Delayed semantic actions in Yakker

Trevor Jim
AT&T Labs–Research
Florham Park, NJ, USA
trevor@research.att.com

Yitzhak Mandelbaum
AT&T Labs–Research
Florham Park, NJ, USA
yitzhak@research.att.com

ABSTRACT

Yakker is a parser generator that supports semantic actions, lexical binding of semantic values, and speculative parsing techniques such as backtracking and context-free lookahead. To avoid executing semantic actions in speculative parses that will eventually be discarded, we divide parsing into two conceptually independent phases. In the first (early) phase, the parser explores multiple possible parse trees without executing semantic actions. The second (late) phase executes the delayed semantic actions once the first phase has determined they are necessary. Execution of the two phases can be overlapped.

We structure the early phase as a transducer which maps the input language to an output language of labels. A string in the output language is a *history* of the semantic actions that would have been executed in a parse of the input. The late phase is implemented as a deterministic, recursive descent parse of the history.

We formalize delayed semantic actions and discuss a number of practical issues involved in implementing them in Yakker, including our support for regular right part grammars and dependent parsing, the design of the data structures that support histories, and memory management techniques critical for efficient implementation.

Categories and Subject Descriptors

F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems—Grammar Types, Parsing; D.3.1 [**Programming languages**]: Formal Definitions and Theory—Syntax; D.3.4 [**Programming languages**]: Processors—Parsing

1. INTRODUCTION

Yakker is a parser generator that supports *dependent parsing*, in which semantic actions can be executed in the middle of a parse, and can influence the rest of the parse [1, 2]. For example, a parser may need to calculate the value of a length field before parsing the remainder of a record.

Some semantic actions, however, are not used to influence parsing itself. Executing these actions during parsing can be undesireable—they could introduce pointless overhead and extraneous side effects in parsers that support ambiguity or use backtracking or other speculative parsing techniques. Therefore, Yakker also supports semantic actions whose execution can be *delayed* until after it has determined a final parse tree.

Yakker parses in two conceptually independent phases. The first, early phase does all the work needed to determine a final parse tree, including executing any semantic actions necessary for dependent parsing, and resolving any ambiguities as required by the the input language. We structure the early phase as a transducer whose output is a language of labels that record a *history* that gives the sequence of delayed semantic actions that should be executed for the chosen parse. This sequence is *replayed* by the second, late phase to execute the delayed semantic actions. The late phase is structured as a very simple parser for the output language of the early phase. In practice, the execution of the early and late phases can overlap.

Our two-phase strategy has several advantages.

First, the implementation details of the early phase are irrelevant to the late phase, so long as the early phase transduces each input to the correct output. This is useful because our early phase is complicated: it supports arbitrary context-free languages, dependent parsing, and several parsing back ends (including Earley, GLR and backtracking) [1]. We have also reimplemented it several times.

Second, the output language of the early phase can be chosen to achieve technical goals. For example, we have chosen to make our output language a deterministic context-free language, so that our late phase can be a simple, fast, recursive-descent parser; additionally, this lets us easily support lexically-bound semantic values in regular right part grammars. We will see that other choices of output language can reduce the time and space overhead of our early phase.

Contributions.

This paper is a practical guide to implementing our two-phase technique. We use a fragment of Yakker that is sufficient to illustrate our treatment of delayed semantic actions, while avoiding most of the complexities of dependent parsing. Dependent parsing is treated in more detail in our previous work [1, 2], and our complete implementation is available online.

We show how to construct an early-phase transducer by a grammar transformation from a user-level language, Gul$_L$, to an intermediate-level language, Gil. Gul$_L$ includes delayed semantic actions, lexical binding for semantic values, and regular right sides. Gil is a simpler language that can be easily implemented on top of a variety of parsing algorithms, as we have shown previously [1]. Semantic actions in Gil are not delayed, and we use them to build a partial parse forest that encodes all of the histories for an input. An output (a single history) is read from the forest by a simple postfix traversal.

The construction of the late-phase replay parser depends almost entirely on the choice of the early-phase output language. We construct the output language through a notion of *relevance* that essentially lets us record only those parts of a full parse forest that are necessary for later replay. For scannerless grammars, relevance has the added advantage that it implicitly distinguishes lexical nonterminals, which have few or no associated semantic actions, from the rest of the grammar. Finally, we have chosen to use a deterministic context-free output language, so that our late-phase parser can be a simple recursive-descent parser.

2. GRAMMAR LANGUAGES

We now define three different languages that we use in our implementation: *Gul$_L$*; *labeled Gul$_L$*; and *Gil*.

Gul$_\mathbf{L}$ is our user-level language for defining context-free grammars with delayed semantic actions.[1] An example Gul$_L$ grammar can be found at the beginning of Section 3; formally, its syntax is defined as follows:

$$\begin{array}{rcl} G & = & (A_1(x_1) = R_1), \ldots \\ R & = & \epsilon \mid c \mid (R \mid R) \mid R\ R \mid (*R) \\ & \mid & \{e\} \mid A(e) \mid (x{=}R\ R) \mid *^{x=e}R \mid \textsc{pos} \end{array}$$

A *grammar* is a sequence of definitions for *nonterminals* in terms of *right sides.* We use G, A, and R to range over grammars, nonterminals, and right sides, respectively. Right sides are based on regular expressions, including the empty string ϵ, *terminals* c, alternation, and Kleene closure (written as a prefix '$*$'). *(Delayed) semantic actions* are written $\{e\}$, where e is an expression taken from some expression language. We assume that the expression language is a subset of some general-purpose programming language, which we call the *target* language. Nonterminals are defined with formal parameters ranging over semantic values, and are applied to target-language expressions. The right side $(x{=}R_1\ R_2)$ is the concatenation of R_1 and R_2, where x is bound to the semantic value of R_1 in the scope R_2. We write $(R_1\ R_2)$ for a concatenation which does not require binding. We include a folding version $(*^{x=e}R_1)$ of Kleene closure, which starts with an initial expression e, and then (implicitly) folds over the input to produce a final value. The POS construct evaluates to the current input position without consuming any input. It is most useful for scannerless grammars. Tokenized grammars, which are also compatible with Gul$_L$, can usually get positions from tokens.

Our techniques are not specific to any particular target language, but for the sake of concreteness we will assume some variant of the untyped, call-by-value lambda calculus. Note that the target language may have its own binding forms, and there is no requirement that target expressions must use only Gul$_L$-bound variables—expressions may reference library functions or globals.

In our examples, we will write concrete terminals in quotes, for example, **'b'** for the ASCII character lowercase b. We assume that the target language has a distinguished unit value, written (), as well as booleans. If the parameter of a nonterminal A is not used in its right side, we omit it. We may omit the parentheses in $(*R)$, $(R_1\ R_2)$, $(R_1 \mid R_2)$, and $(x{=}R_1\ R_2)$ when this does not cause confusion.

Labeled Gul$_\mathbf{L}$ is an extension of Gul$_L$ in which right sides can be annotated with *labels* ℓ:

$$R = \ldots \mid {}^{\ell}R \mid R^{\ell}$$

Concrete labels are written $\underline{1}$, $\underline{2}$, ... A labeled Gul$_L$ grammar defines a transducer from terminals to labels and input positions: each label indicates an output (matching the empty string), and POS outputs the current input position. For example, the labeled Gul$_L$ right side

$$*(\textbf{'a'}^{\underline{1}})\ *(\textbf{'b'})\ \textsc{pos}$$

transduces the input **aaabbbb** into the output $\underline{1}\ \underline{1}\ \underline{1}\ 7$.

Gil is an intermediate-level language of grammars that we use to implement early-phase transducers. Its semantic actions are executed in the middle of (during) parsing, in contrast to Gul$_L$'s delayed semantic actions; we will record histories using Gil's semantic actions. Gil's syntax is defined by the following rules:

$$\begin{array}{rcl} g & = & (A_1 = r_1), \ldots \\ r & = & \epsilon \mid c \mid (r \mid r) \mid (*r) \mid (r\ r) \mid \{f\} \mid A(f_{\text{arg}}, f_{\text{ret}}) \end{array}$$

Gil, like Gul$_L$, is based on regular expressions over terminals and nonterminals. To distinguish between Gul$_L$ and Gil we use g, r, and f to range over Gil grammars, right sides, and target language expressions, instead of Gul$_L$'s G, R, and e.

Gil lacks the binding forms of Gul$_L$: nonterminals are defined without a formal parameter, and there is no binding concatenation. Instead, Gil right sides are *value transformers*: they map an implicit semantic value input to an output. The full semantics of Gil are given in our earlier paper [1]. For convenience, we present a few of the key rules here:[2]

$$\frac{f(i)(v) = v_1}{\langle v, i\rangle \xrightarrow{\{f\}} \langle v_1, i\rangle} \qquad \frac{\langle v, i\rangle \xrightarrow{r_1} \langle v_1, i_1\rangle \quad \langle v_1, i_1\rangle \xrightarrow{r_2} \langle v_2, i_2\rangle}{\langle v, i\rangle \xrightarrow{(r_1 r_2)} \langle v_2, i_2\rangle}$$

$$\frac{\begin{array}{l} f_{\text{arg}}(i)(v) = v_1, (A = r) \in g \\ \langle v_1, i\rangle \xrightarrow{r} \langle v_2, i_2\rangle \\ f_{\text{ret}}(v)(v_2) = v_3 \end{array}}{\langle v, i\rangle \xrightarrow{A(f_{\text{arg}}, f_{\text{ret}})} \langle v_3, i_2\rangle}$$

Semantic actions are the base value transformers, and our rule for concatenation shows that Gil threads values from left to right across parses. An instance of a nonterminal in a right side has two arguments: the first is used to calculate the initial semantic value for the nonterminal's right side; and the second is used to combine the output of the nonterminal's right side with the value from before the nonterminal's invocation.

[1]We use the subscript L (for 'late') to distinguish Gul$_L$ from the language Gul of [1].

[2]We have slightly modified the rules for semantic actions and nonterminals to provide the current position as an argument to f and f_{arg}, respectively.

3. DELAYED SEMANTIC ACTIONS

We will explain our technique using the following example. It is a Gul$_L$ grammar that calculates sums, e.g., on input **12+345**, a parser for the grammar calculates 357:

$$\begin{array}{rcl}
\textsc{start} & = & \textsc{sum}(0) \\
\textsc{sum}(x) & = & y{=}\textsc{number}\ \{x+y\} \\
 & \mid & y{=}\textsc{number}\ \text{‘+’}\ \textsc{sum}(x+y) \\
\textsc{number} & = & p_1{=}\textsc{pos}\ \textsc{digit}\ {*}\textsc{digit}\ p_2{=}\textsc{pos}\ \{\text{atoi}(\text{sub}(p_1,p_2))\} \\
\textsc{digit} & = & \textbf{‘0’} \mid \textbf{‘1’} \mid \textbf{‘2’} \mid \textbf{‘3’} \mid \textbf{‘4’} \mid \textbf{‘5’} \mid \textbf{‘6’} \mid \textbf{‘7’} \mid \textbf{‘8’} \mid \textbf{‘9’}
\end{array}$$

The nonterminal SUM(x) does most of the work. It recursively parses a sum, calculates its value, and returns the value plus x. The START nonterminal initializes the parse as SUM(0). The rule for NUMBER parses a sequence of digits, remembering the input position before and after the sequence, and then uses library functions sub (to extract the sequence from the full input) and atoi (to convert the sequence to an integer semantic value).

If the parse of a SUM occurs in the context of a larger, speculative parse, we would clearly like to delay the action $\{\text{atoi}(\text{sub}(p_1,p_2))\}$ in the right side of NUMBER until after we have determined that it is necessary, because it is fairly expensive. However, the positions bound to p_1 and p_2 obviously become available during parsing. We therefore would like to calculate p_1 and p_2 during parsing, store them in the history, and delay all of the other semantic actions until after parsing.

Figures 1–4 show how we achieve this. We first annotate the grammar with labels (written $\underline{1}$, $\underline{2}$, ...) before each semantically-significant construct (POS, bindings, actions, parameters, and, recursively, nonterminals with labels). A labeled version of the original grammar is shown in Figure 1. Recall that in labeled Gul$_L$, the labels are outputs (matching ϵ), and POS outputs the input position (in addition to returning the position as a semantic value, as in unlabeled Gul$_L$). We call the sequence of labels and positions output during a single parse of an input a *history*, and, as we will see, a history records all of the information necessary to execute the semantic actions in the late stage.

On input **12+345**, a transducer for the grammar in Figure 1 calculates 357 and outputs the history shown in Figure 2. The essence of our technique is to separate this combined computation into an early phase that only outputs the history, and an independent late phase that consumes the history and calculates the result, 357.

The grammar in Figure 3 implements the early phase. It is a labeled Gul$_L$ grammar derived from the grammar in Figure 1 by dropping all semantic computations: bindings and actions are replaced by ϵ, and parameters are omitted. Since we have started with a grammar whose semantic actions do not influence *what* is parsed, this results in a grammar with exactly the same parses, and hence outputs, as that of Figure 1. Unlike that grammar, the early phase grammar does not calculate the sum.

The grammar in Figure 4 implements the late phase. It is an unlabeled Gul$_L$ grammar that consumes histories produced by the grammar in Figure 3 and calculates the desired sum. It is derived from the grammar in Figure 1 by omitting all of the original terminals, and by transforming labels into new terminals (notice that this lets us completely omit the nonterminal DIGIT, whose original definition was simply a set of terminals with no semantic actions). In addition, POS is replaced by a special nonterminal, POS$_V$, which both matches and returns semantic values of positions (nonnegative integers).

$$\begin{array}{rcl}
\textsc{start} & = & {}^{\underline{1}}\textsc{sum}(0) \\
\textsc{sum}(x) & = & {}^{\underline{2}}y{=}\ {}^{\underline{3}}\textsc{number}\ \ {}^{\underline{4}}\{x+y\} \\
 & \mid & {}^{\underline{5}}y{=}\ {}^{\underline{6}}\textsc{number}\ \ \text{‘+’}\ {}^{\underline{7}}\textsc{sum}(x+y) \\
\textsc{number} & = & {}^{\underline{8}}p_1{=}\ {}^{\underline{9}}\textsc{pos}\ \textsc{digit}\ ({*}\textsc{digit})\ {}^{\underline{10}}p_2{=}\ {}^{\underline{11}}\textsc{pos} \\
 & & {}^{\underline{12}}\{\text{atoi}(\text{sub}(p_1,p_2))\} \\
\textsc{digit} & = & \textbf{‘0’} \mid \textbf{‘1’} \mid \textbf{‘2’} \mid \textbf{‘3’} \mid \textbf{‘4’} \mid \textbf{‘5’} \mid \textbf{‘6’} \mid \textbf{‘7’} \mid \textbf{‘8’} \mid \textbf{‘9’}
\end{array}$$

Figure 1: The original grammar after labeling.

$\underline{1}\ \underline{5}\ \underline{6}\ \underline{8}\ \underline{9}\ 0\ \underline{10}\ \underline{11}\ 2\ \underline{12}\ \underline{7}\ \underline{2}\ \underline{3}\ \underline{8}\ \underline{9}\ 3\ \underline{10}\ \underline{11}\ 6\ \underline{12}\ \underline{4}$

Figure 2: The output sequence of labels and positions for input 12+345.

$$\begin{array}{rcl}
\textsc{start} & = & {}^{\underline{1}}\textsc{sum} \\
\textsc{sum} & = & {}^{\underline{2}}\epsilon\ {}^{\underline{3}}\textsc{number}\ {}^{\underline{4}}\epsilon \\
 & \mid & {}^{\underline{5}}\epsilon\ {}^{\underline{6}}\textsc{number}\ \text{‘+’}\ {}^{\underline{7}}\textsc{sum} \\
\textsc{number} & = & {}^{\underline{8}}\epsilon\ {}^{\underline{9}}\textsc{pos}\ \textsc{digit}\ ({*}\textsc{digit})\ {}^{\underline{10}}\epsilon\ {}^{\underline{11}}\textsc{pos}\ {}^{\underline{12}}\epsilon \\
\textsc{digit} & = & \textbf{‘0’} \mid \textbf{‘1’} \mid \textbf{‘2’} \mid \textbf{‘3’} \mid \textbf{‘4’} \mid \textbf{‘5’} \mid \textbf{‘6’} \mid \textbf{‘7’} \mid \textbf{‘8’} \mid \textbf{‘9’}
\end{array}$$

Figure 3: The early-phase grammar.

$$\begin{array}{rcl}
\textsc{start} & = & \underline{1}\ \textsc{sum}(0) \\
\textsc{sum}(x) & = & \underline{2}\ y{=}\ (\underline{3}\ \textsc{number})\ \ \underline{4}\ \{x+y\} \\
 & \mid & \underline{5}\ y{=}\ (\underline{6}\ \textsc{number})\ \underline{7}\ \textsc{sum}(x+y) \\
\textsc{number} & = & \underline{8}\ p_1{=}\ (\underline{9}\ \textsc{pos}_V)\ \underline{10}\ p_2{=}\ (\underline{11}\ \textsc{pos}_V)\ \underline{12} \\
 & & \qquad \{\text{atoi}(\text{sub}(p_1,p_2))\}
\end{array}$$

Figure 4: The late-phase grammar.

Our construction has the following useful properties:

1. The original grammar is equivalent to the composition of the early- and late-phase grammars.

2. The early-phase grammar has no semantic actions.

3. The late-phase grammar is deterministic.

(1) is our basic correctness property and it also implies that we can take advantage of standard techniques for grammar composition, e.g., we can overlap execution of the early- and late-phase grammars. (2) shows that we have successfully separated parsing from semantic actions, and (3) implies that it will be easy to implement the semantic actions, even though they involve lexical bindings and regular right sides.

Implementing the early-phase grammar. There is a wide literature on both regular and context-free transducers. Our needs are a bit unusual, however: we need context-sensitive transducers, to support POS and dependent parsing as described in [1, 2]. Therefore we will go into some detail.

Transducers in general can be ambiguous, mapping a single input to many outputs. To avoid exponential space complexity, our transducers encode outputs using a data structure much like Tomita's shared packed parse forests [11].

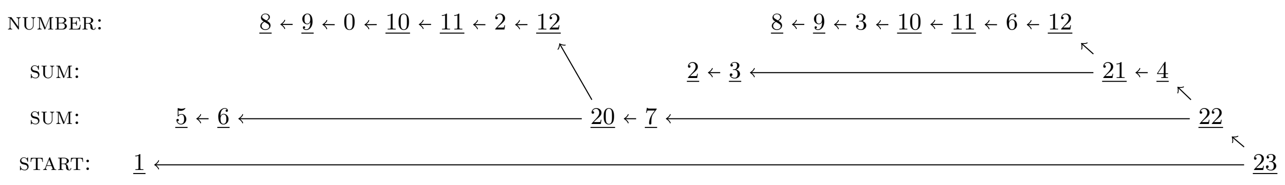


Figure 5: Forest constructed for a parse of 12+345, with root, $\underline{23}$, at lower right. A postfix traversal that omits the merge labels $\underline{20}$–$\underline{23}$ gives the history of Figure 2.

The forest for our example input is given in Figure 5. The input has only a single parse, so this particular forest is just a tree, written bottom-up: label sequences in rows indicate a parse of a nonterminal, and upwards edges indicate where another nonterminal was parsed. Binary nodes are used to "merge" the parse of one nonterminal into another; we choose the labels $\underline{20}$–$\underline{23}$ on these merge nodes to be distinct from the labels used in Figure 1. For example, the sequence $\underline{2} \leftarrow \underline{3} \leftarrow \underline{21} \leftarrow \underline{4}$ in the second row corresponds to a parse of SUM, and the arrow from $\underline{21}$ to the top row indicates that a parse of NUMBER occurred between labels $\underline{3}$ and $\underline{4}$. A simple postfix traversal of the tree in Figure 5, omitting merge labels, produces the history in Figure 2.

To build forests, we transform the early-phase grammar of Figure 3 into the following Gil grammar:

$$\begin{array}{rcl}
\text{START} & = & \{\mathrm{P}(\underline{1})\}\ \text{SUM}(\mathrm{E}, \mathrm{M}(\underline{26})) \\
\text{SUM} & = & \{\mathrm{P}(\underline{2})\}\ \{\mathrm{P}(\underline{3})\}\ \text{NUMBER}(\mathrm{E}, \mathrm{M}(\underline{23}))\ \{\mathrm{P}(\underline{4})\} \\
& | & \{\mathrm{P}(\underline{5})\}\ \{\mathrm{P}(\underline{6})\}\ \text{NUMBER}(\mathrm{E}, \mathrm{M}(\underline{24})) \\
& & \quad \text{`+'}\ \{\mathrm{P}(\underline{10})\}\ \text{SUM}(\mathrm{E}, \mathrm{M}(\underline{25})) \\
\text{NUMBER} & = & \{\mathrm{P}(\underline{8})\}\ \{\mathrm{P}(\underline{9}); \mathrm{P}(\mathrm{pos}())\}\ \text{DIGIT}\ (\text{*DIGIT})\ \{\mathrm{P}(\underline{10})\} \\
& & \{\mathrm{P}(\underline{11}); \mathrm{P}(\mathrm{pos}())\}\ \{\mathrm{P}(\underline{12})\} \\
\text{DIGIT} & = & \textbf{`0'} \mid \textbf{`1'} \mid \textbf{`2'} \mid \textbf{`3'} \mid \textbf{`4'} \mid \textbf{`5'} \mid \textbf{`6'} \mid \textbf{`7'} \mid \textbf{`8'} \mid \textbf{`9'}
\end{array}$$

It builds the forest shown in Figure 5 on our example input. It includes new actions which build forest nodes during the early phase, using three forest constructors. The *empty* constructor E builds the empty forest; the *push* constructor P(ℓ) builds a new forest node with label ℓ and with the old forest as its single child; and the *merge* constructor M(ℓ) is used after a nonterminal to build a new forest node labeled ℓ with two children—the forest before the parse of the nonterminal, and the forest describing the parses of the nonterminal itself. The function pos() returns the current input position.

Implementing the late-phase grammar. We have already noted that the late-phase "replay" grammar is a deterministic context-free grammar: by construction, all labels in the grammar are distinct, and, every choice point in the grammar is preceded by a distinct label. Therefore, it can be implemented with a straightforward recursive-descent parser that takes as input the label/position sequence, and executes the late stage semantic actions. For our example, we use three recursive parsing functions:

```
replay$_{\text{START}}$() = skip($\underline{1}$); replay$_{\text{SUM}}$(0)

replay$_{\text{SUM}}$($x$) =
  match next() with
    $\underline{2}$. let $y$ = (skip($\underline{3}$); replay$_{\text{NUMBER}}$());
       skip($\underline{4}$); $x + y$
  | $\underline{5}$. let $y$ = (skip($\underline{6}$); replay$_{\text{NUMBER}}$());
       skip($\underline{7}$); replay$_{\text{SUM}}$($x + y$)

replay$_{\text{NUMBER}}$() =
  skip($\underline{8}$);
  let $p_1$ = (skip($\underline{9}$); next());
  skip($\underline{10}$);
  let $p_2$ = (skip($\underline{11}$); next());
  skip($\underline{12}$); atoi(sub($p_1$,$p_2$))
```

The functions use a stateful interface to the label/position sequence: there is an implicit pointer to the current position of the sequence, and the function next() returns the element at that position and advances the pointer. A stateful implementation is appropriate for replay, because, in this second stage, the final parse has been decided.

Here skip(ℓ) is short for $\boxed{\text{(match next() with } \ell\text{. ())}}$. So, skip($\ell$) asserts that the next element of the input is ℓ and advances the input. In fact, we know that the assertion cannot fail, because we only run the replay parser on output sequences of the early-phase transducer. The presence of skip() in our replay functions shows that the outputs of the early phase are overly-detailed—they encode almost a complete parse tree. This can easily be improved by noting that all of the labels that are skipped in our replay functions are in fact unnecessary for replay; we do not have to put them into forests in the first place, and we do not need to skip() them in the replay functions. For our example, this means that only labels $\underline{2}$ and $\underline{5}$ need to be replayed, and only labels $\underline{2}$, $\underline{5}$, and the merge labels $\underline{20}$, $\underline{21}$, $\underline{22}$, and $\underline{23}$ need to be stored in the forest (the merge labels are never replayed).

Our implementation builds transducers and replay functions using this observation, and it reduces forest space considerably, but we will omit the optimization from our formal treatment to simplify its presentation. Similarly, we note that there are still other choices for output languages (e.g., we could omit still more labels but remain in LL(1)) but leave that to future work.

4. IMPLEMENTING FORESTS

The forest datatype represents a set of trees. Here is its OCaml definition:

```
type 'a forest = {v:'a; mutable pack:'a tree list}
and 'a tree =
  | Zero
  | One of 'a forest
  | Two of 'a forest * 'a forest
```

A forest contains one or more trees, all of which have the same value, `v:'a`, at their root nodes. We only need trees whose nodes have zero, one or two children, represented by the constructors of the `tree` type. The `pack` of a forest enumerates the children for each of the trees represented by the forest. For example, the forest

```
{v=1; pack=[Zero; One {v=2;branchings=[Zero]}]}
```

represents two trees, 1 and $1 \rightarrow 2$. The use of pack thus allows us to encode multiple trees with the same node value, and corresponds to Tomita's "local ambiguity packing." Note that pack is a mutable field, so that forests can be constructed incrementally.

The forest datastructure, by itself, does not enforce sharing. We ensure sharing by hashconsing forests, rather than constructing them directly. The hashconsing is hidden beneath the interface of a shared packed forest class:

```
class ['a] spf : ('a forest -> 'a forest) ->
  object ('self)
    method e : 'a -> 'self
    method p : 'a -> 'self
    method m : 'a -> 'self -> 'self
    method get_value : 'a
    method history : 'a history
  end
```

The spf class takes an initializer of type `('a forest -> 'a forest)`, namely, a memoizer for forests. The spf implementation uses the memoizer to make sure that all trees with the same value are represented by the same forest node. We omit the full details of the spf implementation (it is available online), noting only that we are careful to use OCaml's weak hashtable library to implement the memoizer, so that memoization itself does not retain history nodes that become garbage as a result of speculative parsing. The methods `e`, `p` and `m` correspond to the constructors E, P, and M described in Section 3.

The spf class is parameterized over the type of value carried by nodes; it is not specific to parsing. We implement a shared packed *parse* forest by instantiating the type parameter with a value containing the grammar label and left and right input positions that a node represents. The sppf type and the types of constructors E, P and M are:

```
type sppf = (label * pos * pos) spf
val e : pos -> sppf
val p : label -> pos -> sppf -> sppf
val m : label -> pos -> sppf -> sppf -> sppf
```

Their implementation simply manages construction of appropriate values for each node, delegating the rest of the work to the corresponding spf methods. The P and M constructors use the `get_value` method to access the left position of an argument sppf's value, for inclusion in the value of the result.

Every spf object also has a `history` method that returns an object that can access the serialized node values of the spf:

```
class ['a] history :
  object
    method next : unit -> 'a
  end
```

A history object just conducts a lazy postfix traversal of a forest, arbitrarily choosing a branching if it encounters multiple branchings.[3] The history object is the basis of the function next() described in Section 3.

5. THE GRAMMAR TRANSFORMATIONS

In this section, we formally present the transformation from a Gul$_L$ grammar to a Gil grammar that constructs a history forest during parsing and a replay function that traverses a history and executes the semantic actions from the user's grammar. The transformation occurs in a number of phases. The initial phases (normalization, labeling and erasing) closely parallel those used to transform the Gul language [1]. We discuss those phases briefly and refer the reader to our previous work for more detail.

At the core of our transformation is the notion of *relevance*, which defines when nonterminals and right sides are semantically meaningful.

Definition 1
The relevance of the nonterminals and right sides of a grammar are defined as the least relations satisfying the following properties:

- *A right side is* relevant *if it includes a target-language expression or a* relevant *nonterminal.*
- *A nonterminal is* relevant *if its right-side is* relevant.

Notice that bindings do not impact relevance, because what matters for history construction is whether the binding is used.

Irrelevant right sides and rules only require recognition, and not reconstruction. They therefore need not be recorded in the forest and can be ignored by our transformations. However, the relevance of a right side is not necessarily apparent from its top-level syntax. So, for the purposes of our formalism, we restrict inputs to grammars which have been normalized so that their syntax reflects their relevance.

Normalization.

We say that a right side R is *normalized* if every subterm R' of R satisfies the following properties:

- If R' is $(x{=}R_1\ R_2)$ then both R_1 and R_2 are relevant.
- If R' is $(R_1\ R_2)$ then at least R_1 is not relevant.
- If R' is $(R_1 \mid R_2)$ then R_1 and R_2 share the same relevance.
- If R' is $({*}R_1)$ then R_1 is not relevant.
- If R' is $({*}^{x=e}R_1)$ then R_1 is relevant.

We say that a grammar G is *normalized* if every right-side in the grammar has been normalized.

Definition 2 (Gul$_L$-to-Gil Transformation)
We say that a normalized Gul$_L$ grammar G transforms to a Gil grammar g, written $G \Rightarrow g$, iff g is the least grammar such that the following conditions hold:

- *If $(A(x) = R) \in G$ and R is irrelevant, then $(A = R) \in g$.*
- *If $(A(x) = R) \in G$ and R is relevant, then $(A = \mathcal{H}[\![R_\ell]\!]) \in g$, and $\mathit{replay}_{\mathrm{A}}(x) = $ match next() with $\big((\mathcal{R} \circ \mathcal{E})[\![R_\ell]\!]\big)$, where $R_\ell = \mathcal{L}[\![R]\!]$.*

We describe each of the transformations mentioned above in turn.

[3]Disambiguation of forests can be applied before this point to avoid such arbitrary choices. Yakker supports a variety of disambiguation methods, which we will not discuss here.

$\boxed{\mathcal{L}[\![R]\!] = R'}$ (R is normalized)

If R is not relevant, then $\mathcal{L}[\![R]\!] = R$.

Otherwise, $\mathcal{L}[\![R]\!]$ is defined by the following cases. In each case, ℓ and ℓ' denote fresh labels.

$$\mathcal{L}[\![\textsc{pos}]\!] = {}^{\ell}\textsc{pos} \qquad \mathcal{L}[\![\{e\}]\!] = {}^{\ell}\{e\}$$

$$\mathcal{L}[\![A(e)]\!] = {}^{\ell}A(e)^{\ell'}$$

$$\mathcal{L}[\![(R_1 \mid R_2)]\!] = (\mathcal{L}[\![R_1]\!] \mid \mathcal{L}[\![R_2]\!])$$

$$\mathcal{L}[\![(R_1\ R_2)]\!] = (R_1\ \mathcal{L}[\![R_2]\!])$$

$$\mathcal{L}[\![(x{=}R_1\ R_2)]\!] = {}^{\ell}(x{=}\mathcal{L}[\![R_1]\!]\ \mathcal{L}[\![R_2]\!])$$

$$\mathcal{L}[\![(*^{x=e}R)]\!] = {}^{\ell}(*^{x=e}\mathcal{L}[\![R]\!])^{\ell'}$$

Figure 6: Labeling right sides

$\boxed{\mathcal{E}[\![R]\!] = R'}$ (R is relevant and normalized)

$$\mathcal{E}[\![{}^{\ell}\textsc{pos}]\!] = {}^{\ell}\textsc{pos} \qquad \mathcal{E}[\![{}^{\ell}\{e\}]\!] = {}^{\ell}\{e\}$$

$$\mathcal{E}[\![{}^{\ell}A(e)^{\ell'}]\!] = {}^{\ell}A(e)^{\ell'}$$

$$\mathcal{E}[\![(R_1 \mid R_2)]\!] = (\mathcal{E}[\![R_1]\!] \mid \mathcal{E}[\![R_2]\!])$$

$$\mathcal{E}[\![(R_1\ R_2)]\!] = \mathcal{E}[\![R_2]\!]$$

$$\mathcal{E}[\![{}^{\ell}(x{=}R_1\ R_2)]\!] = {}^{\ell}(x{=}\mathcal{E}[\![R_1]\!]\ \mathcal{E}[\![R_2]\!])$$

$$\mathcal{E}[\![{}^{\ell}(*^{x=e}R)^{\ell'}]\!] = {}^{\ell}(*^{x=e}\mathcal{E}[\![R]\!])^{\ell'}$$

Figure 7: Erasing irrelevant subterms

Labeling $\mathcal{L}[\![\cdot]\!]$.

Our first step is to add *labels* to Gul right sides. Each label identifies a control-flow point in the right side. These labels serve to synchronize the construction of replay functions with the insertion of forest constructors. The insertion of labels considerably simplifies the specification of those two phases, which otherwise could not be specified independently.

We only need to add labels to relevant subterms of a right side. The labeling transformation is given in Figure 6. We use underlined integers for labels, and use ℓ to range over integers used as labels.

Erasing $\mathcal{E}[\![\cdot]\!]$.

The replay function for a Gul right side is constructed exclusively from relevant subterms of the right side. We can simplify the definition of replay-function construction if we first erase all subterms that are not relevant. A suitable transformation is given in Figure 7. It has the important property that the resulting right side exactly preserves the control-flow of the labels of the original right side.

History $\mathcal{H}[\![\cdot]\!]$.

The transformation $\mathcal{H}[\![\cdot]\!]$, shown in Figure 8, uses the three

$\boxed{\mathcal{H}[\![R]\!] = r}$ (R is normalized)

If R is not relevant, then $\mathcal{H}[\![R]\!] = R$. Otherwise, $\mathcal{L}[\![R]\!]$ is defined by the following cases.

$$\mathcal{H}[\![{}^{\ell}\textsc{pos}]\!] = \{\textsc{p}(\ell); \textsc{p}(\text{pos}())\} \qquad \mathcal{H}[\![{}^{\ell}\{e\}]\!] = \{\textsc{p}(\ell)\}$$

$$\mathcal{H}[\![{}^{\ell}A(e)^{\ell'}]\!] = \{\textsc{p}(\ell)\}\ A(\textsc{e}, \textsc{m}(\ell'))$$

$$\mathcal{H}[\![{}^{\ell}(x{=}R_1\ R_2)]\!] = \{\textsc{p}(\ell)\}\ \mathcal{H}[\![R_1]\!]\ \mathcal{H}[\![R_2]\!]$$

$$\mathcal{H}[\![{}^{\ell}(*^{x=e}R)^{\ell'}]\!] = \{\textsc{p}(\ell)\}\ *(\mathcal{H}[\![R]\!])\ \{\textsc{p}(\ell')\}$$

$$\mathcal{H}[\![(R_1 \mid R_2)]\!] = (\mathcal{H}[\![R_1]\!] \mid \mathcal{H}[\![R_2]\!])$$

$$\mathcal{H}[\![(*R)]\!] = *\mathcal{H}[\![R]\!]$$

$$\mathcal{H}[\![(R_1\ R_2)]\!] = \mathcal{H}[\![R_1]\!]\ \mathcal{H}[\![R_2]\!]$$

$$\mathcal{H}[\![\epsilon]\!] = \epsilon \qquad \mathcal{H}[\![c]\!] = c$$

Figure 8: History-forest construction

$\boxed{\mathcal{R}[\![R]\!] = C}$

$$\mathcal{R}[\![{}^{\ell}\textsc{pos}]\!] = \ell.\text{next}() \qquad \mathcal{R}[\![{}^{\ell}\{e\}]\!] = \ell.e$$

$$\mathcal{R}[\![{}^{\ell}A(e)^{\ell'}]\!] = \ell.\text{replay}_A(e)$$

$$\begin{aligned}\mathcal{R}[\![{}^{\ell}x{=}R_1\ R_2]\!] = \ell.&\text{let } x = \text{match next() with } \mathcal{R}[\![R_1]\!];\\ &\text{match next() with } \mathcal{R}[\![R_2]\!]\end{aligned}$$

$$\begin{aligned}\mathcal{R}[\![{}^{\ell}(*^{x=e}R)^{\ell'}]\!] = \ell.&\text{let rec } g\ x = \text{match next() with}\\ &\qquad \ell'.x \mid y.g(\text{match } y \text{ with } (\mathcal{R}[\![R]\!]));\\ &g(e)\end{aligned}$$

where g and y are fresh variables

$$\mathcal{R}[\![(R_1 \mid R_2)]\!] = \big(\mathcal{R}[\![R_1]\!] \mid \mathcal{R}[\![R_2]\!]\big)$$

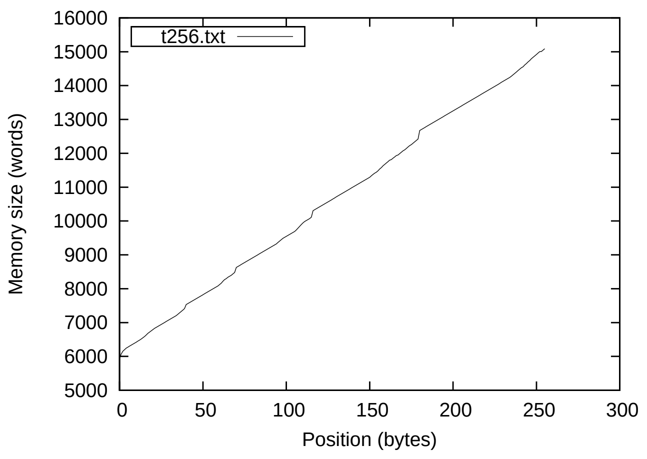
Figure 9: Replay function

forest constructors described in Sections 3 and 4. The nonterminal and POS cases are notable. $\mathcal{H}[\![{}^{\ell}A(e)^{\ell'}]\!]$ shows that forest construction does not depend upon arguments. The POS case uses the function pos(), which returns the current input position.

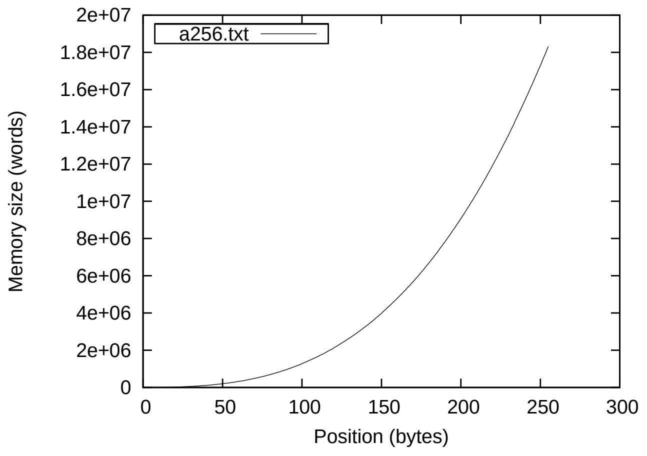
Notice that $\mathcal{H}[\![{}^{\ell}A(e)^{\ell'}]\!]$ adds the label ℓ' to the history, but $\mathcal{R}[\![{}^{\ell}A(e)^{\ell'}]\!]$ does not skip the ℓ'. This is because next() skips over merge labels when doing its postfix traversal of the forest.

Replay $\mathcal{R}[\![\cdot]\!]$.

Finally, we construct replay functions from grammars (after erasing), as shown in Figure 9. The rules follow the example discussed in Section 3. The only new case is that of folding Kleene closure. The replay function for this case essentially performs a left fold over a subsequence of the history (ending immediately before ℓ' rather than at the end of the history), using the replay code for the body R as the combining function of the fold.



(a) S = *A A = **‘s’** {`print "A"`}



(b) S = **‘s’** {`print "S"`} S S | ϵ

Figure 10: Memory consumption for two simple grammars describing a series of ‘s’ characters. Note the significant difference in the y-scale of the the two plots.

6. EVALUATION

We briefly evaluate the memory consumption of Yakker’s parsers when using histories. Since the main contribution of histories relates to design, and not performance or memory size, our goal is simply to confirm that our implementation’s memory profile conforms to standard asymptotic bounds.

We consider three grammars. The first two accept an arbitrary number of **‘s’** characters and print some output. They are shown in Figure 10, along with plots of their memory consumption compared to input position for a series of 256 **‘s’** characters. The first grammar is deterministic and the linear growth of its memory profile is, therefore, in line with our expectations. The second grammar is highly ambiguous and results in exponentially many parses relative to the input length. The cubic (rather than exponential) nature of the graph demonstrates the effect of the sharing used in our forests.

The third grammar is a grammar for OCaml, adapted from the official lexer and grammar to be scannerless and use regular right sides. In Figure 11(a), we plot, for each source file of Yakker, the size of the parse forest representing that file. The size, shown on the y-axis, is measured in number of distinct nodes, while the x-axis is the size of the respective source file. Note that both axes use a logarithmic scale. The data suggests a linear growth in memory consumption, which is consistent with the (almost) LR(1) nature of the OCaml grammar. The three outliers from the otherwise linear relation all belong to generated files. Those above the line are denser in code (vs. whitespace) than the handwritten files, and the one below the line is sparser.

In Figure 11(b), we examine a particular file, “engine.ml,” which is the source code for Yakker’s parsing engine. We plot the position in the file against the parser’s total memory footprint. Note the flat section of the graph between approximately positions 2500 and 8000. That part of the source is a comment, which contains no semantic actions and, therefore, results in no history growth.

7. RELATED WORK

Our work touches on a number of different elements of parsing theory and implementation, including semantic actions, AST inference and parse forests.

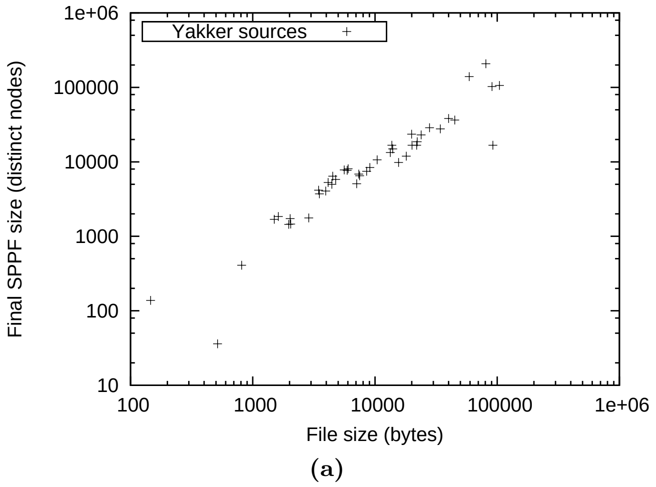
Regarding semantic actions, there is a broad spectrum of techniques for supporting user-specified semantics for a grammar. Some systems, including the many variants of Yacc, execute semantic actions during parsing [3]. While more typical of deterministic parsers, some general parsers do the same, for example, Elkhound executes actions during parsing, and provides the user with low-level control over semantic-value management to address the complications that arise from ambiguities [5].

On the other end of the spectrum are systems like SDF, which automatically constructs a default parse forest and leaves forest processing to other tools [12, 13]. Attribute grammars, too, typically delay attribute evaluation until after parsing, at which point attributes are evaluated based on the entire parse tree [6].

In between these approaches are systems that provide a special-purpose language of annotations for constructing parse trees [7], or for transforming parse trees to more useable forms [9, 4]. Our work is most closely related to these “in-between” approaches, such as the *restructured derivation trees* (RDTs) of Johnstone and Scott.

RDTs are like abstract syntax trees derived by transformation from the derivation (parse) tree, rather than by fiat. They leverage the insight that abstract syntax trees are generally a small delta from parse tree, and that, therefore, the effort of building up ASTs in semantic actions from scratch is largely redundant. They extend context-free grammars with a language of annotations for transforming parse trees to RDTs, which can then be used like ASTs in downstream processing. Our forests are parallel to their RDTs, keeping only information related to execution of semantic actions. However, whereas they provide an explicit language for transforming parse trees to RDTs, we infer the transformation based on placement of semantic actions. They do specify an algorithm for automatically inferring a *canonical RDT* for any BNF grammar[9], but the transformations employed are based on syntactic considerations, whereas we utilize semantic considerations.

Our implementation methods also parallel those of Johnstone and Scott. They compile annotated grammars (called TIF grammars) into a lower-level language of syntax-directed



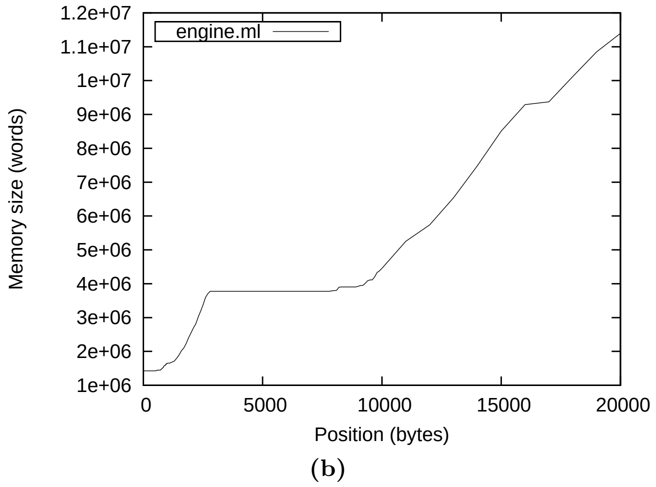


Figure 11: History evaluation for the OCaml grammar. Plot (a) compares, for all of Yakker's source files, the numbers nodes of the SPPF representing that source against the source's size. Plot (b) shows, for a single OCaml source, the memory footprint of our parser for each position in the source.

definitions with explicit parse tree construction. While the intermediate language in their more recent paper is semantically closer to Gul, they also describe compilation to a Gil-like language of *translation schemes* in their earlier paper. Finally, their "TIF transformed grammars" (TTGs)—the grammar of the RDTs derivable from a TIF grammar—correspond to our grammars for histories.

Note that our technical distinction from systems which omit support for semantic actions does not imply an *ideological* distinction on the question of whether embedded semantic actions harm the declarativity of a grammar. Gul could play the role of a target language for a deforesting-style optimization applied to, for example, an SDF grammar and corresponding tree transform.

Regarding parse forests, the terminology of *shared packed parse forest* originally comes comes from Tomita [11], although similar data structures were used in earlier work. The particular style of SPPF which we implement as the `forest` data structure is most closely related to the binarized SPPFs of Scott et al. [10, 8]. Their focus on sharing prefixes and suffixes of subparses, rather than sharing and packing parses of entire nonterminals, was most appropriate to our support for regular right sides (in addition to being asymptotically more efficient).

8. REFERENCES

[1] T. Jim and Y. Mandelbaum. A new method for dependent parsing. In *Twentieth European Symposium on Programming*, 2011.

[2] T. Jim, Y. Mandelbaum, and D. Walker. Semantics and algorithms for data-dependent grammars. In *Proceedings of the 37th Annual ACM Symposium on Principles of Programming Languages*, New York, NY, USA, 2010. ACM.

[3] S. C. Johnson. Yacc: Yet another compiler compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[4] A. Johnstone and E. Scott. Tear-Insert-Fold grammars. In *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, LDTA '10, pages 6:1–6:8, New York, NY, USA, 2010. ACM.

[5] S. McPeak and G. C. Necula. Elkhound: A fast, practical GLR parser generator. In *CC 2004: Compiler Construction, 13th International Conference*, 2004.

[6] J. Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Computing Surveys (CSUR)*, 27(2):196–255, 1995.

[7] T. Parr. ANTLR parser generator.

[8] E. Scott. SPPF-style parsing from Earley recognisers. In *Proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007)*, volume 203 of *Electronic Notes in Theoretical Computer Science*, pages 53–67. Elsevier, 2008.

[9] E. Scott and A. Johnstone. Constructing reduced derivation trees. Technical Report CSD–TR–98–09, Computer Science Department, Royal Holloway, University of London, Surrey, UK, October 2003.

[10] E. Scott, A. Johnstone, and G. Economopoulos. BRN-table based GLR parsers. Technical Report TR–03–06, Computer Science Department, Royal Holloway, University of London, London, UK, 2003.

[11] M. Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems.* Kluwer Academic Publishers, Norwell, MA, USA, 1985.

[12] M. van den Brand, , P. Klint, and J. Vinju. *The Syntax Definition Formalism SDF*, May 2007.

[13] E. Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, September 1997.