Lazy Parsing in TXL

TXL Working Paper 8 (Also published as Legasys WP98-145-JRC)
James R. Cordy
May 1998

This paper describes a simple new paradigm for programming grammars in TXL that can be used to minimize backtracking and thus maximize parsing speed in certain highly ambiguous grammars. In particular, it improves PL/I parsing speed by more than 70 percent.

1. TXL Parsing.

In general, TXL's parser uses an "eager" parsing algorithm, in which each nonterminal attempts to eat as much of the input as possible at each invocation. This algorithm is fundamental to understanding TXL and is the basis on which deterministic parsing, and thus pattern matching, on ambiguous grammars is possible. Without this property, TXL would be much more difficult to program and use effectively.

This property is most deeply built in to TXL in the nonterminal modifiers, such as [**opt** X], which necessarily means :

```
define opt_X
        [X]
      | [empty]
end define
```

if the parse is to be eager.

Most often this strategy serves us well, and in particular has allowed us to exploit the power of ambiguous and superset grammars in the implementation of design recovery and analysis, making these processes much simpler and easier to understand.

2. Problems with Eager Parsing.

However, the eager parsing algorithm has a nasty downside in certain situations. For example, in the PL/I unique namer, the nonterminal [label] has been overridden to be [declaration] in order to abstract the concept of declared items and thus make scope analysis simpler and easier to understand.

```
define label
        [declaration] :
end define
```

Where [declaration] is a rich and complex nonterminal capable of partially parsing almost any input for several tokens in several different ways.

Because every PL/I statement may be labeled, the [statement] form has the definition :

```
define statement
        [opt condition_prefix] [opt labels] [unlabeled_statement]
end define
```

Because PL/I has no keywords, any keyword can be mistaken for an identifier, and therefore many statements can be partially parsed as [declaration] (and thus [label]) much or all of the way to the terminating semicolon in several ways before failing and backtracking to an empty [label]. This has been observed to cause anomalously slow parses of PL/I programs in the Unique phase of design recovery, particularly in programs with a relatively large number of statements, because 99.9 % of PL/I statements do not have any [labels], and the eager parser spends all of its time backtracking to discover that fact.

3. The Solution : Lazy Nonterminals.

The solution to this problem, and many others like it, is to use a lazy rather than eager parse to encode the unlikelihood of a match and thus speed parsing. This works only, of course, when the final result parse is not in doubt - as is the case for [labels] since each label ends in a colon which cannot be part of the labeled statement itself.

Fortunately, laziness can be programmed into the grammar itself without change to the TXL parser. The paradigm for programming a lazy nonterminal is simple - just begin with the [empty] alternative. For example, instead of [**opt** labels] in the PL/I grammar, we use [lazy_opt_labels], where :

```
define lazy_opt_labels
        [empty]
      | [labels]
end define
```

This explicitly encodes the fact that we expect that labels will be rare, and pays for a backtrack only in those cases where a label actually is present. In practice, this one simple change speeds up parsing of PL/I by a factor of three or more depending on the input.

Similarly, [**repeat** X] nonterminals can be programmed in lazy fashion using the paradigm :

```
define lazy_repeat_X
        [empty]
    |  [X] [lazy_repeat_X]
end define
```

Which can yield similar improvements in parser performance for rarely present or rarely repeated items.