Notes on Table Driven - Bottom Up Parsers aka Simple LR Parsers

Daniel Beatty

May 5, 2006

Parsing Concepts:

1. Syntax Directed Definitions
2. Translation Schemes

Things these concepts have in common:

1. Both parse an input token stream
2. Build a parse tree
3. Traverse tree required to execute semantic actions

Syntax directed definitions also have dependency graphs and syntax tree construction (page 284, 287). Translation routines that are involved during parsing have two restrictions:

1. Grammar suitable for parsing may not be reflect the natural hierarchical structure of the langauage constructs.
2. Parsing method constrains the order in which nodes are evaluated.

0.1 Action and Goto Tables

Thee is a generic algorithm that takes as input a matching action and goto table plus an input stream. Its output is a parsed stream and state of acceptance. (Cooke Lecture #12 time 5:45)

Table based parsing has a stack, input stream. These two determine the present state. The algorithm consumes information from the input stream, pushes and pops information to the stack, and uses the states to determine whether a state and input are pushed on to the stack or whether to pop information off of the stack. The shifts push states on to the stack, consumes the next token, and pushes that token on to the stack. The shift information encoded in the table has the next state. A reduce has both the production and sub-production indicated in the entry. The size of the production determines how much to pop. Also, if any semantic actions occur at the end of the production, then those get called as well. Note that table based parsing forbids "embedded semantic actions".

State	Operand	(	+	-	*	div	)	$
0	S5	S4						
1			S7	S8				ACCEPT
2			R1.1	R1.1	S10	S11	R1.1	R1.1
3			R2.1	R2.1	R2.1	R2.1	R2.1	R2.1
4	S5	S4						
5			R3.2	R3.2	R3.2	R3.2	R3.2	R3.2
6	S5	S4						
7	R4.1	R4.1						
8	R4.2	R4.2						
9	S5	S4						
10	R5.1	R5.1						
11	R5.2	R5.2						
12			S7	S8			S24	
17			R1.2	R1.2	S10	S11	R1.2	R1.2
21			R2.2	R2.2	R2.2	R2.2	R2.2	R2.2
24			R3.1	R3.1	R3.1	R3.1	R3.1	R3.1

Figure 1: Action Table for example grammar

A good question is does table based semantics allow for epsilon productions which could be embedded which would generate "embedded semantic actions?" Answers is yes, but why do it. The only conceivable benefit has to do with embedded semantic actions. An ϵ production may be a good choice in these categories since they force selections to handle them. A compromise would be short "cheese" embedded productions which are not necessary for the syntax, but payoff in the semantic action section.

source (Cooke's lecture, and Knuth's Dissertation)

State	EXPRESS	TERM	FACTOR	MDOP	ASOP
0	1	2	3		
1					6
2				9	
3					
4	12	2	3		
5					
6		17	3		
7					
8					
9			21		
10					
11					
12					6
17				9	
21					
24					

Figure 2: Goto table for example grammar

0	E'	::=	EXPRESS		
1	EXPRESS	::=	TERM		
		::=	EXPRESS	ASOP	TERM
2	TERM	::=	FACTOR		
		::=	TERM	MDOP	FACTOR
3	FACTOR	::=	(	EXPRESS	)
		::=	OPERAND		
4	ASOP	::=	plus		
		::=	minus		
5	MDOP	::=	mul		
		::=	div		

Figure 3: Reduction Table for an Example Grammar

Algorithm 1 Table Driven Parse

```
ACCEPT:=FALSE;
ERROR := FALSE;
push (s_0)
repeat
  Examine next input symbol = a_i and state s_m = TOP(Stack);
  if ACTION[s m , a i ] = a. shift s  then
    consume a i
    push(a i)
    push(s )
  else
    if reduce B → β  then
      pop 2r items from stack where r = |β|
      s = GOTO[smr, B] where s, m  r = TOP(stack after the 2r items are popped)
      push(B)
      push(s )
    else
      if accept  then
        ACCEPT:=TRUE
      else
        if error  then
          ERROR:=TRUE
        end if
      end if
    end if
  end if
until ACCEPT or ERROR
```

Semantic actions can only occur in reductions, in particular at the end of the reductions. The heavy lifting by the Table Driven Parse is done by the shifts and reduce. The three actions for a shift are:

1. consume a_i:
   This function calls the lexical analyzer's lex operation. As a result, the value of a_i is made available to the algorithm.

2. push(a_i)
   The a_i is pushed on to the parsing stack.

3. push(s_i)
   Both the current state and next state are available as result of the iteration of the algorithm. These two states were made available by the top of the stack which has a state at the end of each iteration, and the reference into the action table. The reference into the token are from the state s_m from the top of the stack, and the consumed token a_i. In the case of the shift encoding, the next state is also include which is denoted by s_i. This s_i is pushed and will be referenced in the next iteration.

In the case of reduce, the elements included with the production are B which is the production, and *beta* the specific sub-production of B. The consequences of this reduction action are as follows:

1. pop $2r$ items from stack where $r = |\beta|$
   Every sub-production has a length (number of terminals and non-terminals). The value for this is $r = |\beta|$. The parsing stack is simply popped $2r$ times. The values of this popping may be stored in array and sent into semantic analyzer as an argument for the semantic action associated with the production.

2. $s = \text{GOTO}[s_{mr}, B]$
   where $s_{mr} = TOP$ (stack after the 2r items are popped). Note that the first operation popped off the elements associated with a sub-production. The goto table has reference information for the current stack top state s_{m-r} and where the input stream could be next.

3. push(B)
   The production itself is pushed and happens to define particular action that may be used later in some semantic action.

4. push(s)
   The goto state is pushed and is the next top state for use in the next iteration.

One use analysis of Dr. Knuth's algorithm is the complexity of this algorithm. The complexity is based off of the number of productions and the length of the input stream (token wise). Error recovery may also be a useful analysis of this algorithm.

Example: Try the example tables with the following sentence in the calculator grammar:

a + b

As a result the following table is generated indicating the steps taken during the execution of Dr. Knuth's algorithm.

Table 1: Example execution of Dr. Knuth's algorithm for a + b in the calculator grammar.

Step	Stack (left to right)	Input Stream	Action table value
1	S_0	a + b $	S_5
2	S_0 a S_5	+ b $	R3.2
3	S_0, Factor , S_3	+ b $	R2.1
4	S_0 , Term , S_2	+ b $	R1.1
5	S_0 , Express, S_1	+ b $	S7
6	S_0 , Express, S_1, +, S_7	b $	R4.1
7	S_0 , Express, S_1, ASOP, S_6	b $	S5
8	S_0 , Express, S_1, ASOP, S_6 b S_5	$	R3.2
9	S_0 , Express, S_1, ASOP, S_6, Factor, S_3	$	R2.1
10	S_0 , Express, S_1, ASOP, S_6, Term S_{17}	$	R1.2
11	S_0 , Express S_1	$	Accept

Again, the calculator grammar under table parsing can not use embedded semantic actions without embedding simple productions. In this case, there is such a thing. The ASOP and MDOP productions are simple productions for plus, minus, multiply and divide symbols. A push operation on the semantic action stack can be used with these productions. The factor is a little more complex. In its operand production, it is also a push. However, most of it is simply reclaiming items from the semantic action stack. It is the reclaiming that allows such semantic actions to properly process what is parsed. In the case of terms and expressions, they reclaim both the operation and operands from the semantic action stack. Thus Knuth's algorithm is augmented to include semantic actions:

Algorithm 2 Table Driven Parse with Semantic Action Calls

```
ACCEPT:=FALSE;
ERROR := FALSE;
push (s_0)
repeat
  Examine next input symbol = a_i and state s_m = TOP(Stack);
  if ACTION[s m , a i ] = a. shift s  then
    consume a i
    push(a i)
    push(s )
  else
    if reduce B → β  then
      Semantic Action for B → β.
      pop 2r items from stack where r = |β|
      s = GOTO[smr, B] where s, m  r = TOP(stack after the 2r items are popped)
      push(B)
      push(s )
    else
      if accept  then
        ACCEPT:=TRUE
      else
        if error  then
          ERROR:=TRUE
        end if
      end if
    end if
  end if
until ACCEPT or ERROR
```

source (Cooke's lecture, and Knuth's Dissertation)

Another example: (a + b) * c . Also genquads for the following actions:

Table 2: Productions and Semantic Rules for the example calculator grammar

Production Number	Production	Follow Set	Semantic Rules
0	$E' \rightarrow E$		
1.	$E \rightarrow T$	+, -) $	
	$E \rightarrow EAT$		[imCode op]
2.	$T \rightarrow F$	+ , - , *, div ,) $	
	$T \rightarrow TMF$		[imCode op]
3.	$F \rightarrow (E)$	+ , - , *, div ,) $	
	$F \rightarrow o_p$		[imCode p]
4.	$A \rightarrow +$	o_p , (	[imCode p]
	$A \rightarrow -$		[imCode p]
5	$M \rightarrow *$	o_p , (	[imCode p]
	$M \rightarrow \text{div}$		[imCode p]

Examples from Ullman, page 190-192 and 216-220

Table 3: Another example for (a + b) * c

Step	Stack	Input	State
1	S_0	(a + b) * c $	S4
2	S_0 (S_4	a + b) * c $	S5
3	S_0 (S_4 a S_5	+ b) * c $	R3.2
4	S_0 (S_4 F S_3	+ b) * c $	R2.1
5	S_0 (S_4 T S_2	+ b) * c $	R1.1
6	S_0 (S_4 E S_{12}	+ b) * c $	S7
7	S_0 (S_4 E S_{12} + S_7	b) * c $	R4.1
8	S_0 (S_4 E S_{12} A S_6	b) * c $	S5
9	S_0 (S_4 E S_{12} A S_6 b S_5	) * c $	R3.2
10	S_0 (S_4 E S_{12} A S_6 F S_3	) * c $	R2.1
11	S_0 (S_4 E S_{12} A S_6 T S_{17}	) * c $	R1.2
12	S_0 (S_4 E S_12	) * c $	S24
13	S_0 (S_4 E S_12) S_{24}	* c $	R3.1
14	S_0 F S_3	* c $	R2.1
15	S_0 T S_2	* c $	S10
16	S_0 T S_2 * S_{10}	c $	R5.1
17	S_0 T S_2 M S_9	c $	S5
18	S_0 T S_2 M S_9 c S_5	$	R3.2
19	S_0 T S_2 M S_9 F S_{21}	$	R2.2
19	S_0 T S_2	$	R1.1
20	S_0 E S_1	$	Accept

0.2 Constructing the Simple Left to Right Shift Reduce (SLR) Parsing Table

The key for Simple LR parsing to work is to use an algorithm that recognizes viable prefixes of the grammar. This is done by using tables (action and goto) which is derived from a deterministic finite automation. Ullman places in a disclaimer with this algorithm in that "will not produce uniquely defined parsing action tables for all grammars". LL1 is not a problem. Context Free Grammars are not a problem. Follow sets are required for this grammar. First sets are not. However, first and follow sets will yield selection sets. However, the LL1 condition is not required on these first and follow sets.

- Given a grammar G, apply an augment G' that produces G.
  - "if G is a grammar with start symbol S, then G' , then G', the *augmented grammar* for G, is G with a new start symbol S' and production $S' \rightarrow S$. "
  - "The purpose of this new starting productions is to indicate to the parser when it should stop parsing and announce acceptance of the input."
- The "LR parser can determine from the state on top of the stack everything that it needs to know about what is in the stack." Furthermore a "grammar that can be parsed by an LR parser examining up to k input symbols on each move is called an LR(k) grammar."'
- "An LR(0) item (item for short) of a grammar G is a production of G with a dot at some position of the right side."
- The central idea for the SLR parser is "first construct from the grammar a deterministic finite automaton to recognize viable prefixes."
- canonical LR(0) collections provides the basis for constructing SLR parsers.
- Closure categories:
  - *Kernel item* include the initial item, and all items whose dots are not at the left end.
  - *Nonkernel items* which have their dots at the left ends.

The collection algorithm uses the augmented production to start the Collection process. The closure function will move the period around. Intuitively this period can be thought of as a special delimiter. It allows for all of the LR(0) items to be found.

Algorithm 3 Function collection(G:grammar): collection of sets of items

Collection:= {Closure({$A \rightarrow .A$}) };
repeat
  For each set of items I in Collection and each symbol X
  if GOTO (I,X) $\neq$ null is not in Collection **then**
    Add GOTO(I,X) to Collection;
  end if
until no more Is are added to Collection
return Collection

Algorithm 4 Function Closure(I:set of Items): set of items;

repeat
  For each item $B \rightarrow \alpha.C\beta \in I$ and each production $C \rightarrow \gamma \in G$
  if $C \rightarrow .\gamma \ni I$ **then**
    Add $C \rightarrow .\gamma$ to I;
  end if
until no more items can be added to I
return Closure := I;

Algorithm 5 Function Goto(I:set of Items, X: symbol in the grammar): set of items;

S = { } ;
for all $B \rightarrow \alpha.X\beta \in I$ **do**
  S := $S\cup$ Closure ($\{B \rightarrow \alpha X.\beta\}$)
end for
return goto := S;

Algorithm 6 Function Action Tables

if $[B \rightarrow \alpha.b\beta] \in I_i$ and GOTO$(I_i, b) = i_j$ **then**
  ACTION$[i, b]$:= s_j
end if
if $[B \rightarrow \alpha.] \in l_i]$ **then**
  ACTION$[i, b]$:= r_n
end if

Creating Action Table

1. if $[B \rightarrow \alpha.b\beta] \in I_i$ and $\mathrm{GOTO}(I_i, b) = i_j$, then $\mathrm{ACTION}[i, b] := s_j$
2. if $[B \rightarrow \alpha.] \in l_i]$ then $\mathrm{ACTION}[i, b] := r_n$ for all $b \in \mathrm{FOLLOW}(B)$ and where n is the number of $B \rightarrow \alpha$
3. if $[A' \rightarrow A.] \in l_i$ then $\mathrm{ACTION}[i, \$] := \mathrm{ACCEPT}$.

Indicies into grammar comment at 44:24

The shift actions are fairly straight forward. The follow sets can be a bit confusing.

The first example of this occurs in I_2 where $E \rightarrow T.$ is found. This corresponds to production 1.1. Consequently, that production number is also the n of r_n.

The follow sets are required for action table

Table 4: Example: The Calculator Grammar and its goto table

Item	Production	Actions
I_0	$E' \to .E$	Establish Collection with Closure ($E \to .E$)
	$E \to .T$	
	$E \to .EAT$	
	$T \to .F$	
	$T \to .TMF$	
	$F \to .(E)$	
	$F \to .o_p$	
I_1	$E' \to E.$	Apply Goto (I_0, E)
	$E' \to E.$	
	$E \to E.AT$	Apply to A
	$A \to .+$	
	$A \to .-$	
I_2	$E \to T.$	Apply Goto (I_0, T)
	$T \to T.MF$	Closure on $.M$
	$M \to .*$	
	$M \to ./$	
I_3	$T \to F.$	Apply Goto (I_0, F), no closure
I_4	$F \to (.E)$	Apply Goto (I_0, ( )
	$E \to .T$	
	$E \to .EAT$	
	$T \to .F$	
	$T \to .TMF$	
	$F \to .(E)$	
	$F \to .o_p$	
I_5	$F \to o_p.$	Apply Goto (I_0, o_p)
I_6	$E \to EA.T$	Apply Goto (I_1, A)
	$T \to .F$	closure
	$T \to .TMF$	
	$F \to .(E)$	
	$F \to .o_p$	
I_7	$A \to +.$	Apply Goto (I_1, +)
I_8	$A \to -.$	Apply Goto (I_1, $-$)
I_9	$T \to TM.F$	Apply Goto (I_2, M)
	$F \to .(E)$	Closure on $.F$
	$F \to .o_p$	
I_{10}	$M \to *.$	Apply Goto (I_2, *)
I_{11}	$M \to /.$	Apply Goto (I_2, /)

Table 5: Example: The Calculator Grammar and its goto table continued

Item	**Production**	**Actions**
I_{12}	$F \rightarrow (E.)$ $E \rightarrow E.AT$ $A \rightarrow .+$ $A \rightarrow .-$	Apply Goto (I_4, .E) Apply closure on A
I_{13}	$E \rightarrow T.$	Apply Goto (I_4, T) = I_2
I_{14}	$T \rightarrow F.$	Apply Goto (I_4, F) = I_3
I_{15}	$F \rightarrow (.E)$	Apply Goto (I_4, ( ) = I_4
I_{16}	$F \rightarrow o_p.$	Apply Goto (I_4, ( ) = I_5
I_{17}	$E \rightarrow EAT.$ $T \rightarrow T.MF$ $M \rightarrow *$ $M \rightarrow /$	GOTO (I_6, T) Closure on M
I_{18}	$T \rightarrow F.$	GOTO (I_6, F) = I_3
I_{19}	$F \rightarrow (.E)$	GOTO (I_6, ( ) = I_4
I_{20}	$F \rightarrow o_p.$	GOTO (I_6, o_p) = I_5
I_{21}	$T \rightarrow TMF.$	GOTO (I_9, F)
I_{22}	$F = (.E)$	Apply GOTO (I_9, ( ) = I_4
I_{23}	$F \rightarrow o_p.$	Apply GOTO (I_9, o_p) = I_5
I_{24}	$F \rightarrow (E).$	GOTO (I_{12},))
I_{25}	$E \rightarrow EA.T.$	GOTO (I_{12}, A) = I_6
I_{26}	$A \rightarrow +..$	GOTO (I_{12}, +) = I_7
I_{27}	$A \rightarrow +..$	GOTO (I_{12}, +) = I_8
I_{28}	$T \rightarrow TM.F$	GOTO (I_{17}, M) = I_9
I_{29}	$T \rightarrow TM.F$	GOTO (I_{17}, *) = I_{10}
I_{30}	$T \rightarrow TM.F$	GOTO (I_{17}, /) = I_{11}