

Parsers

Dan Beatty

The Role of the Parser

“The parser obtains a string of tokens from the lexical analyzer, and verifies that the string can be generated by the grammar for the source language. ” It is expected that the parser should recover from commonly occurring errors so that it can continue processing the remainder of its input.

The parser is typically applied to context free grammars.

“Why use regular expressions to define the lexical syntax of a language?”

1. The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as powerful as grammars
2. Regular expression generally provide a more concise and easier to understand notation for tokens than grammars.
3. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.
4. Separating the syntactic structure of a language into lexical and non-lexical parts provides a convenient way of modularizing the front end of a compiler into two manageable-sized components.
5. Regular expressions are most useful for describing the structure of lexical constructs such as identifiers, constants, keywords, and so forth.
6. Grammars are useful in describing nested structures such as balanced parentheses, matching begin and end statements, corresponding if-then-else statements, and the like.

Elimination of Left Recursion Eliminating Left Recursion is one of the steps for preparing a CFG to be parsed. The general rule is for mapping $A \rightarrow A\alpha|\beta$ to a non-left recursive production

- $A \rightarrow \beta A'$
- $A' \rightarrow \alpha A'|\epsilon$

Left Factoring Why is left factoring useful as a grammar transformation? When it is not clear which of two alternative productions to use to expand a nonterminal A , we may be able to re-write the A -productions to defer the decision until we have seen enough of the input to make the right choice. Left-factoring is also known as removing common prefixes.

Given a production of the form $A \rightarrow \alpha\beta_1|\alpha\beta_2|\gamma$

Transformed to: Input: Grammar G

- Output: An equivalent left-factored grammar.
- $A \rightarrow \alpha A'|\gamma$
 - $A' \rightarrow \beta_1|\beta_2$

Method: For each non-terminal A find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ i.e., there is a nontrivial common prefix, replace all the A productions $A \rightarrow \alpha\beta_1|\alpha\beta_2|\gamma$ where γ represents all alternatives that do not begin with α by

- $A \rightarrow \alpha A'|\gamma$
- $A' \rightarrow \beta_1|\beta_2$

Here A' is a new non-terminal. Repeatedly apply this transformation until no two alternatives for a non-terminal have a common prefix.

Top Down Parsing Notes

Purpose: To introduce the basic ideas behind top down parsing and show how to construct an efficient non-backtracking form of top-down parser called a predictive parser.

Goal: Construct a parse tree for an input string from the root and create the nodes of the parse tree in pre-order.

Buzz words

- Predictive parsing
- Recursive descent

Backtracking

- Seldom necessary for context free grammar.
- inefficient on CSG

A left-recursive grammar can cause a recursive-descent parser, even one with back-tracking, to go into an infinite loop.

Predictive Parser Qualities:

- Eliminates left recursion
- left factors
- Proper alternatives are detected by examining the first symbols derives.

Differences between lexical analyzer and predictive parser transition

- One diagram for each non-terminal
- Labels are both tokens and non-terminals
- Transition on tokens next symbols
- Transition on non-terminal equates to a procedure calls

To build a syntax analyzer

1. Apply left recursion removal
2. Left factor the grammar
3. Create initial and final (return) state
4. For each production, $A \rightarrow X_1, X_2, \dots, X_n$ create a path from the initial to the final state with edges labeled X_1, X_2, \dots, X_n .

What happens for a predictive parser which works off the transition diagram?

1. It begins in the start state for the start symbol.
2. After some actions, the parser arrives in state S .
 - Possibility: There is an edge to t labeled a . If the next input symbol is a , then the transition to t occurs.
 - Possibility: There is an edge to t labeled by non-terminal A
 - The parser goes to the start state of A without moving the input cursor.
 - If a final state in A is reached, the transition is made
 - Possibility: There is an edge to t labeled by ϵ . Transition is made without advancing the input cursor.

Big idea “A predictive parsing program based on a transition diagram attempts to match terminal symbols against the input, and makes a potentially recursive procedure call, whenever it has to follow an edge labeled by a nonterminal.”

- For non-determinism this does not work
- For deterministic automata this can work.

Non-recursive Predictive Parsing A stack may be used explicitly to build a non-recursive predictive parser.

- Problems determining the production to be applied
- Solution: parse table look up- table driven predictive parser.

First, Follow, and Selection Sets

- First and follow sets form functions associated with the grammar and aids in the construction of a predictive parser by filling in the predictive parsing table.
- The $FIRST(\alpha)$ is the set of terminals that begin the set of strings derived from α
- The $FOLLOW(A)$ for a non-terminal A , is the set of terminals that can appear immediately to the right of A in some sentential form. The set of terminals of a such that there exists a derivation of the form $S \rightarrow^* \alpha A a \beta$ for some α and β

Process to Compute $FIRST(X)$

1. If X is terminal, then $FIRST(X) = \{X\}$
2. If $X \rightarrow \epsilon$ is a production, then add ϵ to $FIRST(X)$.
3. If X is a non-terminal and $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production then $\forall Y_i \in X \rightarrow Y_i$ $FIRST(X)_+ = FIRST(Y)$.

Process to compute $FOLLOW(A)$

1. Place end of string \$ in $FOLLOW(S)$ where S is the start symbol.
2. If there is a production $A \rightarrow \alpha B \beta$ where $\epsilon \notin FIRST(\beta)$ then make the following statement so: $FOLLOW(B) = FOLLOW(B) \cup FIRST(\beta)$
3. If there is a production $A \rightarrow \alpha B \beta$ where $\epsilon \in FIRST(\beta)$ then make the following statement so: $FOLLOW(B) = FOLLOW(A) \cup FOLLOW(B)$

Selection Sets (reference Cooke page IV-24)

1. The selection set of a non- ϵ right hand side option is the option's first set.
2. The selection set of a ϵ right hand side option is the option's follow set.

“To complete the syntax analysis routines, one follows clearly defined rules for implementing the grammar specification”

\forall LHS (or element of C) there is a boolean function: \forall RHS option of a given LHS there will be an if-statement block and if a given RHS option has multiple symbols, there will be, in that block, a nested if for each symbol. For a given RHS symbol there are three possible actions:

1. In the case of matching an element of the Vocabulary, V such that if $lex \in V : ,$ consume the element (invoke the LA) and if the last item in the RHS option set the LHS symbol to true. If you were supposed to match an element of V and did not, set LHS to false and output an error message.
2. In the case where the RHS is an element of the syntactic category, C : invoke C 's function to see if it is true. If true and if at the end of the RHS option, set the LHS symbol to true otherwise set the LHS to false.
3. In the case of an epsilon production, check to see if lex is an element of the selection set. If it is, set the LHS to true. Otherwise set LHS to false and put out an error message.

Reserved Words and Symbols

$\alpha_0 = \text{program}$	$\alpha_1 = \text{array}$	$\alpha_2 = \text{integer}$	$\alpha_3 = \text{read}$	$\alpha_4 = \text{write}$	$\alpha_5 = \text{rdln}$
$\alpha_6 = \text{wrln}$	$\alpha_7 = \text{when}$	$\alpha_8 = \text{until}$	$\alpha_9 = \text{from}$	$\alpha_{10} = \text{..}$	$\alpha_{11} = \text{eof}$
$\alpha_{12} = \{$	$\alpha_{13} = \}$	$\alpha_{14} = [$	$\alpha_{15} =]$	$\alpha_{16} = ;$	$\alpha_{17} = ,$
$\alpha_{18} \leftarrow ($	$\alpha_{19} \leftarrow)$	$\alpha_{20} \leftarrow :=$	$\alpha_{21} \leftarrow <$	$\alpha_{22} \leftarrow >$	$\alpha_{23} \leftarrow =$
$\alpha_{24} \leftarrow <=$	$\alpha_{25} \leftarrow >=$	$\alpha_{26} \leftarrow <>$	$\alpha_{27} \leftarrow +$	$\alpha_{28} \leftarrow -$	$\alpha_{29} \leftarrow *$
$\alpha_{30} \leftarrow /$	$\alpha_{31} \leftarrow :$	$\alpha_{32} \leftarrow \text{and}$	$\alpha_{33} \leftarrow \text{or}$	$\alpha_{34} \leftarrow \text{not}$	$\alpha_{35} \leftarrow \text{mod}$

Table 1: Selection Sets for Given Grammar

Production	First	Follow	Selection
$P \rightarrow \alpha_0 D_1 \alpha_{12} S_L \alpha_{13}$	α_0	$\$$	α_0
$D_1 \rightarrow I_L \alpha_{31} D$	$\epsilon, \alpha_{-1}, \alpha_{31}$	α_{12}	$\epsilon, \alpha_{-1}, \alpha_{31}$
$D_1 \rightarrow \epsilon$			
$D \rightarrow \alpha_1 \alpha_{14} \alpha_{-2} D_2 \alpha_{15} \alpha_{16} D_1$	α_1, α_2	α_{12}	α_1
$D \rightarrow \alpha_2 \alpha_{16} D_1$	α_2		α_2
$D_2 \rightarrow \alpha_{17} \alpha_{-2} D_2$	ϵ, α_{17}	α_{15}	α_{17}
$D_2 \rightarrow \epsilon$		α_{15}	
$I_L \rightarrow \alpha_{-1} I_{L_1}$	α_{-1}	α_{19}, α_{31}	α_{-1}
$I_{L_1} \rightarrow \alpha_{17} I_L$	$\alpha_{17} \epsilon$	α_{19}, α_{31}	α_{17}
$I_{L_1} \rightarrow \epsilon$			α_{19}, α_{31}
$S_u \rightarrow \alpha_{-1} S'_u$	α_{-1}, α_{-2}	α_{15}	α_{-1}
$S_u \rightarrow \alpha_{-2} S''_u$			α_{-2}
$S'_u \rightarrow \alpha_{17} S_u$	α_{17}		α_{17}
$S''_u \rightarrow \alpha_{17} S_u$	α_{17}		α_{17}
$S \rightarrow \alpha_{12} S_L \alpha_{13} S'$	$\alpha_5, \alpha_6, \alpha_{12}, \alpha_{18}$	α_{13}	α_{12}
$S \rightarrow \alpha_5$	α_5	α_{13}	α_5
$S \rightarrow \alpha_6$	α_6	α_{13}	α_6
$S \rightarrow \alpha_{18} I_L \alpha_{19} S''$	α_{18}		α_{18}
$S' \rightarrow \alpha_7 C$	$\alpha_7, \alpha_8, \alpha_{-1}$	α_{13}	α_7
$S' \rightarrow \alpha_8 C$	α_8	α_{13}	α_8
$S' \rightarrow \alpha_{-1} \alpha_9 \alpha_{14} E \alpha_{10} E \alpha_{15}$	α_{-1}	α_{13}	α_{-1}
$S'' \rightarrow \alpha_{18} E_L \alpha_{19} \alpha_{20}$	$\alpha_{18}, \alpha_3, \alpha_4$	α_{13}	α_{18}
$S'' \rightarrow \alpha_3$	α_3	α_{13}	α_3
$S'' \rightarrow \alpha_4$	α_4	α_{13}	α_4
$T \rightarrow I_D$	α_{-1}, α_{-2}	$\alpha_{10}, \alpha_{15}, \alpha_{17}, \alpha_{19}$	α_{-1}
$T \rightarrow \alpha_{11}$	α_{11}	$\alpha_{10}, \alpha_{15}, \alpha_{17}, \alpha_{19}$	α_{11}
$T \rightarrow \alpha_{-2}$	α_{-2}	$\alpha_{10}, \alpha_{15}, \alpha_{17}, \alpha_{19}$	α_{-2}
$I'_D \rightarrow \alpha_{-1} I'_D$	α_{-1}	$\alpha_{10}, \alpha_{15}, \alpha_{17}, \alpha_{19}$	α_{-1}
$I'_D \rightarrow \alpha_{14} S_u \alpha_{15}$	$\alpha_{-1}, \alpha_{-2}, \epsilon$	$\alpha_{10}, \alpha_{15}, \alpha_{17}, \alpha_{19}$	$\alpha_{-1}, \alpha_{-2}, \epsilon$
$I'_D \rightarrow \epsilon$		$\alpha_{10}, \alpha_{15}, \alpha_{17}, \alpha_{19}$	$\alpha_{10}, \alpha_{15}, \alpha_{17}, \alpha_{19}$
$E \rightarrow \alpha_{18} E_L \alpha_{19} \alpha_{18} E_L \alpha_{19} E'$	α_{18}	$\alpha_{15}, \alpha_{16}, \alpha_{17}, \alpha_{19}$	α_{18}
$E \rightarrow T$	$\alpha_{11}, \alpha_{-2}, \alpha_{-1}$	$\alpha_{15}, \alpha_{16}, \alpha_{17}, \alpha_{19}$	$\alpha_{11}, \alpha_{-2}, \alpha_{-1}$
$E' \rightarrow \alpha_{27}$	α_{27}	$\alpha_{15}, \alpha_{16}, \alpha_{17}, \alpha_{19}$	α_{27}
$E' \rightarrow \alpha_{28}$	α_{28}		α_{28}
$E' \rightarrow \alpha_{29}$	α_{29}		α_{29}
$E' \rightarrow \alpha_{30}$	α_{30}		α_{30}
$E' \rightarrow \alpha_{35}$	$\alpha_{27}, \alpha_{28}, \alpha_{29}, \alpha_{30}, \alpha_{35}$	$\alpha_{15}, \alpha_{16}, \alpha_{17}, \alpha_{19}$	α_{27}
$E_L \rightarrow E E'_L$	$\alpha_{18} \alpha_{11}, \alpha_{-2}, \alpha_{-1}$		$\alpha_{18}, \alpha_{11}, \alpha_{-2}, \alpha_{-1}$
$E_L \rightarrow \epsilon$	$\alpha_{11}, \alpha_{-2}, \alpha_{-1} \epsilon$	α_{19}	α_{19}
$E'_L \rightarrow \alpha_{17} E_L$	α_{17}	$\alpha_{11}, \alpha_{-2}, \alpha_{-1}, \alpha_{19}$	α_{17}
$E'_L \rightarrow \epsilon$			$\alpha_{11}, \alpha_{-2}, \alpha_{-1}, \alpha_{19}$
$C \rightarrow \alpha_{18} E_L \alpha_{19} \alpha_{18} E_L \alpha_{19} C''$	α_{18}	$\alpha_{32}, \alpha_{33}, \alpha_{34}, \alpha_{13}$	α_{18}

Table 2: Selection Sets for Given Grammar

Production	First	Follow	Selection
$C' \rightarrow \alpha_{16} C C'''$	α_{16}	$\alpha_{32}, \alpha_{33}, \alpha_{34}, \alpha_{13}$	α_{16}
$C'' \rightarrow \alpha_{21} C'$	α_{21}		α_{21}
$C'' \rightarrow \alpha_{22} C'$			α_{22}
$C'' \rightarrow \alpha_{23} C'$			α_{23}
$C'' \rightarrow \alpha_{24} C'$			α_{24}
$C'' \rightarrow \alpha_{25} C'$			α_{25}
$C'' \rightarrow \alpha_{26} C'$	$\alpha_{21}, \alpha_{22}, \alpha_{23}, \alpha_{24}, \alpha_{25}, \alpha_{26}$	$\alpha_{32}, \alpha_{33}, \alpha_{34}, \alpha_{13}$	α_{26}
$C''' \rightarrow \alpha_{32} C''$	α_{32}		α_{32}
$C''' \rightarrow \alpha_{33} C''$	α_{33}		α_{33}
$C''' \rightarrow \alpha_{34} C''$	$\alpha_{32}, \alpha_{33}, \alpha_{34}$	$\alpha_{32}, \alpha_{33}, \alpha_{34}, \alpha_{13}$	α_{34}
$S_L \rightarrow S S'_L$	$\alpha_5, \alpha_6, \alpha_{12}, \alpha_{18}$	α_{13}	$\alpha_5, \alpha_6, \alpha_{12}, \alpha_{18}$
$S'_L \rightarrow \alpha_{16}$	α_{16}	α_{13}	α_{16}
$S'_L \rightarrow \epsilon$	ϵ	α_{13}	α_{13}

There was a grammar change made on the grammar. Watch out for this an update the syntax analysis. There is direct recursion on both the C and E productions that must be removed. There was a question as to what to do with the E production. The question was described as question of semantics and was postponed for that discussion.

An example of left recursion is provided with production $C \rightarrow (E_L)(E_L) < |C C \text{ and}$. There is obvious left recursion. There is a non-left-recursion option that used to define C: $C \rightarrow (E_L)(E_L) < C'$ and a the left recursion options which fill in C' : $C' \rightarrow C \text{ and } C'|\epsilon$ This opens the need to work the first, follow and selection sets for these values:

$C \rightarrow (E_L)(E_L) < C C \text{ and}$	FIRST	FOLLOW	SELECTION
$C \rightarrow (E_L)(E_L) < C'$	(\$, and	
$C' \rightarrow C \text{ and } C'$	(
$C' \rightarrow \epsilon$	\$, and		

Syntax Analysis Completion

\forall LHS (or element of C) there is a boolean function: \forall RHS option of a given LHS there will be an if-statement block and if a given RHS option has multiple symbols, there will be, in that block, a nested if for each symbol. For a given RHS symbol there are three possible actions:

1. In the case of matching an element of the Vocabulary, V such that if $lex \in V : ,$ consume the element (invoke the LA) and if the last item in the RHS option set the LHS symbol to true. If you were supposed to match an element of V and did not, set LHS to false and output an error message.
2. In the case where the RHS is an element of the syntactic category, C : invoke C's function to see if it is true. If true and if at the end of the RHS option, set the LHS symbol to true otherwise set the LHS to false.
3. In the case of an epsilon production, check to see if lex is an element of the selection set. If it is, set the LHS to true. Otherwise set LHS to false and put out an error message.

Viable prefix error - horrid death problem

$LL(1)$ viable prefix Error recovery is an issue. If there is a syntax error, the compiler simply dies and will not catch any other mistakes. The premise of viable prefix is establish the sections of the code that were considered to be correct prior to the error. After that point identify a position which the compiler can continue with some kind of error recovery (not error correction). .

Scan set are needed to provide viable prefixes and is associated with the syntax analyzer. What a scan set does provides a place where the lexical analyzer can pushed to, and from that point identify to section to be good, and continue compilation.

Example: $S \rightarrow S|S_L$

In the case of this production, grammar specifies the semicolon and end bracket as valid end points. Thus after that point, compilation can begin again and everything would be okay. In developing these scan sets, we look at each of the productions, look at the point that can be scanned to, find its end , and restart compilation. The ; and } come from the production of


$S \rightarrow a_{12}S_La_{13}a_7C|a_{12}S_La_{13}a_8C|$

$S \rightarrow a_{18}I_La_{19}a_{18}E_La_{19}a_{20}|a_{12}S_La_{13}i_da_9a_{14}Ea_{10}Ea_{15}|$

$S \rightarrow a_{12}I_La_{13}a_3|a_{18}I_La_{19}a_4|a_5|a_6$

In reality, a scan set could nothing more than $\{ \} ,$ and be okay. Similar things need to be done for declarations. For example, S_L and S . An example of this would a recursive decent routine for S_L

```
function  $S_L$ : boolean
if  $S$  then
if nt = ';' then
if  $S_L$  then  $S_L := \text{true}$ 
else
 $S_L := \text{true}$ 
numbererrors := numbererrors
scan ( } ; )
else
 $S := \text{true}$ 
error
numbererror :=
scan
else
 $S_L := \text{true}$ 
```



Note that it is probable that if an error is to be had, it will be had a the semicolon before the else. Thus the else could be a no operation.