Solving LR Conflicts Through Context Aware Scanning

C. Rodriguez Leon and L. Garcia Forte

Departamento de EIO y Computación, Universidad de La Laguna, Tenerife, Spain

Abstract. This paper presents a new algorithm to compute the exact list of tokens expected by any LR syntax analyzer at any point of the scanning process. The lexer can, at any time, compute the exact list of valid tokens to return only tokens in this set. In the case than more than one matching token is in the valid set, the lexer can resort to a nested LR parser to disambiguate. Allowing nested LR parsing requires some slight modifications when building the LR parsing tables. We also show how LR parsers can parse conflictive and inherently ambiguous languages using a combination of nested parsing and context aware scanning. These expanded lexical analyzers can be generated from high level specifications.

Keywords: Parsing, Lexical Analysis, Syntactic Analysis, LR
PACS: 68N15, 68N20

INTRODUCTION

Usually the type of a token - the language that the token describes - can be defined by a regular expression or some other autonomous mechanism. There are cases, however, where a token's type depends upon context information that the scanner can not manage. The most quoted case being the PL/I language, where statements like this are legal:

```
if then=if then if if=then then then=if
```

The scanner should anticipate which uses of `if` and `then` conform to identifiers and which to keywords. The problem arises because keywords like `if` and `then` are not reserved, and can be used in other contexts.

The main contribution of this paper is a new algorithm to compute the exact list of tokens expected by the syntax analyzer at any point of the scanning process. The lexer can, at any time, compute the exact list of valid tokens and return only tokens in this set. In the case than more than one token can be returned, the lexer can resort to a nested LR parser to decide which one to return.

Aycock and Horspool [1] proposed the concept of Schröedinger token to solve the problem of context aware scanners. A token does not have a unique type but instead has a superposition of types. Their idea requires a GLR parser to work. Another solution for GLR was introduced by Visser [2]: The scanner/parser dichotomy is eliminated by the use of character-level grammars. XGLR [3] also extends GLR to allow a different scanner state/input position to be associated with each parse thread and thus to use context to support lexically ambiguous input. Despite being robust, GLR algorithms cannot guarantee determinism in conflictive grammars: ambiguities must be solved by the user. Aho, Sethi, and Ullman [4, p.84-85], Aycock and Horspool [1] and Wyk and Schwerdfeger [5] among others, mention several reasons why it is best to kept apart parser and scanner. Packrat parsers [6, 7] are another kind of scannerless parsers. Packrat parsers use parsing expression grammars or PEGs, which look similar to context-free grammars but have a lower level interpretation, which is closer to how string recognition is done by a recursive descent parser. Some implementations [7] give support to non-declarative specifications such as the well known C typename/identifier ambiguity. Packrat parsers, however, are character-level backtracking LL(1) parsers and consequently are unable to parse left-recursive grammars without special modifications. Wyk and Schwerdfeger [5] introduced a new context-aware scanning algorithm in which the scanner uses contextual information to disambiguate lexical syntax. The LR parser algorithm is modified: it passes to the scanner the set of valid symbols that the scanner may return at that point in parsing. An analysis is given that can statically verify that the scanner will never return more than one token for a single input. Our approach does not requires the modification of the LR algorithm and can be combined with nested parsing to select the right token in the case than more than one token matches the current input.

This paper is divided in four sections. The new algorithm is presented in the next section. Having an algorithm to predict the exact set of expected tokens has multiple important applications [5]. The following section presents an important application of the algorithm: difficult shift-reduce and reduce-reduce LR conflicts can be solved by doing the lexer to return a *fictitious* token when some nested parsing succeeds. We illustrate this technique using two examples. The final section summarizes our contributions.

Numerical Analysis and Applied Mathematics ICNAAM 2011
AIP Conf. Proc. 1389, 829-832 (2011); doi: 10.1063/1.3636861

COMPUTING THE EXPECTED TOKENS

This section describes the proposed algorithm to compute the exact set of expected tokens at any point of the parsing process. The algorithm uses simulation on the stack - also known as symbolic interpretation - to achieve its goal. The Algorithm *YYExpected* on the left side of the figure is called with a reference to the LR stack[1]:

YYExpected($\mathscr{S}$)

Require: *Stack* $\mathscr{S}$
Ensure: The set of all expected tokens $\mathscr{E}$
1: $s = top(\mathscr{S})$
2: $\mathscr{E} = \{a \in \Sigma : \exists t \in Q \text{ such that } ACTION(s,a) = shift\ t\ \}$
3: $\mathscr{R} = \{A \to \alpha \in P : ACTION(s,b) = reduce\ A \to \alpha\}$
4: **if** $\mathscr{R} \neq \emptyset$ **then**
5: $\quad \mathscr{E} = \mathscr{E} \cup YYSimStack(\mathscr{S},\mathscr{R})$
6: **end if**
7: **return** $\mathscr{E}$

YYSimStack($\mathscr{S}$,$\mathscr{R}$)

Require: *Stack* $\mathscr{S}$, *Productions* $\mathscr{R}$
1: **for all** $A \to \alpha \in \mathscr{R}$ **do**
2: $\quad$ **if** $length(\mathscr{S}) > length(\alpha)$ **then**
3: $\quad\quad$ *Stack* $\mathfrak{S} = \mathscr{S}$
4: $\quad\quad$ *pop* $\mathfrak{S}$, $length(\alpha)$
5: $\quad\quad$ $s = top(\mathfrak{S})$
6: $\quad\quad$ $n = GOTO(s,A)$
7: $\quad\quad$ *push* $\mathfrak{S}, n$
8: $\quad\quad$ $\mathscr{E} = \mathscr{E} \cup YYExpected(\mathfrak{S})$
9: $\quad$ **end if**
10: **end for**
11: **return** $\mathscr{E}$

The Algorithm *YYExpected* starts traversing the action table for the state s in the top of the stack. Tokens $a \in \Sigma$ for which there is a shift action are directly pushed in the set of expected tokens $\mathscr{E}$ (line 2). The next step is to compute the set $\mathscr{R}$ of productions $A \to \alpha$ in P for which there is a reduction. The remaining expected tokens are computed calling *YYSimStack*, whose code appears on the right side of the Figure.

The Algorithm *YYSimStack* consists of a simulation recursive process that branches at each available reduction. The exploration is kept until no more productions can be applied for reduction. The simulation mimics the LR parsing algorithm: For each production $A \to \alpha \in \mathscr{R}$ the step of reducing by it is performed using a local stack $\mathfrak{S}$: As many states as the number of symbols in the right hand side α are extracted from the stack (line 4), leaving some state s at the top. The *GOTO* table of the LR parser is then consulted to find which is the next state n. The algorithm recursively calls back (line 8) to *YYExpected* to compute the set of tokens expected in state n. The new tokens are added to $\mathscr{E}$.

SOLVING DIFFICULT LR CONFLICTS USING CONTEXT AWARE SCANNING

Having an algorithm to predict the exact set of expected tokens has multiple important applications [5]. This section presents one: a technique to solve difficult shift-reduce and reduce-reduce LR conflicts. The technique combines the use of the former algorithm and nested parsing. To solve the conflict, a *fictitious* token is introduced at the apropriate point of the grammar body. The scanner returns that token only if expected by the parser and some nested parser succeeds. We will illustrate the technique through three examples: an inherently ambiguous language, a non LR(K) language and the PL/I example.

An Inherently Ambiguous Language

A context-free language is inherently ambiguous if all context-free grammars generating that language are ambiguous. While some context-free languages have both ambiguous and unambiguous grammars, there are context-free languages for which no unambiguous context-free grammar can exist. An example of an inherently ambiguous lan-

[1] In this and the next algorithms we will follow these conventions: Q is the set of the states of the LR automaton, Σ is the set of tokens or grammar terminals, V is the set of non-terminals, P is the set of grammar productions and *ACTION* denotes the LR action table:

$$ACTION : Q \times \Sigma \to Q \cup P$$

Where the kind of action is either a shift or a reduce: $ACTION(q,a) = shift\ q' \in Q$ or $ACTION(q,a) = reduce\ by\ A \to \alpha \in P$. The table $GOTO : Q \times V \to Q$ is the LR automaton transition table restricted to V: $GOTO(q,A) = q' \in Q$.

guage is the set $\{a^n b^m c^r$ such that $n = m$ or $m = r$ and $n, m, r \geq 0\}$ which is context-free, since it is generated and solved by the following `eyapp` grammar [8] (download it from [9]):

```
1  %token AB=%//=ab
2  %token a b c
3  %%
4  s:    AB aeqb     | beqc   ;
5  aeqb: ab cs                ;
```

```
6  ab:   /* empty */ | a ab b ;
7  cs :  /* empty */ | cs c   ;
8  beqc: as bc                ;
9  bc:   /* empty */ | b bc c ;
10 as:   /* empty */ | as a   ;
11 %%
```

The lexical analyzer is automatically generated by the compiler from the token declarations. The order in which the token declarations are checked is as follows: Single quoted string tokens inside the body of the grammar are checked first, if two patterns match the same string, the longest match wins. The other declared tokens `%token TOKENNAME = ...` are processed according to their appearance inside the text.

To help the parser, we create a fictitious token `AB` which is used at line 4 to decide which of the two productions applies. When the regular expresion - in this case empty - defining the token, (line 1) is prefixed by a `%`, the token is returned only if it is expected by the syntactic analyzer. When the regular expression defining a token is postfixed by an '=' followed by a syntactic variable - as also occurs in line 1, where is followed by `ab`- the token is returned only if the input that follows matches the language defined by that variable (that is, by `ab`: when the number of `a`'s matches the number of `b`s). The definition of token `AB` at line 1 is translated by the compiler into the following pseudo-code fragment of the lexical analyzer[2]:

```
if (self.expects('AB')) {
   if (self.YYPreParse('ab')) { return 'AB' } }
```

Where `self` denotes the parser object and the call to method `expects` returns the set of valid tokens. The call `YYPreParse('ab')` checks that a prefix of the incoming input belongs to the language defined by `ab`.

A Non LR(k) Gramamr

The ficticious token strategy used in the previous example can also be used to fix non LR(k) grammars. The grammar on the left side of the figure below (see [9]) can not be parsed by any LR(k) parser without the presence of the ficticious token `CsD`. The grammar on the right side contains the definitionof `CsD`:

```
1  %token CsD = %//=CsD
2  %token c d f x y
3  %%
4  A: B CsD C d   | E C f ;
5  B: x y                 ;
6  E: x y                 ;
7  C: /* empty */ | C c   ;
8  %%
```

```
1
2  %token c d f x y
3  %%
4  CsD: Cs 'd'                ;
5  Cs:  /* empty */ |  Cs 'c' ;
6  %%
```

Solving the PL/I Problem

The `eyapp` [8] grammar below (download it from [9]) presents a simplified version of the PL/I problem and solves it. The `%token ID` declaration at line 3 is translated by the compiler into a code fragment that checks if the current

[2] For the sake of clarity, code related with the maintenance of the current scanning position has been omitted

unexpended input matches the regular expression:

```
/\G([a-zA-Z_][a-zA-Z_0-9])/gc and return ('ID', $1);
```

The expression is evaluated in short-circuit. The anchor `\G` stands for *the current position inside the input*.

```
1  %token then = %/(then)/
2  %token if   = %/(if)/=expr_then
3  %token ID   =  /([a-zA-Z_][a-zA-Z_0-9]*)/
4  %%
5  stmt:       ifstmt    | assignstmt ;
6  ifstmt:     if expr then stmt      ;
7  assignstmt: id '=' expr            ;
8  expr:       id '=' id | id         ;
9  id:         ID                     ;
10 %%
```

When the token definition, as in lines 1 and 2 is prefixed by a `%`, the token is returned only if it is expected by the syntactic analyzer. It is translated by the compiler into:

```
self.expects('then') and /\G(then)\b/gc and return 'then';
```

The definition of token `if` at line 2 says that an input '`if`' stands for the keyword if, and only if, the token is expected by the syntax analyzer and is followed by a correct expression followed by the keyword `then`. The grammar for variable `expr_then` is as follows:

```
1  %token then = %/(then)\b/
2  %token ID   = /([a-zA-Z_])\w*/
3  %%
4  expr_then: expr then ; expr: id '=' id | id ; id: ID ;
5  %%
```

CONCLUSIONS

We have presented a new algorithm to compute the exact list of tokens expected by the syntax analyzer at any point of the scanning process. As an application, we have explained how this knowledge can be used to parse ambiguous languages.

REFERENCES

1. J. Aycock, and R. N. Horspool, "Schrödinger's token," in *Software, Practice & Experience.*, 2001, pp. vol. 31 803–814.
2. M. van den Brand, J. Scheerder, J. Vinju, and E. Visser., "Disambiguation filters for scannerless generalized LR parsers.," in *Proc. 11th International Conf. on Compiler Construction*, volume 2304 of LNCS, 2002, pp. 143–158.
3. A. Begel, and S. L. Graham, "XGLR - an algorithm for ambiguity in programming languages," in *Science of Computer Programming, 61(3)*, 2006, pp. 211–227.
4. A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*, Addison Wesley, 1986.
5. V. Wyk, E. R., and A. C. Schwerdfeger, "Context-aware scanning for parsing extensible languages," in *Proceedings of the 6th international conference on Generative programming and component engineering. (GPCE '07)*, Salzburg, Austria, 2007, pp. 63–72.
6. B. Ford, "Parsing expression grammars: a recognition-based syntactic foundation," in *SIGPLAN Not., 39(1)*, 2004, pp. 111–122.
7. R. Grimm, "Better extensibility through modular syntax," in *In PLDI 2006: ACM SIGPLAN conference on Programming language design and implementation*, New York, 2006, pp. 38–51.
8. C. Rodríguez-León, Parse::Eyapp manuals (2007), [Online]. Available at CPAN: `http://search.cpan.org/dist/Parse-Eyapp/`.
9. C. Rodríguez-León, and L. García-Forte, A repository of LALR conflictive grammars (2010), [Online]. Available at google-code: `http://code.google.com/p/grammar-repository/`.