1 Chart parsing

This section summarizes the chart parsing component in the *agree* grammar engineering environment. This component includes a number of important optimizations from the DELPH-IN literature. As in van Louhizen's concurrent adaptation of the PET parser, my system supports multi-processor operation. Extending that author's work, I reduce the "communication phase" to a sequence of lock-free operations. Management of the parsing agenda is currently delegated to the tasking facility which was recently added to the runtime environment. The sophisticated component reportedly incorporates techniques from the latest research in concurrent processing, such as adaptive hill-climbing, work-stealing, CPU cache sensitivity, and more. Beyond throttling the number of tasks queued to this component, the *agree* parser does no task scheduling or prioritization.

1.1 Interlocked parse chart

The *agree* chart parser uses an interlocking global sequence counter to coordinate lock-free transactions between passive and active edges. This section describes the data structures and state transitions for lock-free operations which are complete under chart parsing, modulo ambiguity packing. Fully lock-free operations include inserting or deleting passive edges into chart cells and inserting or deleting active edges into notification lists. A locking mechanism appears to be required to support ambiguity packing, but by maintaining edge-level lock granularity and restricting lock acquisition to narrowly-identified conditions, contention penalties have not been severe in extensive 8-way testing. In the next section, I discuss the organization of the chart data structures.

1.1.1 Chart data structures

The chart parser is a conventional bottom-up design. For sentence $S = (w_0, w_1, \dots, w_n)$, the chart consists of $\sum_1^n n$ independent cells, arranged as a zero-indexed jagged array where $cell[i][j]$ holds all passive edges which cover the inclusive zero-based token span $[i, n-j]$. Each "cell" is actually no more than a tuple of object references for managing a singly-linked list, `m_first` and `m_last`. Active edges are stored in a different set of lists, discussed next. For the remainder of this discussion, 'list' refers to either type of interlocked linked list: a chart cell or, a list of active edges subscribed to spans abutting a token position.

As noted, active edges are not stored in the chart. Instead, each active edge is placed in exactly one notification list. There are two sets of notification lists, each containing $n-1$ independent lists. One set of notification sources is for leftwards notifications, and the other set is for rightwards notifications. The lists in the former allow the enrolled active edges to be notified of new passive edges whose right edges abut (the leftwards edge of) token positions $(1, 2, \dots, n-1)$. Similarly, the lists in the latter pertain to passive edges whose left edges abut (the rightwards edge of) token positions $(0, 1, \dots, n-2)$. Naturally, these lists exclude the possibility of zero-spanning passive edges. Upon creation, an active edge is permanently designated as leftwards- or rightwards-growing. This designation is based on the parser's key strategy; various options are configurable by the grammar author or through configuration parameters. It then subscribes for either a leftwards- or rightwards-notifications—but not both—for the position of its active daughter position. The

completed portion of the active edge establishes the active edge's chart alignment in the non-growth direction, but spans of any length are welcome candidates for the active position. It is for this reason—and because the interlocking mechanism requires that each active edge be enrolled in only a single list—that each subscription list pertains to all edges—of any span—which abut a given position.

To eliminate extraneous allocations, the `m_next` fields which continue the lists are reserved fields contained within the respective listed objects themselves. With strong-typing, this means that chart cells can only contain passive edges, and notification lists can only contain active edges. The linked lists used for chart cells and active edges operate in a standard lock-free way.

Items are always added to the tail of the list; the `m_last` pointer is an opportunistically maintained pointer to some point near the tail of the list, and this hint greatly improves the performance of the append operation. Obviously, only forward traversal of these lists starting from the head is possible, and this operation is sufficient for all parsing tasks, including packing. This being so, the presence of the `m_last` hint requires further explanation, because it would be simpler—and a bit more efficient for the actual operation itself—to insert new items at the head of the list, rather than the tail. One reason for always adding to the tail is that it better separates reading from writing CPUs. The list head is more heavily accessed than the tail: it must be read by any thread that starts an enumeration of any number of list elements; if the list head is constantly being written, readers will encounter frequent cache misses. However, there is a much more critical justification for the practice of appending to the list tail; the locking conditions introduced by ambiguity packing, discussed below, are greatly relaxed when newly-added objects are not the first to be enumerated.

1.1.2 Parsing sequence

The key issue for using a singleton chart in multi-threaded parsing is to ensure that a given passive edge is never evaluated for an available active edge position more than once. Failure to ensure this results in either extra work—a gross inefficiency, or duplicate derivations—an incorrect result, depending on the failure or success of the redundant unification.

Between a given active edge *A* and an eligible candidate passive edge *P*, the situation that needs arbitration is whether it is the responsibility of *A* to evaluate the match on its own—because *A* was not yet enrolled in a subscription list at the time *P*'s creation was broadcast—or whether *A* should do nothing, because *A* was properly enrolled, and so will eventually be notified of *P*'s creation via the "normal" subscription broadcast.

To resolve these disputes, it is a simple matter to register both passive and active edges in the same global sequence, thus imposing an arbitrary total order over parse objects. Upon insertion into its list, each passive or active edges is issued a permanent, integer sequence value taken from an atomically incremented counter that is global to the parse operation. The precise sequence enregistration process, which is the heart of the overall technique, is discussed below, but for continuity we finish

the discussion by assuming a deterministic ordering over active and passive edges, and note the conditions that are observed to ensure efficient and correct operation:

1. After creating a new *active edge* and inserting it into exactly one subscription list, the parse task is privately responsible for evaluating every and only the passive edges in that list that have a lower sequence identifier than the new active edge.
2. After inserting a new *passive edge* into a chart cell, the parse task is privately responsible for evaluating the edge against all of the active edges amongst the zero, one, or two subscription lists that apply, according to the token positions the new passive edge abuts. From this collection of active edges, every and only the active edges that have a lower sequence identifier than the new passive edge are evaluated.

Note that objects must be inserted in their respective lists before any matching occurs. Every sequence number partitions the set of all parse objects into two classes: objects that came before, and objects that came afterwards. It is critical that the former—that is, all prior objects—be visible in some list at the moment this partition occurs. This observation suggests that the above procedure is not complete without one additional guarantee; this is the discussion of the next section.

1.1.3 Interlocking sequence enregistration

The two conditions detailed in the previous section ensure that new items get “caught-up” with exactly and only those items that existed prior to their creation. Once an object is caught-up to that point in time, its responsibility ends, and it becomes dormant, because further checking will be the responsibility of the (parse tasks associated with) later objects to come, during *their* “catch-up” responsibilities. The remaining detail is to describe the process for registering an object in the interlocking sequence. This is a critical step in the process because, as it stands, the procedure manifests a prominent race condition.

The simplest way to describe the concurrency error in the parsing sequence is to state the additional constraint that eliminates it.

3. At the moment a parse object is inserted into a list, it must have the highest sequence identifier amongst all objects; no object in any other list may have a higher sequence identifier.

This condition prevents matching responsibilities from falling through the cracks, wherein an object D dutifully enumerates all of the objects that it is responsible for matching, and seemingly completes the task of matching those with lower sequence identifiers, yet misses out on one or more objects, M. It is easy to immediately rule out misuse of the sequence counter as a source of this error. Advancing the global sequence counter is achieved with an interlocked processor instruction, so it is never possible to issue a lower identifier value than one that was issued earlier in time. Therefore, the only way that object M can be missed is if D was created after M has been issued a sequence identifier, but finishes all of its matching duties before M becomes visible in its designated list. This will be

possible if there is any window of time between an object getting its identifier and it being inserted in a list. The additional constraint prohibits this race because, when M eventually does get inserted in its list, it will have a lower sequence number than D, which violates condition (3.).

It is clear from the preceding discussion that condition (3.) intricately relates atomic sequencing to atomic list management. Since no object in any list may have a higher identifier than an object the instant it is inserted, the process of issuing an identifier is interdependent with list insertion. This is troublesome because both processes already independently use interlocked (lock-free) techniques. The solution to this problem will also illustrate the importance of the chart's list structures being established so that each object—whether passive or active—joins exactly one list.

To retain lock-freedom, the method shown here uses optimistic speculation and interlocked compare-exchange, which introduces zero-wait spinning in the event of contention. The combined list-insert-sequence-issue atomic operation, during which speculative changes to the global sequence counter are rejected, is compressed into only three sequential memory accesses. The global counter is incremented in two steps. The first is protected by atomic compare-exchange (CMPEXCH) such that it will fail if the value is even. If this operation succeeds, the caller

this point neither active edges nor passive edges have a parse task associated with

for the notification of new passive edges active edges to Like a chart cell, each notification source is simply the head of a singly-linked list.

1.1.4 Concurrent proactive packing

Proactive packing requires minimal locking: an existing passive edge is locked only for the duration of time required to add a subsumed edge to its packing list. Locks on existing passive edges are not required during subsumption testing. However, new edges, which exist

1.1.5 Concurrent retroactive packing

Retroactive packing entails a more substantial locking condition