VICTORIA UNIVERSITY OF WELLINGTON

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

Deriving a Predictive Parsing Algorithm

Lindsay Groves*

Technical Report CS-TR-94/22
December 1994

Abstract

We show how a predictive parsing algorithm can be derived by first deriving a generic language recognition algorithm, and then specialising this to a class of context-free languages. This strategy allows us to work out the basic structure of the algorithm, based on very general properties of the specification, and then develop more intricate parts of the algorithm and introduce suitable data representations afterwards. The resulting derivation is more intelligible than if we considered all aspects of the specification from the outset, and allows the derivation to be more easily adapted to handle other classes of languages.

Publishing Information

Appeared in: Steve Reeves (Ed), *Proceedings of the First New Zealand Formal Program Development Colloquium*, Waikato University, November 1994, pp 91–103.

*Email: `lindsay@comp.vuw.ac.nz`.

1 Introduction

One of the major problems in software development, formal or otherwise, is that of controlling complexity. In order to make formal software development feasible for anything other than toy problems, we need to find ways of structuring the development so as to control complexity.

One strategy for controlling complexity in software development is the use of data abstraction. We initially ignore details of data representation and derive an abstract program which does not depend on any particular data representation; we then introduce the details of a particular data representation, and perhaps make various optimisations admitted by the chosen representation. This process has been studied extensively and has become a major part of modern software design methods. The use of data abstraction has been formalised within the refinement calculus as *data refinement* (e.g. [Morgan and Gardiner, 1988], [Morris, 1989]).

Another strategy for controlling complexity is to begin by considering a *generalisation* of the problem, obtained by ignoring some of the requirements; in particular we can ignore some of the assumptions about the inputs. We can then develop software for the generalised problem, and later *specialise* it to address the omitted requirements. Making use of these additional assumptions often allows the software to be made more efficient — in particular it may allow data refinements that were not previously possible — and may make it feasible.

The use of program specialisation in the context of program refinement has been described by Gravell [Gravell, 1991], who shows that a number of different search algorithms and some arithmetic algorithms can be obtained by specialising a generic search algorithm. In [Groves, 1994a], we illustrated the use of program specialisation in tandem with data refinement to derive a family of language recognition algorithms by deriving a generic language recognition algorithm, and then specialising it obtain algorithms for some specific classes of languages.

In this paper we again derive a generic language recognition, then specialise it to obtain a table-driven predictive parsing algorithm for a class of context free languages. The generic algorithm differs from the one in [Groves, 1994a] in a number of ways: the notation used and the derivation are (hopefully) more elegant, and the resulting algorithm is more general. The derivation of the predictive parsing algorithm is presented in more detail than in [Groves, 1994a], identifying more carefully the properties on which the algorithm depends.

We begin by formulating the language recognition problem in a very general way (Section 2) and deriving a generic language recognition algorithm (Section 3). In Section 4 we specialise the algorithm to recognise a class of context free languages and introduce suitable data refinements to obtain a concrete algorithm. In Section 5 we present our conclusions.

The paper assumes some familiarity with the refinement calculus, as given, for example in [Morgan, 1990] or [Groves, 1994b]. We also assume standard terminology and definitions from language theory (see, for example, [Hopcroft and Ullman, 1979]).

2 Defining the problem

We begin by considering the general problem of recognising sentences in a given language, making no assumptions about the kind of language being recognised. We will derive a generic algorithm for this generalised version of the problem, and then specialise it to a particular class of languages. The generic algorithm, however, provides the basic structure of the specialised algorithm, and can be similarly specialised to provide recognisers for other classes of languages.

We initially assume only that we are given some alphabet $\mathcal{A}$ (i.e. a finite set of symbols), a string s over $\mathcal{A}$ and a language L over $\mathcal{A}$, and we require to determine whether s is a sentence in L. We can formalise this as a specification statement [Morgan, 1990]. A specification statement $\tilde{w}\colon \left[\mathit{Pre} \;\middle/\; \mathit{Post} \right]$ will establish the postcondition *Post* altering only variables in $\tilde{w}$, provided the precondition *Pre* is true initially. If we return the result as a boolean variable, r, we can express our specification as:

$$r\colon \left[s \in \mathcal{A}^* \ \wedge\ L \subseteq \mathcal{A}^* \;\middle/\; r \equiv s \in L \right] \tag{1}$$

We will assume that string comparison is not a primitive operation, so we are not able to simply compare s with each string in L — in any case, L may not be finite. Instead, we will inspect one symbol of s at a time, and use that to successively narrow down the part of L in which s could occur. We will follow standard convention and inspect symbols in s from left to right, though other choices are certainly possible.

3 Deriving a generic language recognition algorithm

We will now proceed to derive a generic language recognition algorithm to implement the above specification. In doing the derivation, we will need to introduce a number of concepts and notations relating to strings and languages, and give properties required in deriving the algorithm. We introduce some of these now, and present others as they are needed, so their introduction is clearly motivated.

Since our algorithm will consider prefixes of s, we need notation for the prefix relation; we also introduce a suffix relation here, since it will be required later. We write $p \preceq s$ to mean that p is a *prefix* of s, and $s \succeq p$ to mean that p is a *suffix* of s, defined as follows:

Definition $p \preceq s \ \widehat{=}\ (\exists u \bullet p \frown u = s)$
$s \succeq p \ \widehat{=}\ (\exists u \bullet u \frown p = s)$

As indicated above, the algorithm will look at symbols in s one at a time, from left to right. We observe that each symbol of s inspected effectively reduces the part of L in which s could occur. We capture this idea by introducing a function d which gives the *derivative* of a language with respect to a string[1], defined thus:

Definition For any string u and language L, the *derivative* of L with respect to u, written $d(u, L)$, is the set of all strings which, when right concatentated with u, give a sentence in L; i.e. $d(u, L) \ \widehat{=}\ \{ v \mid u \frown v \in L \}$, where $\frown$ is the string concatenation operator.

For the purposes of the abstract algorithm, we will consider d to be executable only when u is a singleton (i.e. $|u| = 1$).

The derivative operation has a number of important properties. The following theorem captures an essential relationship between prefixes and deriviatives.

Theorem For any strings u and v, $u \frown v$ is in L iff v is in $d(u, L)$; i.e. $u \frown v \in L \equiv v \in d(u, L)$.

This theorem has two important consequences:

Corollary 1 Taking $u = s$ and $v = \lambda$ (where λ is the empty string), we see that $s \in L$ is equivalent to $\lambda \in d(s, L)$.

Corollary 2 If $d(p, L) = \varnothing$ for any prefix p of s, then $d(q, L) = \varnothing$ for any longer prefix q of s. In particular, $d(s, L) = \varnothing$; i.e. $p \preceq s \wedge d(p, L) = \varnothing \Rightarrow d(s, L) = \varnothing$.

[1] This notion is motivated by the idea of derivatives of regular expressions [Brzozowski, 1964].

The first corollary shows that we can determine whether s is in L by computing $d(s, L)$ and checking whether it contains λ. The second shows that s is not in L if $d(p, L)$ is empty for any prefix p of s. Thus, once $d(p, L)$ is found to be empty, for some prefix p, there is no point in looking at further elements of s.

Together, these corollaries suggest that we can determine whether s is in L by looking for a prefix p of s such that either $\lambda \in d(p, L)$ or $d(p, L) = \varnothing$[2]. Then, $s \in L$ iff $p = s$ and $\lambda \in d(p, L)$.

In fact, it turns out to be easier to consider progressively shorter suffixes of s. If t is a suffix of s, we write $s - t$ to mean the prefix of s remaining after suffix t is removed:

Definition For all strings u and v, $(u \frown v) - v = u$.

We can now substitute $s - t$ for p above[3]. If t is a suffix of s such that either $\lambda \in d(s-t, L)$ or $d(s - t, L) = \varnothing$, then $s \in L$ iff $t = \lambda$ and $\lambda \in d(s - t, L)$.

We can now refine the specification to introduce a variable t, for suffixes of s; we also introduce a variable C, for values of $d(s - t, L)$. At the same time, we split the specification statement into two parts: one to find a suffix t satisfying the above condition, and the other using this to determine the value of r. We will also omit the precondition and treat it a global assumption.

$$
\begin{array}{lll}
(1) \sqsubseteq & (\textit{Introduce Local Variables, Split Specification, Contract Frame}) & \\
& \mathbf{var}\ t, C \bullet & \\
& t, C\colon \left[s \succeq t \wedge C = d(s - t, L) \wedge (t = \lambda \vee C = \varnothing) \right]; & (2) \\
& r\colon \left[s \succeq t \wedge C = d(s - t, L) \wedge (t = \lambda \vee C = \varnothing) \;\middle/\; r \equiv s \in L \right] & (3)
\end{array}
$$

We can immediately refine (3) to an assignment:

$$
\begin{array}{ll}
(3) \sqsubseteq & (\textit{Introduce Assignment}) \\
& r := (t = \lambda \wedge \lambda \in C)
\end{array}
$$

The following proof obligation follows from corollaries 1 and 2:

$$s \succeq t \wedge C = d(s - t, L) \wedge (t = \lambda \vee C = \varnothing) \Rrightarrow ((t = \lambda \wedge \lambda \in C) \equiv s \in L)$$

3.1 Loop design

We now want to refine (2) to introduce a loop which computes $d(s - t, L)$ for decreasing suffixes t of s. Thus, we will take as our loop invariant:

$$Inv \ \widehat{=}\ s \succeq t \ \wedge\ C = d(s - t, L)$$

The postcondition of (2) suggests $t \neq \lambda \wedge C \neq \varnothing$ as the loop guard, and $|t|$ as the variant function.

We split (2) into two parts, one of which will initialise t and C so as to establish I, the other will be the loop.

[2] We could insist that the program find the shortest such prefix, but it is not necessary to do so; our algorithm will in fact find the shortest such prefix.

[3] This could be treated formally as a data refinement, but there seems little point in doing so.

(2) $\sqsubseteq$ (*Split Specification*)

$$t, C\colon \Big[\, Inv \,\Big]\,; \tag{4}$$

$$t, C\colon \Big[\, Inv \;\Big/\; Inv \wedge (t = \lambda \vee C = \varnothing) \Big] \tag{5}$$

We can easily establish *Inv* by establishing $t = s \;\wedge\; C = L$, since $s \succeq s$ and $L = d(\lambda, L)$.

(4) $\sqsubseteq$ (*Introduce Assignment*)

$t, C := s, L$

We now introduce the loop:

(5) $\sqsubseteq$ (*Introduce DO*)

do $t \neq \lambda \wedge C \neq \varnothing \rightarrow$

$$t, C\colon \Big[\, Inv \wedge t \neq \lambda \wedge C \neq \varnothing \;\Big/\; Inv \wedge 0 \leq |t| \leq |t_0| \,\Big] \tag{6}$$

od

We further split (6) into two parts: one will update t to ensure that the loop makes progress; the other will update C to ensure that the invariant is maintained.

In order to "make progress" towards termination, the loop body must decrease $|t|$. Since t must always be a suffix of s, this can only be done by removing some prefix of t. In particular, we can decrease $|t|$ by one, by removing its head, i.e. a prefix of length one. We will assume the availability of two (executable) functions hd and tl such that, for any non-empty string t, $t = \mathsf{hd}\,t \frown \mathsf{tl}\,t$ and $|\mathsf{hd}\,t| = 1$. Thus, we wish to remove $\mathsf{hd}\,t$ from t, which can be done using the assignment statement $t := \mathsf{tl}\,t$.

We must now determine how to update C so as to maintain the invariant when t is updated like this. To see how C should be updated, we will refine (6) using the *Following Assignment* rule, and calculating the intermediate assertion:

$$\begin{aligned} & wp(t := \mathsf{tl}\,t,\ s \succeq t \;\wedge\; C = d(s - t, L) \wedge 0 \leq |t| < |t_0|\,) \\ & \quad = s \succeq \mathsf{tl}\,t \;\wedge\; C = d(s - \mathsf{tl}\,t, L) \wedge 0 < |\mathsf{tl}\,t| \leq |t| \end{aligned}$$

We can drop the termination condition, since $t \neq \lambda \Rightarrow 0 \leq |\mathsf{tl}\,t| < |t|$ and $t \neq \lambda$ is given by the precondition.

We can also eliminate the condition $s \succeq \mathsf{tl}\,t$, which follows easily from the definitions:

$$\begin{aligned} & Inv \wedge t \neq \lambda \wedge C \neq \varnothing \\ & \quad \Rightarrow\ s \succeq t \wedge t \neq \lambda \\ & \quad \Rightarrow\ (\exists p \;\bullet\; s = p \frown t) \wedge t \neq \lambda \\ & \quad \Rightarrow\ (\exists p \;\bullet\; s = p \frown \mathsf{hd}\,t \frown \mathsf{tl}\,t) \\ & \quad \Rightarrow\ (\exists p' \;\bullet\; s = p' \frown \mathsf{tl}\,t) \\ & \quad \Rightarrow\ s \succeq \mathsf{tl}\,t \end{aligned}$$

This just leaves $C = d(s - \mathsf{tl}\,t, L)$ as the intermediate assertion:

(6) $\sqsubseteq$ (*Following Assignment*, *Contract Frame*)

$$C\colon \Big[\, Inv \wedge t \neq \lambda \wedge C \neq \varnothing \;\Big/\; C = d(s - \mathsf{tl}\,t, L) \Big]\,; \tag{7}$$

$t := \mathsf{tl}\,t$

In order to refine (7), we observe that $s - \mathsf{tl}\, t = (s - t) \frown \mathsf{hd}\, t$, and $d(x \frown y,\ L) = d(y,\ d(x,\ L))$, for all strings x and y, and any language L.

Thus, we get:

$$
\begin{aligned}
C &= d(s - \mathsf{tl}\, t, L) \\
&= d((s - t) \frown \mathsf{hd}\, t, L) \\
&= d(\mathsf{hd}\, t, d(s - t, L))
\end{aligned}
$$

Since we have $C = d(s - t, L)$ in the precondition, we can refine (7) as follows:

$$
\begin{aligned}
(7) \quad \sqsubseteq \quad & (\textit{Introduce Assignment}) \\
& C := d(\mathsf{hd}\, t, C)
\end{aligned}
$$

The algorithm at this stage is:

$$
\begin{array}{l}
[\![\ \textbf{var}\ t, C \bullet \\
\quad t, C := s, L; \\
\quad \textbf{do}\ t \neq \lambda \wedge C \neq \varnothing \rightarrow \\
\quad\quad C := d(\mathsf{hd}\, t, C); \\
\quad\quad t := \mathsf{hd}\, t \\
\quad \textbf{od}; \\
\quad r := (t = \lambda \wedge \lambda \in C) \\
]\!]
\end{array}
$$

This algorithm is still not really executable because it is expressed in terms of the set variables L and C, and the function d which cannot be implemented for arbitrary sets. In order to obtain an executable program, we will have to place some restriction on L, and then explore suitable representations for L and C, and corresponding implementations for d. Obviously we won't be able to turn this into a concrete algorithm for all Ls: If L is not decidable, we won't be able to complete the data refinement.

4 Recognising Context Free Languages

In order to turn the above abstract algorithm into a more concrete form, we need to find some way to describe L — restricting the original specification to suit a particular descriptive formalism if necessary. Here we will consider the case where L is a non-empty context free language. In this case, we know that L can be described using a context free grammar, G, where G is a 4-tuple (V_N, V_T, S, P) giving the nonterminals, terminals, start symbol and productions, respectively (see, for example, [Hopcroft and Ullman, 1979]); since $L \subseteq \mathcal{A}^*$, we can assume $V_T = \mathcal{A}$. We assume that G is a "proper" grammar (i.e. all nonterminals are defined and there are no circular rules), and that G has no λ-rules (i.e. there are no rules with empty right-hand sides).

We specialise the initial specification with the additional assumption $L \neq \{\} \wedge L = \mathcal{L}(G)$, where $\mathcal{L}(G)$ is the language generated by G, defined as follows:

$$
\mathcal{L}(G) \ \widehat{=}\ \{\ w \in V_T^* \ \mid\ S \ \Rightarrow_G^* \ w\ \}
$$

and $\Rightarrow_G^*$ is the usual "produces" relation.

To specialise a program, we introduce additional assumptions about the inputs, and propagate these through the resulting derivation. This process is justified, since we can show that if $w : [P, R] \sqsubseteq w' : [P', R']$ then, for any additional assumptions A, we have $w : [P \wedge A, R] \sqsubseteq w' : [P' \wedge A, R']$.

4.1 Representing derivatives

Having assumed that L is context free, we must also find a way to represent C, such that we can readily implement the required operations, especially $d(\mathsf{hd}\, t, C)$. For any context free language L and any string s, the derivative $d(x, L)$ is also context free. Thus, we could describe C using a context free grammar. In this case, the implementation of $d(\mathsf{hd}\, t, C)$ would need to construct a new grammar; in particular, it would need to introduce new nonterminals and new rules, which promises to be rather messy.

An alternative approach is suggested by considering sentential forms, which can also be thought of as describing languages: perhaps we can represent C using a string of terminal and/or nonterminal symbols. It is not immediately clear whether we can always represent C in this way (it may depend on how we construct the algorithm), but it seems quite reasonable. So we will pursue this approach and see where it leads. We will ignore, for now, the question of how to represent the set of production rules, P.

To allow for the possibility that $C = \varnothing$, we introduce a special symbol ϕ, not in V_N or V_T, to cover this case (c.f. the use of ϕ in regular expressions). For any string $\sigma \in V^*$, where $V \mathrel{\widehat=} V_N \cup V_T \cup \{\phi\}$, we define $\mathcal{L}(\sigma)$ to be the set of all terminal strings that can be produced from σ:

$$\mathcal{L}(\sigma) \mathrel{\widehat=} \{ w \in V_T^* \mid \sigma \Rightarrow_G^* w \}$$

Note that $\mathcal{L}(\lambda) = \{\lambda\}$, while $\mathcal{L}(\phi) = \varnothing$.

Clearly, L can be represented by the start symbol, S. We will endeavor to represent C by some string σ over V such that $C = \mathcal{L}(\sigma)$.

4.2 Data refinement

We can now data refine the generic algorithm using this data representation. Our coupling invariant will be:

$$CI \mathrel{\widehat=} L = \mathcal{L}(S) \wedge C = \mathcal{L}(\sigma) \wedge \sigma \in V^*$$

To perform the data refinement, we replace the declaration of C by a declaration of σ, then data refine the body. Since the coupling invariant is functional, this essentially involves replacing occurrences of L and C by $\mathcal{L}(S)$ and $\mathcal{L}(\sigma)$, respectively, and appending the representation invariant $\sigma \in V^*$ as appropriate ([Morgan and Gardiner, 1988], [Morris, 1989]). We then perform further refinements to remove occurrences of $\mathcal{L}$, since it is not executable. To simplify the presentation, we will treat $\sigma \in V^*$ as a global invariant and not mention it explicitly.

Data refining the body involves the following steps:

- Data refine the assignment $C := L$:

$$\begin{array}{rcl} C := L & \equiv & C\colon \left[C = L \right] \\ & \preceq & \sigma\colon \left[\mathcal{L}(\sigma) = \mathcal{L}(S) \right] \\ & \sqsubseteq & \sigma\colon \left[\sigma = S \right] \\ & \sqsubseteq & \sigma := S \end{array}$$

- Replace the loop guard, $C \neq \varnothing$, by $\mathcal{L}(\sigma) \neq \varnothing$, which simplifies to $\sigma \neq \phi$.
- Data refine the statement $C := d(\mathsf{hd}\, t, C)$:

$$\begin{array}{rl} C := d(\mathsf{hd}\, t, C) \equiv & C\colon \left[C = d(\mathsf{hd}\, t, C_0) \right] \\ \preceq & \sigma\colon \left[\mathcal{L}(\sigma) = d(\mathsf{hd}\, t, \mathcal{L}(\sigma_0)) \right] \end{array} \qquad (i)$$

  We will consider how to refine this below.

- Data refine the assignment $r := (t = \lambda \wedge \lambda \in C)$:

$$\begin{array}{rcl} r := (t = \lambda \wedge \lambda \in C) & \preceq & r := (t = \lambda \wedge \lambda \in \mathcal{L}(\sigma)) \\ & \sqsubseteq & r := (t = \lambda \wedge \sigma = \lambda) \end{array}$$

  The last simplification is possible because we have assumed that there are no λ-rules in G, so $\lambda \in \mathcal{L}(\sigma) \equiv \sigma = \lambda$.

4.3 Computing the derivative

We will now consider how to refine (i) so as to remove $\mathcal{L}$. We will do this by first introducing a new function d', such that $\mathcal{L}(d'(x, \sigma)) = d(x, \mathcal{L}(\sigma))$. We can thus rewrite (i) as:

$$\sigma\colon \left[\sigma = d'(\mathsf{hd}\, t, \sigma_0) \right] \qquad (i')$$

Our problem now is how to compute $d'(x, \sigma)$. We begin by observing that $d'(x, \sigma)$ is easy to compute when $\mathsf{hd}\, \sigma$ is not a nonterminal, since, for any string σ, and any terminals x and y:

$$\begin{array}{l} d'(x, x \frown \sigma') = \sigma', \\ d'(x, y \frown \sigma') = \phi \text{ if } x \neq y, \text{ and} \\ d'(\phi, \sigma) = \phi \end{array}$$

Thus, our first aim is to get σ into a form where $\mathsf{hd}\, \sigma$ is not a nonterminal. In doing this, we can modify σ in any way so long as $\mathcal{L}(d'(\mathsf{hd}\, t, \sigma))$ remains unchanged. We refine (i') to achieve this as follows:

$$\begin{array}{lll} (i') & \sqsubseteq & (\textit{Split Specification}) \\ & & \sigma\colon \left[d'(\mathsf{hd}\, t, \sigma) \cong d'(\mathsf{hd}\, t, \sigma_0) \ \wedge\ \mathsf{hd}\, \sigma \notin V_N \right]; \qquad (ii) \\ & & \sigma\colon \left[\mathsf{hd}\, \sigma \notin V_N \ \big/\ \sigma = d'(\mathsf{hd}\, t, \sigma_0) \right] \qquad (iii) \end{array}$$

The relation $\cong$ is defined such that, for all strings σ and σ' over V, $\sigma \cong \sigma' \equiv \mathcal{L}(\sigma) = \mathcal{L}(\sigma')$.

Using the easy cases for d' given above, we can now refine (iii) as follows:

$$
\begin{array}{lll}
(iii) & \sqsubseteq & (\textit{Introduce IF, Introduce Assignment}\ \text{(twice)}) \\
& & \textbf{if}\ \ \sigma \neq \phi \wedge \mathsf{hd}\,t = \mathsf{hd}\,\sigma \rightarrow \sigma := \mathsf{tl}\,\sigma \\
& & [\!]\ \ \sigma = \phi \vee \mathsf{hd}\,t \neq \mathsf{hd}\,\sigma \rightarrow \sigma := \phi \\
& & \textbf{fi}
\end{array}
$$

We now consider how to transform σ so that $\mathsf{hd}\,\sigma$ is not a nonterminal. The easiest way to change σ without changing $\mathcal{L}(d'(\mathsf{hd}\,t, \sigma))$ is to replace $\mathsf{hd}\,\sigma$ by the right-hand side of some rule defining $\mathsf{hd}\,\sigma$. Thus, we want to find some string, say α, such that $\mathsf{hd}\,\sigma ::= \alpha \in P$ and $d'(\mathsf{hd}\,t, \sigma) \cong d'(\mathsf{hd}\,t, \alpha \frown \mathsf{tl}\,\sigma)$; we can then replace σ by $\alpha \frown \mathsf{tl}\,\sigma$.

Since the resulting value of σ may still start with a nonterminal, we might need to repeat the process. Thus, we want a loop, with $\mathsf{hd}\,\sigma \in V_N$ as the guard, and a loop invariant which ensures that $d'(\mathsf{hd}\,t, \sigma)$ remains unchanged, i.e.

$$
\mathit{Inv}\ \widehat{=}\ d'(\mathsf{hd}\,t, \sigma) \cong d'(\mathsf{hd}\,t, \sigma_I)
$$

where σ_I is the value of σ at the beginning of the loop, and will be declared as a logical constant surrounding the loop. We will also need a variant function, which we will call f, providing a well-founded ordering on σ; this will be discussed later.

$$
\begin{array}{lll}
(ii) & \sqsubseteq & (\textit{Introduce CON, Introduce DO}) \\
& & \textbf{con}\ \sigma_I \bullet \\
& & \textbf{do}\ \ \mathsf{hd}\,\sigma \in V_N \rightarrow \\
& & \quad \sigma\!: \left[\mathit{Inv} \wedge \mathsf{hd}\,\sigma \in V_N \ \middle/\ \mathit{Inv} \wedge 0 \leq f(\sigma) < f(\sigma_0) \right] \qquad (iv) \\
& & \textbf{od}
\end{array}
$$

The loop body needs to find a suitable α, and update σ as indicated above. Thus, we introduce a local variable, α, and split the specification into two parts. Since we know how to update σ, we use *Following Assignment* to find the condition the must be satisfied by the chosen value of α, as follows:

$$
\begin{array}{l}
\mathit{wp}(\sigma := \alpha \frown \mathsf{tl}\,\sigma,\ d'(\mathsf{hd}\,t, \sigma) \cong d'(\mathsf{hd}\,t, \sigma_I) \wedge 0 \leq f(\sigma) < f(\sigma_0)) \\
\quad = d'(\mathsf{hd}\,t, \alpha \frown \mathsf{tl}\,\sigma) \cong d'(\mathsf{hd}\,t, \sigma_I) \wedge 0 \leq f(\alpha \frown \mathsf{tl}\,\sigma) < f(\sigma))
\end{array}
$$

Thus, we refine (iv) as follows:

$$
\begin{array}{lll}
(iv) & \sqsubseteq & (\textit{Introduce Local Variable, Following Assignment, Contract Frame}) \\
& & \textbf{var}\ \alpha \bullet \\
& & \alpha\!: \left[\begin{array}{c} d'(\mathsf{hd}\,t, \sigma) \cong d'(\mathsf{hd}\,t, \sigma_I) \wedge \\ \mathsf{hd}\,\sigma \in V_N \end{array} \middle/ \begin{array}{c} d'(\mathsf{hd}\,t, \alpha \frown \mathsf{tl}\,\sigma) \cong d'(\mathsf{hd}\,t, \sigma_I) \wedge \\ 0 \leq f(\alpha \frown \mathsf{tl}\,\sigma) < f(\sigma) \end{array} \right]; \qquad (v) \\
& & \sigma := \alpha \frown \mathsf{tl}\,\sigma
\end{array}
$$

Since (v) cannot change σ and $\cong$ is an equivalence relation, we can simplify (v):

$$
\begin{array}{lll}
(v) & \sqsubseteq & (\textit{Weaken Precondition, Strengthen Postcondition}) \\
& & \alpha\!: \left[\mathsf{hd}\,\sigma \in V_N \ \middle/ \begin{array}{c} d'(\mathsf{hd}\,t, \alpha \frown \mathsf{tl}\,\sigma) \cong d'(\mathsf{hd}\,t, \sigma) \wedge \\ 0 \leq f(\alpha \frown \mathsf{tl}\,\sigma) < f(\sigma) \end{array} \right] \qquad (vi)
\end{array}
$$

4.4 Selecting rules

We now consider how to refine (vi) so as to find a suitable value for α, and how to show that the loop terminates. We begin by investigating the first condition in the postcondition (replacing $\mathsf{hd}\, t$ by x):

$$\begin{array}{ll} d'(x, \alpha \frown \mathsf{tl}\,\sigma) \cong d'(x, \sigma) & \\ \quad\iff & \mathcal{L}(d'(x, \alpha \frown \mathsf{tl}\,\sigma)) = \mathcal{L}(d'(x, \sigma)) \\ \quad\iff & d(x, \mathcal{L}(\alpha \frown \mathsf{tl}\,\sigma)) = d(x, \mathcal{L}(\sigma)) \\ \quad\iff & (\forall \beta \bullet \beta \in d(x, \mathcal{L}(\alpha \frown \mathsf{tl}\,\sigma)) \equiv \beta \in d(x, \mathcal{L}(\sigma))) \\ \quad\iff & (\forall \beta \bullet x \frown \beta \in \mathcal{L}(\alpha \frown \mathsf{tl}\,\sigma) \equiv x \frown \beta \in \mathcal{L}(\sigma)) \\ \quad\iff & (\forall \beta \bullet \alpha \frown \mathsf{tl}\,\sigma \Rightarrow^*_G x \frown \beta \equiv \sigma \Rightarrow^*_G x \frown \beta) \end{array}$$

Thus, in order to satisfy $d'(\mathsf{hd}\, t, \alpha \frown \mathsf{tl}\,\sigma) \cong d'(\mathsf{hd}\, t, \sigma)$, we require that any string in $\mathcal{L}(\sigma)$ starting with $\mathsf{hd}\, t$ is also in $\mathcal{L}(\alpha \frown \mathsf{tl}\,\sigma)$, i.e. any string starting with $\mathsf{hd}\, t$ that can be produced from σ can also be produced from $\alpha \frown \mathsf{tl}\,\sigma$. This can be formalised by introducing a function *first* such that, for any string $\gamma \in V^*$,

$$\mathit{first}(\gamma) \;\widehat{=}\; \{\, x \in V_T \mid \exists \gamma' \in V_T^* \bullet \gamma \Rightarrow^*_G x \frown \gamma' \,\}$$

Thus, we require $\mathsf{hd}\, t ::= \alpha \in P \wedge \mathsf{hd}\, t \in \mathit{first}(\alpha)$, if such an α exists. If $d(x, \mathcal{L}(\sigma)) = \varnothing$, we want $\alpha = \phi$.

The above condition also requires that this choice of α be unique, since $\sigma \Rightarrow^*_G x \frown \beta \Rightarrow \alpha \frown \mathsf{tl}\,\sigma \Rightarrow^*_G x \frown \beta$ says that *any* string beginning with x that can be produced from σ can be produced from $\alpha \frown \mathsf{tl}\,\sigma$. If there is more than one such α, the postcondition will be *false*, so (vi) is miraculous!

We will therefore add a further assumption, that for any nonterminal N and terminal x, there is at most one rule $N ::= \alpha$ in P, such that $x \in \mathit{first}(\alpha)$. This is equivalent to the assumption that G is an LL(1) grammar[4].

We thus refine (vi) as follows:

$$(v) \;\sqsubseteq\; (\mathit{Strengthen\ Postcondition}) \qquad\qquad (vi)$$
$$\alpha\!: \left[\mathsf{hd}\,\sigma \in V_N \;\Big/\; \begin{array}{c} (\; \mathsf{hd}\, t ::= \alpha \in P \wedge \mathsf{hd}\, t \in \mathit{first}(\alpha) \;) \;\vee \\ (\; \alpha = \phi \wedge \neg(\exists \beta \bullet N ::= \beta \in P \wedge \mathsf{hd}\, t \in \mathit{first}(\alpha)) \;) \end{array} \right]$$

4.5 Termination

As the variant function, f, we take the length of the longest sequence $\sigma_1, \cdots, \sigma_n$, such that:

1. $\sigma_1 = \sigma$,
2. for $i = 1, \cdots, n-1$, $\mathsf{hd}\,\sigma_i \in V_N$ and for some α, $\mathsf{hd}\,\sigma_i ::= \alpha \in P$ and $\sigma_{i+1} = \alpha \frown \mathsf{tl}\,\sigma_i$, and
3. $\mathsf{hd}\,\sigma_n = \mathsf{hd}\, t$.

[4]The other condition usually applied to LL(1) grammars isn't needed, since we have assumed there are no λ-rules in G.

This function will be bounded below, provided that the grammar has no left-recursive rules (e.g. rules of the form $N ::= N\beta$), which follows from the above assumption that G is an LL(1) grammar; this also prohibits circular rules.

Alternatively, we can define an ordering, $>$, on symbols in $V_N \cup V_T$, such that $x > y$ iff $x ::= y\beta \in P$, for some string β. This is a well-founded ordering provided G has no left-recursive rules.

4.6 Representing rules

In order to implement (vi), we need to be able to compute *first* for any given nonterminal ($\mathsf{hd}\,\sigma$) and input symbol ($\mathsf{hd}\,t$). We may, however, require the value of *first* many times for the same pair of symbols. This suggests that we should compute all possible values for *first* and store them in a table.

We can introduce such a table, which we will call RR, as a further data refinement. The coupling invariant in this case is:

$$
\begin{aligned}
CI \;\widehat{=}\; (\forall N \in V_N, x \in V_T \;\bullet\; \\
&(T(N,x) = \phi \equiv \neg(\exists \alpha \in V^* \;\bullet\; N ::= \alpha \in P \wedge x \in \mathit{first}(\alpha))\,) \wedge \\
&(\forall \alpha \in V^* \;\bullet\; T(N,x) = \alpha \equiv N ::= \alpha \in P \wedge x \in \mathit{first}(\alpha)))
\end{aligned}
$$

A table satisfying this relationship is guaranteed to exist when G is an LL(1) grammar; in particular, the LL(1) property ensures that $RR(N,x)$ is uniquely defined for all N and x.

We will assume that the grammar is provided in this form, since we may wish to construct the table for a given grammar once and then use it many times for recognising different strings. In this case, we can take the above invariant as a specification for a program to construct RR.

Thus, we have:

$$(vi) \quad \preceq \quad \alpha := RR(\mathsf{hd}\,t, \mathsf{hd}\,\sigma)$$

4.7 The final program

The resulting algorithm after performing this data refinement is shown in Figure 1.

The algorithm we have arrived at is a table-driven LL(1) parser or "predictive" parser, in which σ acts as a stack. The operation $\sigma := \alpha \frown \mathsf{tl}\,\sigma$ pushes the symbols in α onto the top of the stack, after popping it; while $\sigma := \mathsf{tl}\,\sigma$ pops the stack. It remains to choose suitable implementations for *First* and RR, most likely as tables, and for the stack; but these details need not concern us here.

It is interesting to compare this algorithm with the predictive parsing algorithm usually given in texts such as [Aho, Sethi and Ullman, 1986]. That algorithm uses a single loop, with cases inside the loop according to whether the top of stack is a terminal or nonterminal. The version derived here makes the nature of the parsing process clearer: for each input symbol, we have to reduce nonterminals at the top of the stack until the top of stack is the same as the input symbol. The termination proof is also easier with this version, since the outer loop advances once for each input symbol (which is a consequence of the design of our abstract algorithm) and the inner loop reduces nonterminals — it is more difficult to find a suitable variant function (or ordering) when these both happen in a single loop. It should, however, be straightforward to transform the algorithm into a single loop version if required.

$$
\begin{array}{l}
[\![\ \textbf{var}\ t, \sigma\ \bullet \\
\quad t, \sigma := s, S; \\
\quad \textbf{do}\ t \neq \lambda\ \wedge\ \sigma \neq \phi \rightarrow \\
\quad\quad \textbf{do}\ \sigma \neq \phi\ \wedge\ \mathsf{hd}\,\sigma \in V_N \rightarrow \\
\quad\quad\quad \alpha := RR(\mathsf{hd}\,t, \mathsf{hd}\,\sigma); \\
\quad\quad\quad \sigma := \alpha \frown \mathsf{tl}\,\sigma \\
\quad\quad \textbf{od}; \\
\quad\quad \textbf{if}\ \ \sigma \neq \phi \wedge \mathsf{hd}\,t = \mathsf{hd}\,\sigma \rightarrow \sigma := \mathsf{tl}\,\sigma \\
\quad\quad [\!]\ \ \ \sigma = \phi \vee \mathsf{hd}\,t \neq \mathsf{hd}\,\sigma \rightarrow \sigma := \phi \\
\quad\quad \textbf{fi}; \\
\quad\quad t := \mathsf{tl}\,t \\
\quad \textbf{od}; \\
\quad r := (t = \lambda \wedge \sigma = \lambda) \\
]\!]
\end{array}
$$

Figure 1: The predictive parsing algorithm

The use of ϕ to denote the empty language simplifies the handling of errors. The fact that we have ended up with tests for $\sigma = \phi$ in both the inner loop and the following conditional, suggests that we might have been better to check for this earlier. Indeed, we could have separated this case when we refined (i'), which would have also simplified the postcondition of (vi) — allowing the second conjunct to be dropped. This did not seem natural at the time, since we would have needed the *first* function, which we had not yet introduced.

5 Conclusions

We have illustrated the use of data refinement and program specialisation in deriving a reasonably complex algorithm. We believe that the derivation is more intelligible that would have otherwise been obtained using program refinement. In particular, we arrived at a parser for LL(1) grammars by adding restrictions to the initial problem in order to simplify implementation problems, rather than starting with that class of grammars in the initial specification. We believe that this is typical of the way in which new algorithms are discovered.

The places where we introduced these assumptions indicate the places where we would have to modify the derivation to handle a larger class of languages. To handle grammars with λ-rules, we would need a more elaborate way of determining whether $\lambda \in \mathcal{L}(\sigma)$, and the code to select a rule would need to be able to determine when to choose a λ-rule. If we wish to handle all context free grammars, we would need to either introduce backtracking (which is one way of avoiding the need for miracles), or represent C by a set of strings over V, or equivalently, using strings of the form $\sigma_1 \mid \sigma_2$, with the interpretation that $\mathcal{L}(\sigma_1 \mid \sigma_2) = \mathcal{L}(\sigma_1) \cup \mathcal{L}(\sigma_2)$. The resulting representations can, however, become very large.

Our decision to represent $d(s - t, L)$ by a string of terminal and/or nonterminal symbols lead us directly to a top-down parsing algorithm; at every step, $(s - t) \frown \sigma$ is a left-sentential form. If, instead, we represented $s - t$ by a string σ of terminal and/or nonterminal symbols, in such a way σ is a prefix of a right-sentential form, we will end up with a bottom-up parsing algorithm (cf [Hesselink, 1992]).

Acknowledgments

Thanks to Ray Nickson for his helpful comments on earlier versions of the paper.

References

[Aho, Sethi and Ullman, 1986] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Brzozowski, 1964] J. A. Brzozowski. "Derivatives of Regular Expressions". *J.A.C.M.* **11** (1964), pp481–494.

[Gravell, 1991] A. Gravell. "Specialising Abstract Programs". *Proc. 4th Refinement Workshop*, J. M. Morris & R. C. Shaw (Eds), Springer-Verlag, 1991, pp34–50.

[Groves, 1994a] Lindsay Groves. "Deriving language recognition algorithms: A case study in combining program specialisation and data refinement". *Proc. 6th Refinement Workshop*, British Computer Society, London, January 1994.

[Groves, 1994b] Lindsay Groves. "Program Derivation in the Refinement Calculus: An Introduction" *New Zealand Formal Program Development Colloquium*, Hamilton, 30 November–1 December, 1994.

[Hesselink, 1992] Wim H. Hesselink. "LR-parsing derived". Science of Computer Programming **19** (1992), pp171–196.

[Hopcroft and Ullman, 1979] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[Morgan, 1990] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[Morgan and Gardiner, 1988] Carroll Morgan and Paul Gardiner. "Data refinement by calculation". *Acta Informatica* **27** (1990), pp481–503.

[Morris, 1989] J. M. Morris. "Laws of data refinement". *Acta Informatica* **26** (1989), pp287–308.