some (severe) restrictions.

5 Other Works

Our work has been initiated by the papers of Burshteyn ([2], [3] and [4]), Cabasino, Paolucci and Todesco ([5]), and Christiansen ([7]).

In ([2]), Burshteyn introduces the notion of *modifiable grammar* and some of their applications. This notion is formalized and the parsing algorithm is presented in ([3]). The name "*rule engine*" and the idea and proof of theorem 1 come from ([3]). This system is based upon a (dynamically produced) LR(1) automaton, which is, in our opinion, inappropriate for at least two reasons: grammars should have as few constrainst as possible, since they are dynamically produced (for a real programming languages, to get a static LR(1) grammar is not so easy so imagine the mess with a dynamic one ...) and the status of this look-ahead symbol seems unclear. In fact, a method whose basic parsing algorithm uses some look-ahead does not seem to fit our goal: the grammar for the look-ahead may be unavailable (not yet specified), or may be modified (the vocabulary of the look-ahead string may even not belong to the current grammar). Moreover, the restrictions upon modifications are not clearly stated. Finally, in ([4]), Burshteyn describes a tool which accepts definitions written in a metalanguage and gives several examples from C++. It should be noted that a multipass mecanism is also available in modifiable grammars.

The notion of *evolving grammars* is defined in ([5]). This formalism only allows a grammar to grow (i.e. new grammar rules can only be added) and though we have seen in the proof of theorem 1 that the formal descriptive power is not decreased by this restriction, an order of magnitude can be lost in the practical usage of such grammars. The authors themselves admit that "The current conceptual framework shows several limitations in the area of "negative grammar changes". Therefore we hardly face problems connected with syntactic scope". Moreover their parsing algorithm is not explicitly described.

In ([7]), Christiansen defines *generative clause grammars* which are an extension of *definite clause grammars* (see [8] and [12]). In Section 2, we can find a short survey of approaches to extensible or adaptable grammar formalisms, and Section 3 discusses more problematic issues with this approach and the way they have been solved (or not). Its very declarative and functional style of adaptable grammars is appealing but does not seem to be capable of the

kind of fine-tuning needed to describe the wide variety of context-dependent rules we can find in programming languages. On the other hand, the power of our dynamic grammars partly relies upon the rule engine which is *the* grammar definition of the language to be processed. The only output interface with the dynamic parser are (only) context-free grammars but the way they are generated is not part of the formalism. For example all the features (except for the error handling which has not yet been considered in our system — see section 6) quoted as difficult in ([7]) (and some others) have been processed (see section 4) by our system.

A lazy and incremental parser generator has been studied in ([13]) in the context of interactive language definition environments. In this thesis, Rekers describes an LR-based parser generator for arbitrary context-free grammars that generates parsers on demand and which handles modifications to its input grammar by updating the parser it has generated so far. In ([10] and [11]) Horspool studied an incremental LALR(1) parsing generation method.

6 Conclusion and Future Works

We have described the theoretical basis for dynamic grammars and their corresponding dynamic parsers. We also have presented both the rule engine, which is the operational view of a dynamic grammar, and an experimental implementation of a dynamic parser which consists of, a lazy, incremental LR(0) parser equiped with an unbounded look-ahead oracle. The intend of this system was to demonstrate the practical capabilities of this formalism in the processing of programming languages. Though our system accepts, with only few restrictions, any dynamic grammar, its global behavior is non ambiguous in order to cope with its intend application field. This means that, for any input string, at most a unique parse is produced, even when the dynamic grammar is ambiguous. The dynamic parser and the rule engine are coupled via syntactic action and predicate calls, and the consistency of any grammar modification with the theory is checked (see Table 2). This experimental system has been used on a sample of significative examples which demonstrate that this method should be considered as a viable alternative solution to usual methods, as far as context-sensitive properties are concerned.

Of course, dynamic grammars are a syntactic device and, as such, are not well suited to perform computations (the output is formally defined as being a dynamic rightmost parse). So we do not claim that they can replace such devices as attribute grammars, but they can be used when an attribute grammar would only perform context-sensitive checking (typically type-checking). Other more strictly semantic parts fall more naturally within the domain of other methods.

A lot of related problems should still be investigated. What is the compared performance (for both a theoritical and practical point of view) of a dynamic parser versus a more usual method ? What kind of error recovery and repair strategies will work with dynamic grammars ? In fact an improvement of context-free methods is probably to be expected since much more information is available. Any correction will both be syntactically valid and typed-checked. It also seems possible to extend the spelling correction mecanisms from keywords to mere (applied occurrences of) identiers. As pointed out in ([4]), dynamic grammars may be used as a generator device (instead of a parser) and then produce texts which are (by nature) fully type-checked. It should be noted that an attribute grammar which describes a given language cannot be used as such to produce a test suite. If it appears that dynamic grammars constitute a valuable practical concept, the way the rule engine is actually written should be improved in a much more descriptive manner. A semantic attribute based system such as the one in ([4]) could be thought of. The same ideas have been used to implement a *dynamic scanner* but its integration into a single system is not yet done.

We suspect that this unifying framework will facilitate the emergence of other applications.

References

[1] Aho, A. V., Sethi R., and Ullman J. D.
"Compilers, Principles, Techniques and Tools".
Addison-Wesley, (1986).

[2] Burshteyn, B.
"On the Modification of the Formal Grammar at Parse Time".

ACM SIGPLAN Notices. 25, 5 (May. 1990), 117-123.

[3] Burshteyn, B.
"Generation and Recognition of Formal languages by Modifiable Grammars".
ACM SIGPLAN Notices. 25, 12 (Dec. 1990), 45-53.

[4] Burshteyn, B.
"USSA - Universal Syntax and Semantics Analyzer".
ACM SIGPLAN Notices. 27, 1 (Jan. 1992), 42-60.

[5] Cabasino, B., Paolucci, P. S., and Todesco, G. M.
"Dynamic Parsers and Evolving Grammars".
ACM SIGPLAN Notices. 27, 11 (Nov. 1992), 39-48.

[6] Cameron, B. D.
"Extending Context-Free Grammars with Permutation Phrases".
Technical Report CMPT TR 92-07, Simon Fraser University, Burnaby, B.C., Canada. (1992), 1-10.

[7] Christiansen, H.
"A Survey of Adaptable Grammars".
ACM SIGPLAN Notices. 25, 11 (Nov. 1990), 35-44.

[8] Colmerauer, A.
"Metamorphosis grammars".
Springer-Verlag, Lecture Notes in Computer Science, 63, (1978), 133-189.

[9] Čulik, K., and Cohen, R.
"LR-Regular Grammars—an Extension of LR(k) Grammars".
Journal of Computer and System Sciences, 7, (1973), 66-96.

[10] Horspool, R. N.
"ILALR: an incremental generator of LALR(1) parsers".
Compiler Compilers and high speed Compilation,
Springer-Verlag, Lecture Notes in Computer Science, 371, (1989), 128-136.

[11] HORSPOOL, R. N.
"Incremental generation of LR parsers".
Computer languages, 15(4), (1990), 205-233.

[12] PEREIRA, F. C. N., and WARREN, D. H. D.
"Definite clause grammars for language analysis—A survey of the formalism and a comparison with augmented transition networks".
Artificial Intelligence, 13, (1980), 231-278.

[13] REKERS, J.
"Parser Generation for Interactive Environments".
Ph.D. Thesis, (Jan. 1992).

[14] TOMITA, M.
"Efficient Parsing for Natural Languages".
Kluwer Academic Publishers, Boston, MA, (1986).

[15] TOMITA, M.
"An efficient augmented-context-free parsing algorithm".
Computational Linguistics, 13, no. 1-2, (Jan. 1987), 31-46.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249- 6399