INTERACTIVE CONSTRUCTION OF SMALL GRAMMARS

JEFFREY CLARK

ABSTRACT. An interactive approach to constructing small LR(1) grammars is presented. An example involving parsing permutations is used to illustrate the approach.

1. INTRODUCTION

Many texts ([1, 2, 3, 4]) present LR(1) grammars as an established fact. When teaching context-free grammars, students often struggle with how to create their own in applications. This paper will provide some suggestions as to how to interactively create a relatively small LR(1) grammar (for a language with a small number of tokens) that accepts all strings, separating out successful completion from a comprehensive set of error conditions. The construction of new productions will be prompted by *proper unaccepted strings*: strings not accepted by a partially completed grammar that can lead to the construction of new productions in an orderly fashion.

At the heart of this approach is the notion that in recognizing and describing errors we can better articulate the productions corresponding to successful parsing.

2. PROPER UNACCEPTED STRINGS

Our goal is to construct a grammar that describes a complete language—every input string (of terminals) is accepted. We will sort errors according to type as well as providing the location of the error.

We will construct the grammar interactively; at each stage of the construction, we will look for a string that is not yet accepted and append productions to the grammar in such a way as to allow the string to be derived deterministically. If we can no longer find such a string we are done. (There is no guarantee beforehand that this process will terminate.)

Such strings can be huge; we will look for strings that are short in the following sense: if w is a string not yet accepted by our grammar such that every proper prefix of w is accepted, then we will call w a *proper unaccepted string*. Note that w may contain non-terminal symbols.

Once we find an unaccepted string, the challenge is to add enough productions to cover the appropriate category containing the string. Too few, and we may never complete; too many and we can make the grammar indeterminate, introducing reduce-shift or shift-shift conflicts in the parser. When in doubt, we will err on the side of too few, as we can always add more productions later. (Otherwise, if we add too many productions, we can end up with reduce-shift or shift-shift conflicts that can be difficult to resolve.)

In order to avoid having to backtrack and revise our grammar, we will wherever possible use general productions, introducing intermediate concepts that can be expanded. A goal is to minimize the semantic distance between the left and right sides of any production.

It is possible using the work of S. Heilbrunner [2] to construct an $LR(1)$ grammar for the complement of a language given by an $LR(1)$ grammar. This work is significant in and of itself, but we only need to use a small portion: we seek to find a short string contained in the complement. We only need to check that at each state in the $LR(1)$ parser generated by our grammar that there is a shift operation defined for every symbol in our language as well as for the end-of-string symbol. When that is not true, we have found a proper unaccepted string.

3. Case Study: Permutations

For a non-trivial example, we will use permutations written in cycle notation.

3.1. **Definition of a permutation.**

Permutations are bijections (one-to-one and onto functions) mapping a set to itself leaving all but a finite number of elements fixed. This is consistent with the common usage of the word "permute".

To be more specific, our permutations will permute a finite set. Here is an example σ in table form:

x	a	b	c	d	e	f
$\sigma(x)$	b	c	a	d	f	e

This particular permutation sends a to b, b to c, and c to a, forming a cycle of length three. By itself d forms a trivial cycle of length 1, and e and f together form a cycle of length two. We can write σ in cycle notation: $(a, b, c)(d)(e, f)$. (Cycles of length one are often omitted from the notation.)

3.2. **General description of permutation strings.**

We will build a parser that will either translate a character string into a permutation (in some data representation) or return an error message detailing where the

parser realized that the string was not a valid permutation. We will articulate a formal description of what constitutes a valid permutation as we proceed.

In practice we would be storing the permutation into some data structure in the course of the parsing; the specifics depend on the programming language used and will be omitted.

3.3. **Terminal and non-terminal symbols.** We can see from our example that parentheses and commas will play a special role in our parser. We will represent them by LEFT, RIGHT, and COMMA tokens. Spaces will be ignored (except as they function to delimit tokens). Every other token POINT will be viewed as the name of a point in the set being permuted. We will define non-terminal symbols as we need them.

4. Constructing the Parser

4.1. **Empty string.** The first proper unaccepted string is the empty string. We need to make a decision at this point. There is a default permutation, the identity function. Should we accept an empty string and interpret it as the identity permutation?

This would be error-prone in practice; it is too easy to submit an empty string by accident. We will require the presence of at least one cycle in our input strings in order to construct a permutation.

We will be very blunt with our error categories: they should describe where the error was found and what should have occurred. In this case, we cannot refer to an index where the error was found since the string has length 0. Rather, this is an error that occurs at the end of the string. We will begin each of our error categories with `ErrorEnd`.

The error is that there are not cycles present in the input string. We needed to open a cycle to get anywhere with our parsing. We will describe this error as occurring because we expected a left parenthesis at the end of the string, abbreviated as `ErrorEndLeft`. As usual, we will use `S` as the start symbol for our grammar.

```
S := ErrorEndLeft
```

We will not try to anticipate all such possible ways that `ErrorEndLeft` could occur at this point; rather we will append this one instance of our error.

```
ErrorEndLeft :=
```

4.2. **Comma.** The second proper unaccepted string is "`COMMA`". One way of describing it as invalid is to say that a comma shouldn't start a permutation. A more useful description is that a permutation shouldn't start with anything but a left parenthesis. (This is more inclusive of other symbols

that are not acceptable at the beginning.) We will prefix errors occurring inside the string with `ErrorIndex`, and label this error as `ErrorIndexLeft`, with the return value being the index of the error in the input string.

```
S := ErrorIndexLeft
ErrorIndexLeft := NotLeft
NotLeft := COMMA | POINT | RIGHT
```

4.3. **Only left parenthesis.** The third proper unaccepted string is "`LEFT`". It is the beginning of a cycle, which would be accepted if it were followed by the rest of the cycle. What should follow a left parenthesis in a cycle?

We are defining `POINT` to be strings and not necessarily numeric, so there is no default non-empty string to use as a `POINT`. When we want to indicate an identity permutation, we will want to be able to do so without having to identify a `POINT` that is fixed, so we will allow empty cycles if only to eventually be able to describe the identity permutation.

The string should therefore continue to a point moved by that cycle, which will begin with a `POINT`, or to a right parenthesis `RIGHT`. We will label this error `ErrorEndPointRight`.

We append the following productions accordingly.

```
S := ErrorEndPointRight
ErrorEndPointRight := LEFT
```

4.4. **Bad cycle start with suffix.** "`POINT COMMA`" is the next unaccepted string. This is the same error that we have seen before `ErrorIndexLeft` but now there is a suffix. We will want to include productions in the grammar to account for any possible non-empty suffix. (This will be used in many other productions as well.) The suffix will be an accumulator `AnyChars`; note that for an LR grammar the recursive production should start with `AnyChars` on the right side.

```
ErrorIndexLeft := NotLeft AnyChars
AnyChars := AnyChar | AnyChars AnyChar
AnyChar := COMMA | LEFT | POINT | RIGHT
```

From now on when we have an error detected at a given index we will also include a production with the `AnyChars` suffix.

4.5. **Bad cycle contents.** The next unaccepted string is "`LEFT COMMA`". This is an error at the comma; it implies that there should have been a point before the comma as well as after, since the comma will serve as a separator.

Thus there should be either a `POINT` where the `COMMA` occurs (as a starting point for the cycle) or a `RIGHT` (for an empty cycle). This error will be labeled `ErrorIndexPointRight`.

```
S := ErrorIndexPointRight
ErrorIndexPointRight := LEFT CommaOrLeft
ErrorIndexPointRight := LEFT CommaOrLeft AnyChars
CommaOrLeft := COMMA | LEFT
```

4.6. **Empty cycle.** The next unaccepted string is "`LEFT RIGHT`". As stated in the previous section, we do want to accept empty cycles. We will consider this a `Cycle`, but we will also leave ourselves the possibility of generalizing to more than one cycle. (The convention in this paper is that plural nonterminals will refer to one or more.) For now we just include a production for a single cycle.

```
S := Cycles
Cycles := Cycle
Cycle := LEFT RIGHT
```

4.7. **Open cycle.** The next unaccepted string is "`LEFT POINT`". This is a cycle that hasn't closed; at the end of the string we would have expected either a comma (to continue with more points in the cycle) or a right parenthesis to close the cycle.

Instead of building this new error, `ErrorEndCommaRight`, around `POINT`, we will again generalize the error to include the possibility of a range of points (separated by commas).

```
S := ErrorEndCommaRight
ErrorEndCommaRight := LEFT Range
Range := POINT
```

4.8. **Waiting for next point.** The next unaccepted string is "`LEFT POINT COMMA`". This error is detected at the end of the string, where we expect a `POINT`, which we will label `ErrorEndPoint`.

```
ErrorEndPoint := LEFT Range COMMA
```

4.9. **Double comma.** The next unaccepted string is "`LEFT RANGE COMMA`". We're still looking for a point, but now at the index of the second comma.

```
S := ErrorIndexPoint
ErrorIndexPoint := LEFT Range COMMA NotPoint
ErrorIndexPoint := LEFT Range COMMA NotPoint AnyChars
NotPoint := COMMA | LEFT | RIGHT
```

4.10. **Unfinished cycle.** The next unaccepted string is "`LEFT RANGE COMMA POINT`". The cycle is not yet closed and could include more points, and is another example of `ErrorEndCommaRight`. This seems redundant, since `Range COMMA POINT` forms another `Range`. Now is the time to include the recursive definition of `Range`.

```
Range := Range COMMA POINT
```

4.11. **Bad end to cycle.** The next unaccepted string is "`LEFT Range COMMA POINT LEFT`". The error here occurs at the last left parenthesis where either a comma or a right parenthesis is expected. We shall call it `ErrorIndexCommaRight`. Note that `Range COMMA POINT` can be reduced to `Range` in our productions.

```
S := ErrorIndexCommaRight
ErrorIndexCommaRight := LEFT Range LeftOrPoint
ErrorIndexCommaRight := LEFT Range LeftOrPoint AnyChars
LeftOrPoint := LEFT | POINT
```

4.12. **Non-empty cycle.** Our next unaccepted string is "`LEFT Range COMMA POINT RIGHT`". We now have a non-empty cycle.

```
Cycle := LEFT Range RIGHT
```

4.13. **Prefixing by cycles.** Our next unaccepted string is "`LEFT Range RIGHT COMMA`". We now start seeing things we've seen before, preceded by one or more cycles. We'll pre-emptively take all of our productions for errors and permutations and add versions that are preceded by `Cycles`.

```
ErrorIndexLeft := Cycles NotLeft
ErrorEndPointRight := Cycles LEFT
ErrorIndexLeft := Cycles NotLeft AnyChars
ErrorIndexPointRight := Cycles LEFT CommaOrLeft
ErrorIndexPointRight := Cycles LEFT CommaOrLeft AnyChars
Cycles := Cycles Cycle
ErrorEndCommaRight := Cycles LEFT Range
ErrorEndPoint := cycles LEFT Range COMMA
ErrorIndexPoint := Cycles LEFT Range COMMA NotPoint
ErrorIndexPoint
     := Cycles LEFT Range COMMA NotPoint AnyChars
ErrorIndexCommaRight := Cycles LEFT Range LeftOrPoint
ErrorIndexCommaRight
      := Cycles LEFT Range LeftOrPoint AnyChars
```

With these productions, the grammar generates the entire language.

5. Summary

By examining proper unaccepted strings for a grammar, constructed interactively, it is possible to extend the grammar to include all strings in the language. This method is only practical for simple grammars with a relatively small number of tokens; in these cases by viewing strings that should be successfully parsed as well as those that should signal an error, we can better articulate the productions in our grammar leading to success.

Error states, categorized by type of error, often are similar enough to be easy to fill out using copy, paste, and revision.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers*, Addison Wesley, Reading, MA, 1986.
[2] S. Heilbrunner, *A direct complement construction for LR*(1) *grammars*, Acta Informatica, **33** (1996), 781–797.
[3] P. Linz, *An Introduction to Formal Languages and Automata*, Jones and Bartlett, Sudbury, MA, 2006.
[4] J. C. Martin, *Introduction to Languages and the Theory of Computation*, McGraw Hill, New York, NY, 2011.

MSC2010: 68N20

Key words and phrases: LR parser, grammar

Dept. of Math & Stat, Elon University, Elon, NC
E-mail address: clark@elon.edu