Chapter V.
Bottom-up Syntactic Analysis

a.hettab@centre-univ-mila.dz

Introduction

- **Bottom-up methods aim to construct a reversed rightmost derivation starting from the string of terminal symbols up to the grammar's start symbol.**

  Equivalently, bottom-up methods attempt to build a parse tree from the leaves upward, using a reverse depth-first traversal from left to right.

Principle of bottom-up parsing

- The **shift-reduce model** is generally used in bottom-up parsing:
  - **shift** : move the input pointer one symbol forward along the input string.
  - **Reduce :** replace a sequence of symbols (terminals and non-terminals) with a non-terminal using one of the grammar's production rules. This sequence appears to the left of the input pointer and ends at the current pointer position.

Example

- Consider the grammar **G** with the following production rules :

  S → aABe

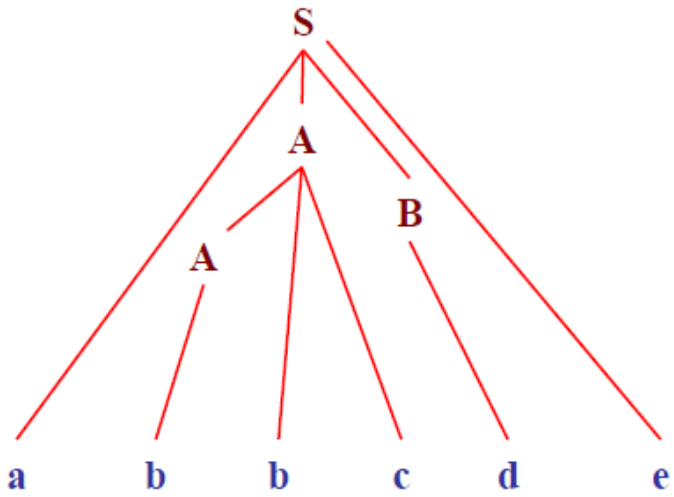
  A → Abc | b

  B → d

- We want to parse the string **abbcde** using **bottom-up parsing**.

Example (continued)

abbcde	shift
a**b**bcde	reduce by **A → b**
a**A**bcde	shift
a**Ab**cde	shift
a**Abc**de	reduce by **A → Abc**
a**A**de	shift
a**Ad**e	reduce by **B → d**
a**AB**e	shift
aABe	reduce by **S → aABe**
S	"String accepted"



Parse tree obtained using bottom-up parsing

Parsing using the LR method

- **LR(K) parsing** is an efficient bottom-up parsing technique that can be used to analyze a wide class of context-free grammars:
- **Meaning of LR(k):**
  - **L** : Left-to-right parsing, reading the input from left to right.
  - **R** : Rightmost derivation in reverse, constructing a reversed rightmost derivation.
  - **k** : Number of tokens used to make a parsing decision. K indicates how many lookahead symbols are considered when making a parsing decision.

Parsing using the LR method

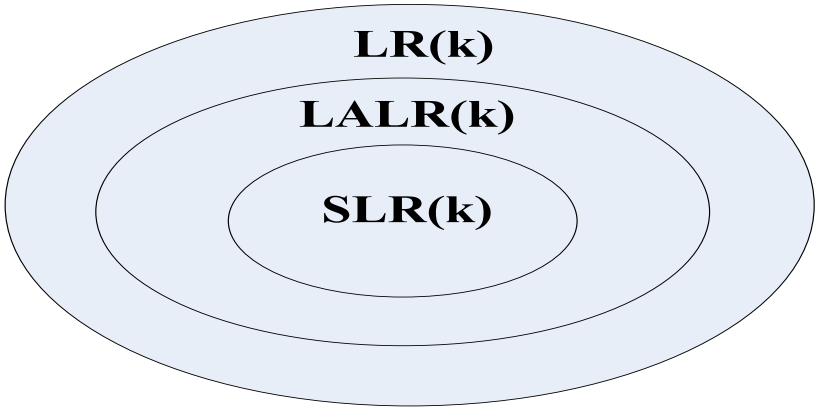
- **The LR(K) parser** is very useful for the following reasons:
  - **LR parsers** can be used in **many programming languages** and in **most parser generators**.
  - The class of grammars that can be parsed using the LR method is a strict **superset** of the class of grammars that can be parsed by predictive methods (LL grammars).
  - An LR parser can detect syntax errors as early as possible.
- **The main drawback** of the LR method is the difficulty of constructing such parsers manually; for this reason, they are generally built using parser generators such as **Yacc**, which implements an LR-based method called **LALR(1)**.

Parsing using the LR method

- **LR parsing** is performed by **constructing a DFA** (Deterministic Finite Automaton) that specifies the different states of the parsing process. This automaton is then converted into a **parsing table.**
- The main LR methods are:
  - **SLR(1):** the simplest to implement, but the least powerful.
  - **LR(1)** : very powerful but requires significant memory.
  - **LALR(1)** : a simplified version of LR(1), powerful and suitable for most programming language grammars.

Parsing using the LR method

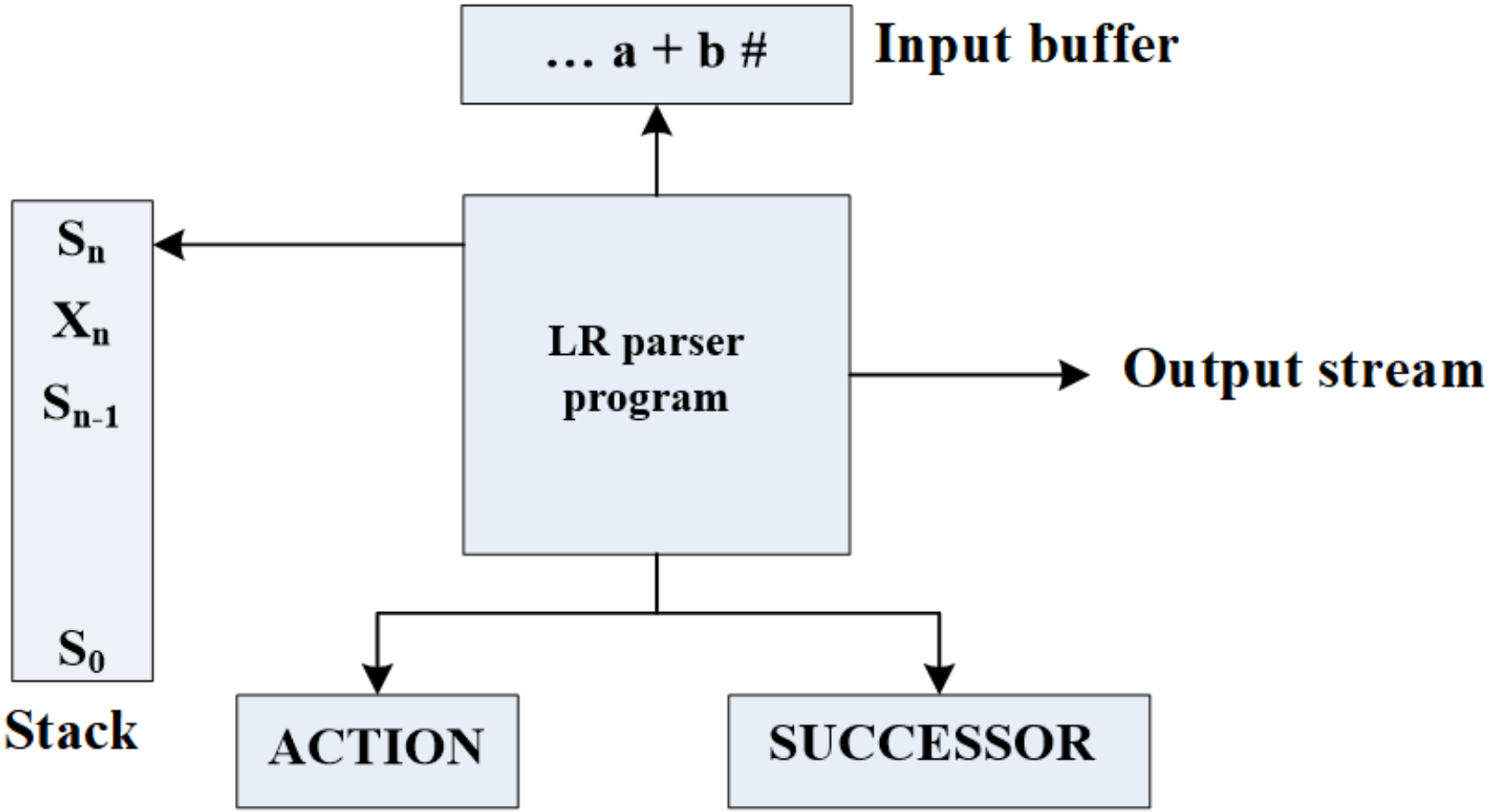
- **The class of LR(1)** grammars **is much larger** than the class of **SLR(1)** grammars and also larger than the class of **LALR(1)** grammars.
- **The class of LALR(1)** grammars is larger than the class of **SLR(1)** grammars.
- An **LALR parser** uses **tables** that have exactly the same number of states as an **SLR parser**.



LR parsing program

- The LR parsing program is the **same** for the different LR methods.
- It allows **deterministic** syntactic analysis if the parsing table associated with a grammar is **single-defined**.
- Each LR method has its own parsing table, and each table is organized into two parts: **ACTION** and **SUCCESSOR** .
- **The LR parsing** program processes the **input stream** and produces an **output stream** using the **parsing table** and a **stack**.
- **The architecture of the LR** parser is illustrated in the following diagram :

The architecture of the LR parser



The architecture of the LR parser

Remarks :

- The **ACTION** part of the parsing table corresponds to the subtable where the column indices are terminals. The **SUCCESSOR** (transition function) part of the parsing table corresponds to the subtable where the column indices are **non-terminals**.
- **The rows** of the parsing table correspond to the **states** of the DFA.
- **The stack** is used to store sequences of the form **S0 X1 S1 X2 … Xn Sn** (with the top of the stack on the right), where **Xi** is a grammar symbol and **Si** is a state.
- **The state at the top of the stack** and the **current input symbol** are sufficient to decide which parsing action to **take**.

LR analysis algorithm

- The stack initially contains the initial state of the automaton S_0 ;
- **Let S** be the state at the top of the stack, and let **a** be the symbol pointed to by **Ps** ;

```
Repeat indefinitely
    If ACTION[ S,a ]="Shift T"  then
        push(a); push(T);        Move Ps to the next symbol;
    Else
        If ACTION[ S,a ]=Reduce by A → α "  then
            pop (2*| α | symbols);
            Let U be the new state at the top of the stack;
            push(A);   push(SUCCESSOR[U,A]);
        Else
            If ACTION[ S,a ]="Accept"  then    return(Success());
            Else  return(Failure());
            EndIf ;
        EndIf ;
    EndIf ;
END;
```

Main LR methods

- The main LR methods are:
  - **SLR(1):** the simplest to implement, but the least powerful.
  - **LR(1)** : very powerful but requires significant memory.
  - **LALR(1)** : a simplified version of LR(1), powerful and suitable for most programming language grammars.

SLR (1) Parsing

- Constructing an **SLR parsing table (LR(0) or SLR(1))** from a grammar is the simplest method to implement (among the three methods **SLR(1), LALR(1),** and **LR(1)),** but it is the least powerful in terms of the number of grammars it can successfully parse.
- **SLR(1) parsing is performed as follows:**
  - Construct the finite state automaton that recognizes the viable prefixes of G (prefixes that can appear on the stack).
  - Convert this automaton into a parsing table.
  - If the parsing table is unambiguous, use the previously described algorithm to perform SLR(1) parsing.

SLR (1) Parsing: Concepts

- The construction of **SLR parsers** for a given grammar is based on the **canonical collection of LR(0) items** (which form the states of the **DFA**).
- To **construct** this **collection**, the following are defined:
  - an **augmented** grammar**,**
  - a **Closure function,**
  - a **GOTO** (**transition) function.**

SLR (1) Parsing: Item

LR(0) Items

- An **LR(0) item** of a grammar **G** is a production with a dot (.) marking a position in its right-hand side.

Examples:

- The production **A → XYZ** gives four items :

  [A → .XYZ] , [A → X.YZ] , [A → XY.Z], [A → XYZ.]

- The production **A → ε** gives the item: **[A → .]**

Remark

- An item indicates the "**amount**" of the right-hand side that **has been recognized** at a given point in the parsing process.

For example, the item **[A → X.YZ]** indicates that a string derived from **X** has just been recognized, and a string derived from **YZ** is expected next .

Closure of a set of LR(0) items

Augmented Grammar:

- The original grammar is augmented with the rule **S' → S** (where **S** is the start symbol of the grammar) to obtain the initial state of the **DFA**.

Closure(I)

- Let **I** be a set of items;
- Place each item of **I** into **Closure(I)** ;
- **If [A → α .B β]** is in **Closure(I)** and **B → γ** is a production: **Then** add the item **[B → . γ]** to **Closure(I)** if it is not already present;
- Apply this rule until no more items can be added to **Closure(I)** .

Closure of a set of LR(0) items

Example :

- Consider the grammar **G** with the following productions:

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid i$$

- The grammar is augmented by the rule $E' \rightarrow E$;

Closure($[E' \rightarrow .E]$) = $\{[E' \rightarrow .E], [E \rightarrow .E+T], [E \rightarrow .T], [T \rightarrow .T*F], [T \rightarrow .F], [F \rightarrow .(E)], [F \rightarrow .i]\}$

GOTO function for LR(0) items

- Let **I** be a set of items and **X** a symbol of the grammar.
- The **GOTO** function, **GOTO(I, X)**, is defined as follows:

GOTO(I, X) = Closure([A → α X . β] such that [A → α . X β] ∈ I).

Example:

- Using the grammar from the previous example,

if I = {[E' → E.], [E → E.+T]} , then the GoTo function is defined as: **GoTo (I, +) = Closure ([E → E+.T]) = {[E → E+.T], [T → .T*F], [T → .F], [F → .(E)], [F → .i]}**

Construction of the canonical collection of LR(0) items

Algorithm:

Construction of the canonical collection **LR(0)** items:

- **Input:** an augmented grammar $\mathbf{G_0}$
- **Output:** a canonical collection of LR(0) item sets **C** for the augmented grammar $\mathbf{G_0}$
- **Method:** the following procedure is executed to add transitions for all symbols starting from the start symbol :

Construction of the canonical collection of LR(0) items

```
Procedure Items(G0);
Begin
    C = { Closure ([S' → . S])};
    Repeat
        For each item set I of C Do
            For each grammar symbol X such that GOTO(I,X)
                                    is non-empty and not yet in C Do
                    Add GOTO(I,X) to C ;
            End for ;
        End for ;
    Until no new item set can be added to C;
END .
```

Construction of the canonical collection of LR(0) items

Example :

- Consider the augmented grammar G_0 for the grammar in the previous example:

$$E' \rightarrow E$$

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid i$$

- The canonical collection of LR(0) items is as follows :

Construction of the canonical collection of LR(0) items

Example (continued 1) :

- $I_0 = \{[E' \rightarrow .E], [E \rightarrow .E+T], [E \rightarrow .T], [T \rightarrow .T*F], [T \rightarrow .F], [F \rightarrow .(E)], [F \rightarrow .i]\}$
- $I_1 = GOTO(I_0, E) = \{[E' \rightarrow E.], [E \rightarrow E.+T]\}$
- $I_2 = GOTO(I_0, T) = \{[E \rightarrow T.], [T \rightarrow T.*F]\}$
- $I_3 = GOTO(I_0, F) = \{ [T \rightarrow F.] \}$
- $I_4 = GOTO(I_0, ( ) = \{[F \rightarrow (.E)], [E \rightarrow .E+T], [E \rightarrow .T], [T \rightarrow .T*F], [T \rightarrow .F], [F \rightarrow .(E)], [F \rightarrow .i]\}$
- $I_5 = GOTO(I_0, i) = \{[F \rightarrow i .]\}$
- $I_6 = GOTO(I_1, +) = \{[E \rightarrow E+.T], [T \rightarrow .T*F], [T \rightarrow .F], [F \rightarrow .(E)], [F \rightarrow .i]\}$

Construction of the canonical set of items LR(0)

Example (continued 2) :

- $I_7 = GOTO(I_2, *) = \{[T \rightarrow T*.F], [F \rightarrow .(E)], [F \rightarrow .\ i\ ]\}$
- $I_8 = GOTO(I_4, E) = \{[F \rightarrow (E.)], [E \rightarrow E.+T]\}$
- $GOTO(I_4, T) = I_2$
- $GOTO(I_4, F) = I_3$
- $GOTO(I_4, (\ ) = I_4$
- $GOTO(I_4, i\ ) = I_5$
- $I_9 = GOTO(I_6, T) = \{[E \rightarrow E+T.], [T \rightarrow T.*F]\}$
- $GOTO(I_6, F) = I_3$
- $GOTO(I_6, () = I_4$

Construction of the canonical set of items LR(0)

Example (continued 3) :

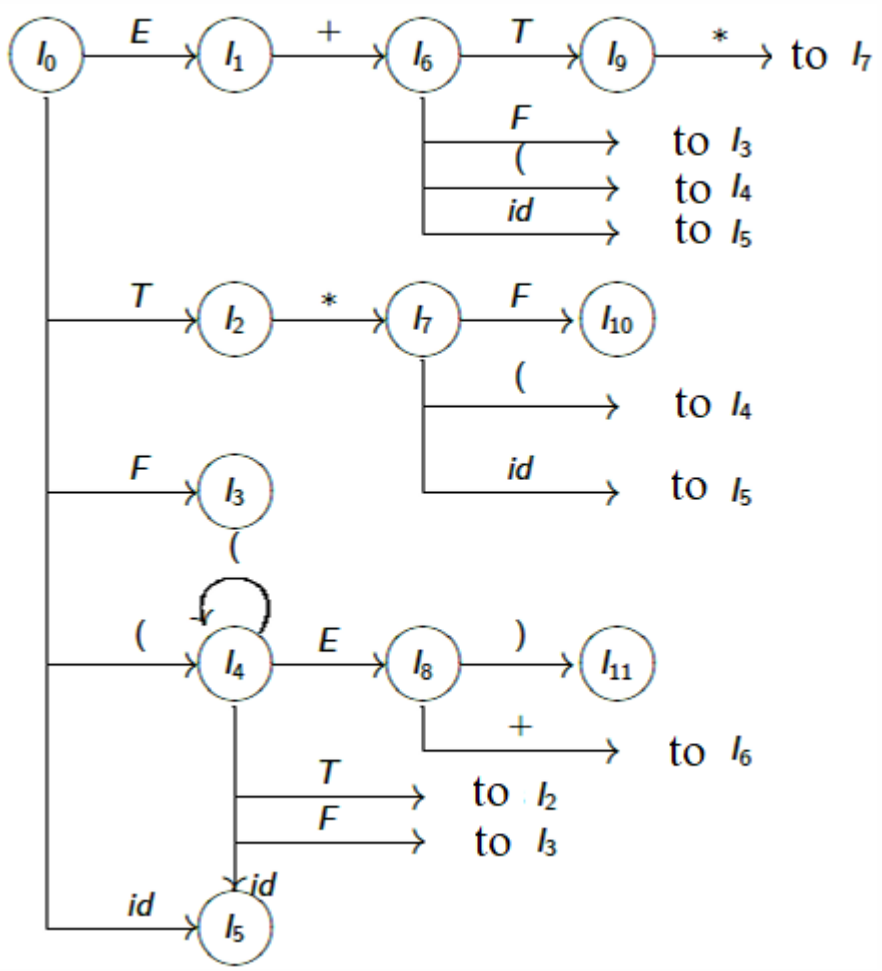
- $GOTO(I_6, i) = I_5$
- $I_{10} = GOTO(I_7, F) = \{[T \rightarrow T*F.]\}$
- $GOTO(I_7, () = I_4$
- $GOTO(I_7, i) = I_5$
- $I_{11} = GOTO(I_8,)) = \{[F \rightarrow (E).]\}$
- $GOTO(I_8, +) = I_6$
- $GOTO(I_9, *) = I_7$

Construction of the canonical set of items LR(0)

Example (continued 4) :

Here is the DFA constructed for the GOTO function.

I_0 is the initial state, and all other states are final states.



Construction of the SLR(1) parsing table Algorithm

Begin

State **i** is constructed from **Ii** .

- The **ACTION part** for state **i** is determined as follows:

If **[A → α .a β] ϵ Ii and GoTo (Ii, a)= Ij (with a ϵ T) Then**

Add Sh j (shift j) to ACTION [i, a]

End if

If [A → α .] ϵ Ii (with A≠S') Then

for all a ∈ Follow(A) do

Add R(k) to ACTION [i, a] (where k is the number of the rule A → α)

End for

End if

If [S' → S.] ϵ Ii Then Set "accept" to ACTION [i, #]

End if

Construction of the SLR(1) parsing table Algorithm (continued)

- The **SUCCESSOR** part for state **i** is determined as follows:

  If Goto (Ii, A)= Ij then

  Add j to SUCCESSOR [i, A]

  End if

- All remaining entries in the table are set to " **ERROR**"
- **The initial state** of the parser is constructed from the set of items containing **[S' → .S]**

END ;

Construction of the SLR(1) parsing table Example

- Consider the augmented grammar G_0 for the grammar from the previous example, with the numbered productions given below :

0) E' → E

1) E → E + T

2) E → T

3) T → T * F

4) T → F

5) F → (E)

6) F → i

	FOLLOW
E	#) +
T	#) + *
F	#) + *

Construction of the SLR(1) parsing table Example (continued)

	+	*	(	)	i	#	E	T	F
0			Sh4		Sh5		1	2	3
1	Sh6					Accept			
2	R (2)	Sh7		R (2)		R (2)			
3	R (4)	R (4)		R (4)		R (4)			
4			Sh4		Sh5		8	2	3
5	R (6)	R (6)		R (6)		R (6)			
6			Sh4		Sh5			9	3
7			Sh4		Sh5				10
8	Sh6			Sh11					
9	R (1)	Sh7		R (1)		R (1)			
10	R (3)	R (3)		R (3)		R (3)			
11	R (5)	R (5)		R (5)		R (5)			

The table is single defined so the previous grammar is SLR(1);

SLR(1) Parsing Example

- Parsing the string "i+i*i" :

0) E' → E

1) E → E + T

2) E → T

3) T → T * F

4) T → F

5) F → (E)

6) F → i

Stack	Remaining to be analyzed	Action
0	i + i * i #	Sh5
0 i 5	+ i * i #	R (6)
0 F 3	+ i * i #	R (4)
0 T 2	+ i * i #	R (2)
0 E 1	+ i * i #	Sh6
0 E 1 + 6	i * i #	Sh5
0 E 1 + 6 i 5	* i #	R (6)
0 E 1 + 6 F 3	* i #	R (4)
0 E 1 + 6 T 9	* i #	Sh7
0 E 1 + 6 T 9 * 7	i #	Sh5
0 E 1 + 6 T 9 * 7 i 5	#	R (6)
0 E 1 + 6 T 9 * 7 F 10	#	R (3)
0 E 1 + 6 T 9	#	R (1)
0 E 1	#	"Accept"

Conflicts in SLR(1) Parsing

- Two types of **conflict exist in SLR(1)** parsing :
  - **Shift-Reduce Conflict:**

Some states may contain an **item** of the form

[A → α .a β] and an item of the form **[A → α .]** .

  - **Reduce-Reduce Conflict:**

Some states may contain an item of the form

[A → α .] and an item of the form **[B → β .]** . With

follow (A) ∩ follow (B) ≠ Φ .

These represent situations where two different productions could be applied for reduction.

Conflicts in SLR(1) Parsing Example

- $I_0 = \{[E' \rightarrow .E], [E \rightarrow .E+T], [E \rightarrow .T], [T \rightarrow .i], [T \rightarrow .(E)], [T \rightarrow .ii[E]]\}$

- $I_1 = GOTO(I_0, E) = \{[E' \rightarrow E.], [E \rightarrow E.+T]\}$
- $I_2 = GOTO(I_0, T) = \{[E \rightarrow T.]\}$
- $I_3 = GOTO(I_0, i) = \{$ **$[T \rightarrow i.]$, $[T \rightarrow i .i [E]]$** $\}$

Shift-Reduce Conflict in LR(0) Items

- $I_4 = GOTO(I_0, ( ) = \{[F \rightarrow (.E)], [E \rightarrow .E+T], [E \rightarrow .T], [T \rightarrow .i], [T \rightarrow .(E)], [T \rightarrow .i[E]]\}$

LR(1) Parsing

- The **LR(1) parsing** method, or **canonical LR**, is the most **general** technique for constructing **LR parsing tables** for a grammar.
- The class of **LR(1)** grammars is **larger** than that of **SLR(1)** grammars.
- The **LR(1)** method helps **avoid** a number of **invalid actions** and conflicts by attaching more information to a state.

LR(1) Parsing Concepts

- The **construction of LR(1) parsers** for a given grammar is based on the **canonical collection of LR(1) items**.
- To construct this **collection**, the following are defined:
  - an **augmented** grammar**,**
  - a **Closure function,**
  - a **GOTO** (transition) **function.**

LR(1) Parsing
LR(1) Item

- The general form of an **LR(1) item** is **[A → α . β , a]** such that: **A → αβ** is a production rule of the grammar and **a** is the **lookahead (prediction) symbol** with **a ϵ T ∪ {#}**.
- The lookahead symbol **a** has no effect in an item of the form **[A → α . β , a]** with **β ≠ ε** ,
- The lookahead symbol **a** means "reduce by **A → α** " only when the next input symbol is **a** in an item of the form **[A → α ., a]** .
- The set of lookahead symbols is a subset of **Follow(A)**.

Closure of a set of LR(1) items

Augmented Grammar:

- The original grammar is augmented with the rule **S' → S** (where **S** is the start symbol of the grammar) to obtain the initial state of the **DFA**.

Closure(I)

- Let **I** be a set of items;
- Place each item of **I** into **Closure(I)** ;
- **If [A → α .B β ,a]** is in **Closure(I)** and **B → γ** is a production: **Then** add the item **[B → . γ,b]** with **b ∈ FIRST (βa)** to **Closure(I)** if it is not already present;
- Apply this rule until no more items can be added to **Closure(I)** .

Closure of a set of LR(1) items

Example :

- Consider grammar G, with the following productions :

$$S \rightarrow AA$$

$$A \rightarrow aA$$

$$A \rightarrow b$$

The grammar is augmented by the rule $S' \rightarrow S$;

Closure([S' → .S, #]) = {[S' → .S, #], [S → .AA, #], [A → .aA , a/b], [A → .b, a/b]}

GOTO function for LR(1) items

- Let **I** be a set of items and **X** a grammar symbol.
- **The transition function GoTo (I, X)** is defined as follows: **GoTo (I, X) = closure [A → α X. β , a]** such that **[A → α .X β, a] ϵ I** .

Example:

- Using the grammar from the previous example, If **I** = {[S' → .S, #], [S → .AA, #], [A → . aA , a/b], [A → .b, a/b]} , then the **GoTo function** is defined as follows:

GoTo (I, A) = Closure ([S → AA , #]) = {[S → AA , #], [A → . aA , #], [A → .b, #]}

Construction of the canonical collection of LR(1) items

Algorithm:

- **Construction of LR(1) item sets**
- **Input:** an augmented grammar **G'**
- **Output:** a canonical collection of LR(1) item sets **C** for the augmented grammar **G'**
- **Method:** execute the following procedure to add transitions for all symbols starting from the start symbol :

Construction of the canonical collection of LR(1) items

```
Procedure Items(G');
Beginning
    C = { Closure ([S' → . S,#])};
    Repeat
        For each items set I of C Do
            For each grammar symbol X such as GOTO(I,X)
                                    is non-empty and not yet in C Do
                Add GOTO(I,X) to C ;
            End for ;
        End for ;
    Until no new item set can be added to C ;
END .
```

Construction of the canonical collection of LR(1) items

Example:

- Consider the augmented grammar G' for the grammar of the previous example:

$$S' \rightarrow S$$
$$S \rightarrow AA$$
$$A \rightarrow aA$$
$$A \rightarrow b$$

- The canonical collection of LR(1) items is as follows:

Construction of the canonical collection of LR(1) items

Example (continued):

- $I_0 = \{[S' \rightarrow .S\ ,\ \#],\ [S \rightarrow .AA\ ,\ \#],\ [A \rightarrow .\ aA\ ,\ a/b],\ [A \rightarrow .b\ ,\ a/b]\}$
- $I_1 = GOTO(I_0, S) = \{[S' \rightarrow S.\ ,\ \#]\}$
- $I_2 = GOTO(I_0, A) = \{[S \rightarrow A.A\ ,\ \#],\ [A \rightarrow .\ aA\ ,\ \#],\ [A \rightarrow .b,\ \#]\}$
- $I_3 = GOTO(I_0, a) = \{[A \rightarrow a.A\ ,\ a/b],\ [A \rightarrow .\ aA\ ,\ a/b],\ [A \rightarrow .b\ ,\ a/b]\}$
- $I_4 = GOTO(I_0, b) = \{[A \rightarrow b.\ ,\ a/b]\}$

Construction of the canonical collection of LR(1) items

Example (continued):

- $I_5 = GOTO(I_2, A) = \{[S \rightarrow AA. , \#]\}$
- $I_6 = GOTO(I_2, a) = \{[A \rightarrow a.A , \#], [A \rightarrow . aA , \#], [A \rightarrow .b, \#]\}$
- $I_7 = GOTO(I_2, b) = \{[A \rightarrow b. , \#]\}$
- $I_8 = GOTO(I_3, A) = \{[A \rightarrow aA . , a/b]\}$
- $GOTO(I_3, a) = I_3$
- $GOTO(I_3, b) = I_4$
- $I_9 = GOTO(I_6, A) = \{[A \rightarrow aA . , \#]\}$
- $GOTO(I_6, a) = I_6$
- $GOTO(I_6, b) = I_7$

Construction of the LR(1) Parsing Table Algorithm

Begin

State **i** is constructed from **Ii** .

- The **ACTION part** for state **i** is determined as follows:

If [A → α .a β , b] ϵ Ii **and** GoTo (Ii, a)= Ij **(with a,b ϵ T) Then**

Add dj (shift j) to ACTION [i, a]

End if

If [A → α ., a] ϵ Ii **(with A≠S') Then**

Add r(k) to ACTION [i, a] (k is the number of the rule **A → α**)

End if

If [S' → S.,#] ϵ Ii **Then**

set "accept" to ACTION [i, #]

End if

Construction of the LR(1) Parsing Table Algorithm (continued)

- The **SUCCESSOR** part for state **i** is determined as follows:

  If Goto (Ii, A)= Ij SO

  Add j to SUCCESSOR [i, A]

  End if

- All remaining entries in the table are set to " **ERROR**"
- **The initial state** of the analyzer is constructed from the set of items containing **[S' → .S,#]**

END ;

Construction of the LR(1) Parsing Table Example (continued)

	a	b	#	S	A
0	**Sh 3**	**Sh 4**		1	2
1			**"Accepted"**		
2	**Sh 6**	**Sh 7**			5
3	**Sh 3**	**Sh 4**			8
4	**R (3)**	**R (3)**			
5			**R (1)**		
6	**Sh 6**	**Sh 7**			9
7			**R(3)**		
8	**R (2)**	**R (2)**			
9			**R(2)**		

The table is monodefined so the previous grammar is LR(1);

LR(1) Parsing Example

0) S' → S
1) S → AA
2) A → aA
3) A → b

- Analysis of the string " aaab"

Stack	Remaining to be analyzed	Action
0	aaab#	Sh 3
0 to 3	aab #	Sh 3
0a3a3	ab#	Sh 3
0a3a3a3	b#	Sh 4
0a3a3a3b4	#	"Error"

- Analysis of the string " aabb"

Stack	Remaining to be analyzed	Action
0	aabb#	Sh 3
0 a 3	abb #	Sh 3
0a3a3	bb #	Sh 4
0a3a3b4	b#	R(3)
0a3a3A8	b#	R(2)
0a3A8	b#	R(2)
0A2	b#	Sh7
0A2b7	#	R(3)
0A2A5	#	R(1)
0S1	#	"Accepted string "

Conflicts in LR(1) Parsing

- Two types of **conflict** exist in **LR(1)** parsing :
  - **Shift-Reduce conflict**

Some states may contain an **item** of the form

[A → β .a γ ,b] and an item of the form **[A → α .,a]** .

  - **Reduce–Reduce conflict**

Some states may contain an item of the form

[A → α .,a] and an item of the form **[B → β .,a]** .

These represent situations where two different reductions are possible for the common lookahead symbol **a** in the follow sets of **A** and **B.**

LALR(1) Parsing

- The **LALR(1)** parsing method (**Look Ahead LR**) is a very good **compromise** between **SLR(1)** and **LR(1)** parsing.
- A **LALR(1) parser** uses tables that have exactly the **same number of states** as an **SLR(1) parser**, but the **class** of **LALR(1)** grammars is **larger than** the **class** of **SLR(1)** grammars.
- The **LALR(1) parsing** method is **widely used in practice**; for example, the compiler generator **YACC** uses the **LALR(1)** method for the syntactic part of the compiler.

Construction of the LALR(1) Parsing Table

- **The construction of the LALR(1) parsing table** is based on the collection of **LR(1)** item sets by merging **LR(1) items** that have the same **core**. Note that an **LR(1) item** is of the form **[core, lookahead symbol].**
- The **LALR(1) parsing table** is built from the collection of **merged** item sets.
- **Remark:** There exists a more efficient method to construct **LALR(1)** tables, based on **LR(0)** items from the **SLR(1)** method. However, in this course, we will use the method based on **LR(1)** items because **it is simpler**.

Construction of the LALR(1) Parsing Table Algorithm

Begin

I. Construct the collection of LR(1) item sets $C=\{I_0, I_1, \ldots, I_n\}$.

II. For each core present among the **LR(1) item sets**, **find** all states sharing this same **core** and replace these states by their union.

III. Let $C' = \{J_0, J_1, \ldots, J_m\}$ **be** the new collection of items; The **GOTO** operation is obtained as follows:

If $J = I_0 \cup I_1 \cup \ldots \cup I_s$ **Then**

$K = \cup$ GOTO(I_i, X) , $i = 1 \ldots s$;

GOTO$(J, X) = K$;

EndIf ;

LALR(1) Parsing Example

Example:

- Consider the augmented grammar G' for the grammar of the previous example:

$$S' \rightarrow S$$
$$S \rightarrow AA$$
$$A \rightarrow aA$$
$$A \rightarrow b$$

- The canonical collection of LR(1) items is as follows:

LALR(1) Parsing
Example (continued)

- $I_0 = \{[S' \rightarrow .S\ ,\ \#],\ [S \rightarrow .AA\ ,\ \#],\ [A \rightarrow .\ aA\ ,\ a/b],\ [A \rightarrow .b\ ,\ a/b]\}$
- $I_1 = GOTO(I_0, S) = \{[S' \rightarrow S.\ ,\ \#]\}$
- $I_2 = GOTO(I_0, A) = \{[S \rightarrow A.A\ ,\ \#],\ [A \rightarrow .\ aA\ ,\ \#],\ [A \rightarrow .b,\ \#]\}$
- $I_3 = GOTO(I_0, a) = \{[A \rightarrow a.A\ ,\ a/b],\ [A \rightarrow .\ aA\ ,\ a/b],\ [A \rightarrow .b\ ,\ a/b]\}$
- $I_4 = GOTO(I_0, b) = \{[A \rightarrow b.\ ,\ a/b]\}$

LALR(1) Parsing Example (continued)

- $I_5 = GOTO(I_2, A) = \{[S \rightarrow AA. , \#]\}$
- $I_6 = GOTO(I_2, a) = \{[A \rightarrow a.A , \#], [A \rightarrow . aA , \#], [A \rightarrow .b, \#]\}$
- $I_7 = GOTO(I_2, b) = \{[A \rightarrow b. , \#]\}$
- $I_8 = GOTO(I_3, A) = \{[A \rightarrow aA . , a/b]\}$
- $GOTO(I_3, a) = I_3$
- $GOTO(I_3, b) = I_4$
- $I_9 = GOTO(I_6, A) = \{[A \rightarrow aA . , \#]\}$
- $GOTO(I_6, a) = I_6$
- $GOTO(I_6, b) = I_7$

LALR(1) Parsing
Example (continued)

Construction of the canonical collection of LR(1) items by merging items with the same core:

States **I_3** and **I_6** have the same **core**:

- $I_3 = \{[A \rightarrow aA\ ,\ \mathbf{a/b}\ ],\ [A \rightarrow .\ aA\ ,\ \mathbf{a/b}\ ],\ [A \rightarrow .b\ ,\ \mathbf{a/b}\ ]\}$
- $I_6 = \{[A \rightarrow aA\ ,\ \#\ ],\ [A \rightarrow .\ aA\ ,\ \#\ ],\ [A \rightarrow .b\ ,\ \#\ ]\}$
- They are **merged** into:
- $I_{36} = \{[A \rightarrow aA\ ,\ \mathbf{a/b/\#}\ ],\ [A \rightarrow .\ aA\ ,\ \mathbf{a/b/\#}\ ],\ [A \rightarrow .b\ ,\ \mathbf{a/b/\#}\ ]\}$

LA LR analysis (1)
Example (continued)

States $\mathbf{I_4}$ and $\mathbf{I_7}$ have the same **core :**

- $I_4 = \{[A \rightarrow b. , \mathbf{a/b}]\}$
- $I_7 = \{[A \rightarrow b. , \mathbf{\#}]\}$
- They are **merged** into :
- $I_{47} = \{[A \rightarrow b. , \mathbf{a/b/\#}]\}$

States I_8 and I_9 have the same **core :**

- $I_8 = \{[A \rightarrow aA . , \mathbf{a/b}]\}$
- $I_9 = \{[A \rightarrow aA . , \mathbf{\#}]\}$
- They are **merged** into :
- $I_{89} = \{[A \rightarrow aA . , \mathbf{a/b/\#}]\}$

LA LR analysis (1)
Example (continued)

The new canonical collection of items is:

- $I_0 = \{[S' \rightarrow .S\ ,\ \#],\ [S \rightarrow .AA\ ,\ \#],\ [A \rightarrow .\ aA\ ,\ a/b],\ [A \rightarrow .b\ ,\ a/b]\}$
- $I_1 = GOTO(I_0, S) = \{[S' \rightarrow S.\ ,\ \#]\}$
- $I_2 = GOTO(I_0, A) = \{[S \rightarrow AA\ ,\ \#],\ [A \rightarrow .\ aA\ ,\ \#],\ [A \rightarrow .b,\ \#]\}$
- $I_{36} = GOTO(I_0, a) = \{[A \rightarrow aA\ ,\ a/b/\#],\ [A \rightarrow .\ aA\ ,\ a/b/\#],\ [A \rightarrow .b\ ,\ a/b/\#]\}$
- $I_{47} = GOTO(I_0, b) = \{[A \rightarrow b.\ ,\ a/b/\#]\}$

LA LR analysis (1)
Example (continued)

- $I_5 = GOTO(I_2, A) = \{[S \rightarrow AA. , \#]\}$
- $GOTO(I_2, a) = I_{36}$
- $GOTO(I_2, b) = I_{47}$
- $I_{89} = GOTO(I_{36}, A) = \{[A \rightarrow aA . , a/b/\#]\}$
- $GOTO(I_{36}, a) = I_{36}$
- $GOTO(I_{36}, b) = I_{47}$

Construction of the LALR(1) Parsing Table Example (continued)

	a	b	#	S	A
0	Sh 36	Sh 47		1	2
1			Accept		
2	Sh 36	Sh 47			5
36	Sh 36	Sh 47			89
47	R(3)	R(3)	R(3)		
5			R(1)		
89	R(2)	R(2)	R(2)		

The table is monodefined so the previous grammar is LALR(1);

LALR(1) Parsing Example

0) S' → S
1) S → AA
2) A → aA
3) A → b

- Parsing the string " aaab"

Stack	Remaining to be analyzed	Action
0	aaab#	Sh 36
0a36	aab #	Sh36
0a36a36	ab#	Sh36
0a36a36a36	b#	Sh47
0a36a36a36b47	#	R(3)
0a36a36a36A89	#	R(2)
0a36a36A89	#	R(2)
0a36A89	#	R(2)
0A2	#	"Error"

- Parsing the string " aabb"

Stack	Remaining to be analyzed	Action
0	aabb#	Sh 36
0a36	abb #	Sh36
0a36a36	bb #	Sh47
0a36a36b47	b#	R(3)
0a36a36A89	b#	R(2)
0a36A89	b#	R(2)
0A2	b#	Sh47
0A2b47	#	R(3)
0A2A5	#	R(1)
0S1	#	"Accepted string"

Conflicts in LALR(1) Parsing

- During the construction of **LALR(1)** tables from **LR(1)** tables, there is a risk of conflicts :
  - **Shift-Reduce Conflict :**

Merging two states **cannot** create new **shift-reduce conflicts** that were not already present in the **LR(1)** states, because the **merged** states share the **same core**.

  - **Reduce-Reduce Conflict :**

This is the **only type of conflict** that can occur during the **merging** of **LR(1)** states. For this reason, there exist **LR(1)** grammars that **are not LALR(1)**.

Yacc, a Syntax Parser Generator

- The **Yacc** program (**Y**et **A**nother **C**ompiler **C**ompiler) is a **syntax parser generator**.
- It takes as input a source file consisting essentially of the productions of a context-free grammar **G**, and outputs a **C program** which, once compiled, produces a syntax parser for the language **L(G)**.
- Yacc is based on the **LALR bottom-up parsing method**.

Yacc, a Syntax Parser Generator

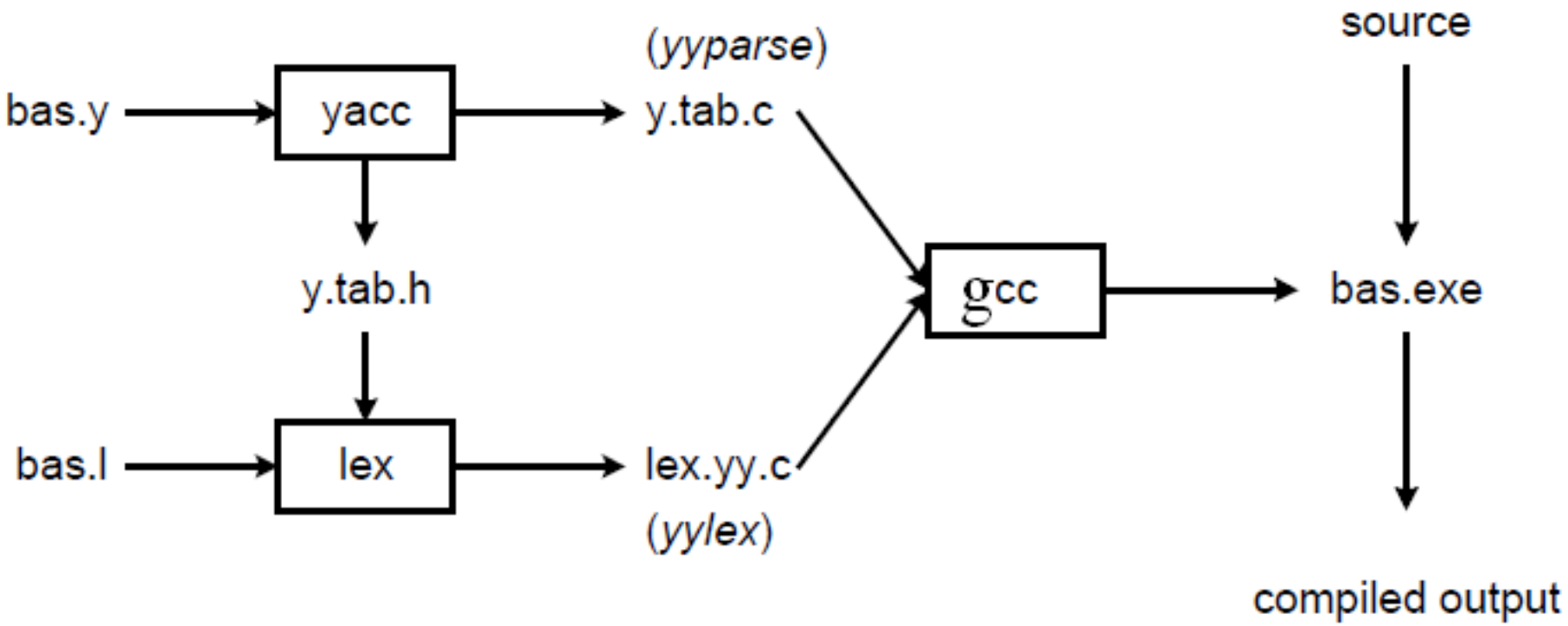
- In the grammar description provided to **Yacc**, **semantic actions** can be associated with the productions; these are **C source code instructions** that **Yacc** inserts at the appropriate places in the generated parser. The parser can then execute actions or produce information derived from the source text.

Yacc and lex

- A syntax parser requires a **lexical analyzer** to work, which provides the **input stream** as **successive lexical units**. By default, **Yacc** assumes that the available lexical analyzer was created using **Lex**.
- For this reason, the program generated by **Yacc** includes calls to the **yylex** function (from **Lex**) at the points where the acquisition of a **lexical unit** is needed.

Yacc and lex

- The diagram below illustrates the steps for constructing a complete syntax parser using **Yacc** and **Lex**.



Yacc and lex

- The commands to create the executable file **bas.exe** (the syntax parser) are:
- **yacc -d bas.y** # create the files **y.tab.h , y.tab.c**
- **lex bas.l** # create the file **lex.yy.c**
- **gcc lex.yy.c y.tab.c -o bas.exe - lfl** # create the executable file **bas.exe**

Structure of a Source File for Yacc

- A source file for **Yacc** must have a name ending with "**.y**". It consists of three sections, separated by the symbol "**%%**":

```
%{
  Declarations for the C compiler
%}
  Declarations for yacc
%%
  Rules (productions + semantic actions)
%%
  Additional C functions
```

Structure of a Source File for Yacc

- The **"declarations for the C compiler"** section is simply copied into the generated file.
- The **"declarations for Yacc"** section is used to specify the declarations of **lexical units.**
- **Example:**

```
%{
  char name[256];
  int b = 50;
%}
  %token number identifier
```

Structure of a source file for yacc

- The **"rules (productions + semantic actions)"** section is used to define the productions of the chosen language grammar.
- Each production is written in the following form:

```
non_terminal :
     body_1 {semantic_action_1}
   | …
   | body_n   { action_semantique_n }
  ;
```

Example:

```
term:
      term '*' factor { printf ("*"); }
      | factor
      ;
```

Structure of a source file for yacc

- The "**additional C functions**" section must include the **main()** function, which generally calls the **yyparse()** function to execute the parser.
- **Example:**

```
int main(void)
{
        if ( yyparse () == 0)
            printf (" Text is correct\n ");
}
```

Structure of a source file for yacc

- **Note:**
- In a **Yacc** program, you must write a function that is called in case of an **error** with the following specification:

void yyerror (char *message)

Example :

```
void yyerror(char *message)
{
    printf(" <<< %s\n", message);
}
```

Lex and Yacc: Example

```
%{
#include   "syntax.tab.h"
external char name[]; /* string shared with the parser */
%}
figure  [0-9]
letter   [A- Za -z]
%%
[" "\ t\n ]                          { }
{digit}+                             { yylval = atoi ( yytext ); return number; }
{letter}({letter}|{number})*         { strcpy (name, yytext ); return identifier; }
.                                    { return yytext [0]; }
%%
int  yywrap ( void )
{ return 1; }                                  /* Lex Specification File */
```

Lex and Yacc: Example

```
%{
char name[256];       /* string shared with the lex file */
%}
% token    number    identifier
%%
expression :   expression '+' term    { printf (" +"); }
| term ;
term: term '*' factor                 { printf ("*"); }
| factor ;
factor : number                       { printf ("%d", yylval ); }
| identifier                          { printf ("%s", name ); }
| '(' expression ')' ;

/* Yacc Specification File */
```

```
%%
void yyerror (char *s)
{
printf ("<<< \ n%s ", s);
}
hand()
{
if ( yyparse () == 0)
printf (" Correct expression
    \n");
}
```

YACC variables

- **YACC** uses **variables** to refer to the **non-terminals** and **terminals** in a grammar.
- **$$** refers to the attribute associated with the left-hand side of a production rule.
- **$i** refers to the attribute associated with the **i-th** symbol on the right-hand side of a production rule.
- The default semantic rule is { **$$=$1 ;**}.
- An empty production is represented by an empty alternative.

Conflicts in YACC

Yacc performs a default action in case of a conflict.

I) Shift/Reduce Conflicts :

- **Yacc** chooses to **shift**.

II) Reduce/Reduce Conflicts :

- **Yacc** uses the first rule in the list. It also displays a warning message whenever a conflict exists.

Conflict resolution in YACC

- It is possible to specify an **ambiguous** grammar in a **YACC** file. However, the **associativity** and **precedence** of operators must be specified in the **declaration section** to **resolve ambiguities**.
- Associativity is specified using the keywords **Left** (for left associativity) or **Right** (for right associativity).
- **Operators** specified this way will have **increasing** precedence according to the order in which they are written.
- Example:
  - **% left ‘+’ '-'**
  - **% left ‘*’ '/'**