How to we resolve parsing conflicts in SLR parsing tables?

- Recall our grammar and its parsing table for a list of simple assignment statements
  1. <program> → L
  2. L → <stmt>
  3. → <stmt> L
  4. <stmt> → id = <expr> ;
  5. <expr> → int
- How do we determine the reduce actions?

States	Action					Goto		
	id	=	int	;	$	L	<stmt>	<expr>
1	s2					g8	g7	
2		s3						
3			s4					g5
4	**r5**	**r5**	**r5**	**r5**	**r5**			
5				s6				
6	**r4**	**r4**	**r4**	**r4**	**r4**			
7	**s2**				**r2**	g9	g7	
8					**acc**			
9	**r3**	**r3**	**r3**	**r3**	**r3**			

Applying the parsing table to an input example

- Let us exam the situations in which parsing conflicts may occur, given input: “a = 3; b = 0;”
- The states in the parsing stack imply the current right-most derivation step in reverse
  - **Stack** **-- next token position**
  - 1 -- ^ a = 1; b = 0;
  - 1 2 -- a ^ = 1; b = 0;
  - 1 2 3 -- a = ^ 1; b = 0;
  - 1 2 id 3 = 4 int (r5) -- a = 1 ^ ; b = 0;
- then
  - 1 2 3 (g5)
  - 1 2 int 3 = 5 <expr>
  - 1 2 int 3 = 5 <expr> 6 ; (r4) -- a = 1 ; ^ b = 0;

Applying the parsing table to an input example

- 1 (g7)
- 1 7 <stmt>   -- a = 1; ^ b = 0 ;

- Then
  - **1 7 <stmt> 2 id   why shift instead of reduce?**
  - 1 7 2 3
  - 1 7 <stmt> 2 id 3 = 4 int   (r5)
  - 1 7 2 3   (g5)
  - 1 7 2 3 5
  - 1 7 <stmt> 2 id 3 = 5 <expr> 6 ;   (r4)   -- a = 1; b = 0 ; ^
  - 1 7   (g7)
- Then
  - **1 7** <stmt> **7**   <stmt>   **(r2) why reduce here?**
  - 1 7   (g9)
  - 1 7 <stmt> 9 L   (r3)
  - 1   (g8)
  - 1 8 L   (acc)
  - 1 <program>

- Imagine if we instead of doing shift to S2 as follows
  - **1 7 <stmt> 2 id**
- **We did reduce L → <stmt>, then next we had**
  - **1 (g8)**
  - 1 8 **L** **-- a = 1; ^ b = 0 ;**
  - 1 8 L The only possible action would be to accept, which would be incorrect because we still have terminals in the input.

FOLLOW sets

- From the example, we see that at any parsing step, the result of concatenating the parsing stack with the remaining input is a result of a sequence of *rightmost* derivation steps starting with <program>
- The key question to resolve the potential shift/reduce conflict is:
  - Whether it is possible for id to follow L in the parsing stack
  - That is, whether it is possible for “L id” to appear in any derivation steps starting from <program>
  - A more general question: What could follow L in any possible derivations starting from <program>
- The FOLLOW sets are computed to answer such a question.

Computing the FOLLOW sets for our example grammar

- Initialize the FOLLOW sets to empty for all non-terminals except <program>
- FOLLOW(<program>) = {$}
- From <program> → L, we get FOLLOW(L) containing $
- From L → <stmt>, we get FOLLOW(<stmt>) containing $
- From L → <stmt> L, because FIRST(L) contains id only (meaning L → id ….), we add id to FOLLOW(<stmt>)
- From <stmt> → id = <expr> ; we get FOLLOW(<expr>) containing “;”
- We iterate the above and find nothing new to add to any of the FOLLOW sets. The final results:
- FOLLOW(<program>)=FOLLOW(L) = {$}
- FOLLOW(<stmt>) = {$, id}
- FOLLOW(<expr>) = {;}

- FOLLOW(<stmt>)={$,id} means that
  - there exists a sequence of derivations <program> → …→ …<stmt>$
    - Which means it is a valid parsing situation to have <stmt> on top of the stack but all input have been exhausted.
  - There also exists a sequence of derivations <program> → … → … <stmt> id …
    - Which means it is a valid parsing situation to have <stmt> id to appear on the top of the parsing stack
  - It is an invalid parsing situation to have <stmt> and a non-id terminal to appear together in any derivations starting from <program>
  - Therefore, it is an invalid parsing situation to have <stmt> on top of stack but the remaining input is a nonempty string beginning with a non-id terminal.

- FOLLOW(L) = {$} means that
  - there exists a sequence of derivations:
    - <program> → …→ …L $
    - Which means it is a **valid** parsing situation to have L on top of the stack but all input have been exhausted
  - It is an invalid parsing situation to have L and any terminal to appear together in any derivations starting from <program>
  - Therefore it is an invalid parsing situation to have L on top of the parsing stack, with a nonempty remaining input.

Resolving parsing conflicts

- In state S7, we do r2, i.e. reduce based on "L → <stmt> ."   if and only if the next input is $
  - We do shift if next input is id
  - Everything else is an error situation
- Although other states do not present conflicts, we can refine the table for "earlier error detection" based on FOLLOW sets
  - In S4, r5 is performed when next input is ";"
  - In S6, r4 is performed when next input is $ or id
  - In S9, r3 is performed when next input is $
  - In S8, accept is performed when next input is $

The SLR parsing table based on the FOLLOW sets and the state diagram

States	Action					Goto		
	id	=	int	;	$	L	<stmt>	<expr>
1	s2					g8	g7	
2		s3						
3			s4					g5
4				**r5**				
5				s6				
6	**r4**				**r4**			
7	**s2**				**r2**	g9	g7	
8					**acc**			
9					**r3**			

Consider an incorrect input

- Let us exam the parsing actions under input: “a = 3 b = 0;” which misses “;” between two statements
  - **Stack** -- **next token position**
  - 1 -- ^ a = 1 b = 0;
  - 1 2 -- a ^ = 1 b = 0;
  - 1 2 3 -- a = ^ 1 b = 0;
  - 1 2 id 3 = 4 int -- a = 1 ^ b = 0;
- Based on the old parsing table, we do r5 and have
  - 1 2 id 3 = (g5) -- a = 1 ^ b = 0;
  - 1 2 int 3 = 5 <expr> We find error because there is no action in S5 under input id

Under the new parsing table

- Let us exam the parsing actions under input: “a = 3 b = 0;” which misses “;” between two statements
  - **Stack** **-- next token position**
  - 1 -- ^ a = 1 b = 0;
  - 1 2 -- a ^ = 1 b = 0;
  - 1 2 3 -- a = ^ 1 b = 0;
  - 1 2 id 3 = 4 int -- a = 1 ^ b = 0;
- In S4, we do not have an action for id as the next token, we detect the syntax error earlier.
- This is a subtle difference from the previous parsing table and, in today’s compiler, is not so important an improvement

The underlying concepts and algorithms leading to the computation of FOLLOW sets

- A grammar symbol is said to be *nullable* if it can eventually derive null
- To compute nullability for all symbols in a grammar:
  - Initially assume all symbols A to be **nonnullable**
  - Repeat the following until there is no change to the nullability of any A
    - For each production rule A → <right-hand side>
      - If right hand side is ε, then mark A as nullable.
      - If right hand side is $X_1X_2 \ldots X_n$ and all X_i is nullable, then mark A as nullable.

Example of nullable symbols

- <param_list> → ε   <param_list> is nullable
- <stmt_list> → <stmt>
- <stmt_list> → <stmt_list> <stmt>
- <stmt> → ε   <stmt> is nullable
- <stmt> → id = <expr> ;
- …
- In second iteration, we find <stmt_list> to be nullable because <stmt_list> → <stmt>

The FIRST sets

- Suppose α is a string of tokens and nonterminals. By expanding the nonterminals in α, various strings can be derived.
  - FIRST(α) is the set of tokens each of which can become the *leading token* in *some* string derived from α.
  - If α => ε , then we say α is nullable.

How to compute FIRST sets?

- For each nonterminal A, initialize FIRST(A) to empty.
- For each terminal a, define FIRST(a) = { a }.
- Repeat the following until there is no change to the FIRST(A) set for any A:
  - For each production rule p: A → <right-hand side>
    - If the right hand side is X_1X_2 … Xn, add FIRST(X1) to FIRST(<right-hand side>).
  - For each i such that X1 through Xi_{-1} are all nullable, add FIRST(Xi) to FIRST(<right-hand side>).
  - Add FIRST(<right-hand side>) to FIRST(A)
- Define FIRST(p) = FIRST(<right-hand side>), where p→<right-hand side> is a production rule

The FOLLOW sets

- Given a nonterminal A, FOLLOW(A) is the set of terminals each of which can immediately follow A in a certain sentential form
- How to compute the FOLLOW sets?
  - Place $ in FOLLOW(S), where S is the start nonterminal of G, $ is the end marker for the input. Initialize FOLLOW(B) as empty for all other nonterminal B.
  - Examine each production, p, in G. For each nonterminal B which appears in the right-hand side of p,
  - suppose p is in the form of $A \Rightarrow \alpha\ B\ \beta$, add FIRST($\beta$) to FOLLOW(B). In addition, if β is null or nullable, then add FOLLOW(A) to FOLLOW(B).

Revisit he second example of constructing a bottom-up parser

- The grammar:

  5. E’ → <expr>
  6. <expr> → <expr> + <term>
  7. → <expr> - <term>
  8. <term> → (<expr>)
  9. → int
  10. <expr> → <term>

- FOLLOW(E’) = {$}
- From rule 5, FOLLOW(<expr>) contains $
- From rule 6, FOLLOW(<expr>) contains + and FOLLOW(<term>) contains FOLLOW(<expr>), i.e. {$,id}
- From rule 7, FOLLOW(<expr>) is now {$,+,-} and FOLOW(<term>) is also {$,+,-}
- From rule 8, FOLLOW(<expr>) is now {$,+,-,)}
- From rule 10, FOLLOW(<term>) is now also {$,+,-,)}
- Another iteration of above will find nothing new to add.

Resolving potential conflicts

- The only place in which a potential conflict exists is state S2
- S2:E’ → <expr> . (accept?) (goto S6, S7)
  <expr> → <expr> . + <term>
  <expr> → <expr> . - <term>

But because FOLLOW(E’) = {$}, we accept if and only if we reach the end of the input

- Like in the previous example, we can refine the parsing table further by finding the valid inputs in states that have only reduce items:
- S3: <expr> → <term> . (r10 under {$,),+,-})
- S4: <term> → int . (r9 under {$,),+,-})
- S9: <expr> → <expr> + <term> . (r6 under {$,),+,-})
- S10: <expr> → <expr> - <term> . (r7 under {$,),+,-})
- S11: <term> →(<expr>) . (r8 under {$,),+,-})

The SLR parsing table based on the FOLLOW sets and the state diagram

States	Action						Goto	
	int	(	+	-	$	)	<term>	<expr>
1	s4	s5					g3	g2
2			s6	s7	acc			
3			r10	r10	r10	r10		g5
4			**r9**	**r9**	**r9**	**r9**		
5	s4	s5					g3	g8
6	**s4**	**s5**					g9	
7	**s4**	**s5**					g10	
8			s6	s7		**s11**		
9			**r6**	**r6**	**r6**	**r6**		
10			**r7**	**r7**	**r7**	**R7**		
11			**r8**	**r8**	**r8**	**r8**		