THEORY OF COMPUTATION AND COMPILERS

Unit - II

CONTEXT FREE GRAMMARS AND PARSING

- Introduction
- Context-Free Grammars - Derivation, Parse trees, Ambiguity
- Types of Parsers
- LL(K) grammars and LL(1) parsing
- Bottom-up Parsing - handle pruning
- **LR Grammar Parsing** 
- LALR parsing
- Parsing ambiguous grammars
- Error Recovery in Parsing
- YACC programming specification

Dr. R. Madana Mohana
Professor, Artificial Intelligence & Data Science | I/c-Head, Artificial Intelligence & Machine Learning
CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY
Hyderabad - 500 075, Telangana, INDIA
www.cbit.ac.in

Unit-II: Syntax Analysis (or) Parser
Introduction to LR Parsing

Outline:

- Introduction
- Why LR Parsers
- Items and the LR(0) Automaton

LR Parsing

Introduction:

The **LR(k)** parser will parse the input string by applying rightmost derivation in reverse. The **LR(k)** parsers are **bottom-up parsers** that are capable of handling **context-free languages** very efficiently.

where

- The **L** is for left-to-right scanning of the input.
- The **R** for constructing a rightmost derivation in reverse,
- The **k** for the number of input symbols of lookahead that are used in making parsing decisions.

The cases **k = 0** or **k = 1** are of practical interest, and we shall only consider **LR** parsers with **k ≤ 1** here. When (**k**) is omitted, **k** is assumed to be **1**.

Why LR Parsers

- **LR** parsers are table-driven, much like the nonrecursive **LL** parsers.
- A grammar for which we can construct a parsing table using one of the **LR** parsing methods (**SLR**- Simple LR / **CLR**- Canonical LR / **LALR**- Look-Ahead LR) is said to be an **LR** grammar.
- For a grammar to be **LR** it is sufficient that a left-to-right shift-reduce parser be able to recognize handles of right-sentential forms when they appear on top of the stack.

Why LR Parsers

LR parsing Advantages:

LR parsing *is attractive for a variety of reasons:*

- **LR parsers** can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written.
- The **LR-parsing** method is the most general nonbacktracking shift-reduce parsing method known, yet it can be implemented as efficiently as other, more primitive shift-reduce methods.

Why LR Parsers

LR parsing Advantages:

- An **LR parser** can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.
- **LR parsers** can describe more languages than **LL grammars**.

Why LR Parsers

LR parsing Disadvantages:

- The main drawback of **LR parsers** is that they consume too much time to construct by hand for a typical programming language grammar.
- But, this disadvantage can be eliminated using **LR parser generators** such as **YACC** (Yet-Another Compiler-Compiler).

Types of LR Parsers

There are three types of **LR parsers** are shown below:

1. **Simple LR parsers (SLR):** This parser is easy to implement. Hence the name **simple LR**. But, sometimes it may fail to produce the parsing table for certain types of grammars.
2. **Canonical LR parsers (CLR):** This parser is more powerful and works on very large class of grammars. But, it is very expensive to implement.

Types of LR Parsers

There are three types of **LR parsers** are shown below:

3. **Look-Ahead LR parsers (LALR):** This parser is intermediate in power between **SLR** and **CLR** parsers. This parser can be implemented for most of the grammars.

Items and the LR(0) Automaton

Definition: Item or LR(0) item

An **LR(0) item** or **item** of a grammar **G** is a production of **G** with a **dot**(**.**) on the **Right-Hand Side (RHS)** of a production.

Example:

Consider the production **A → BCD**. The various items that can be obtained using this production are:

A → .BCD

A → B.CD

A → BC.D

A → BCD.

Items and the LR(0) Automaton

Note:

For the production of the form **A → ϵ**. The item is:

A → .

Using these items called **canonical LR(0)** collection, we can construct a **DFA** (**Deterministic Finite Automata**) which is used to make **parsing decisions** such as **shifting** and **reducing**.

Items and the LR(0) Automaton

The **canonical LR(0)** collection of items for a grammar can be obtained using:

- An **augmented grammar**
- **CLOSURE** function
- **GOTO** function
- **ITEMS** function

Items and the LR(0) Automaton

Augmented grammar:

Let **G = (V, T, S, P)** be grammar. An **augmented grammar G′** for the grammar **G** is defined as:

G′ = (V′, T, S′, P′)

where

V′= V U S′

P′: P U {S′ → S}

S′ is the **start symbol** for the **augmented grammar G′**.

Items and the LR(0) Automaton

Augmented grammar:

The main purpose of the new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input. That is, acceptance occurs whenever the parser is about to reduce using the production **S′ → S**.

Items and the LR(0) Automaton

Augmented grammar:

Example:

Obtain the augmented grammar for the following grammar:

E → E + T | T

T → T * F | F

F → (E)| id

Items and the LR(0) Automaton

Augmented grammar:

Example: ***Solution***

By adding the production **E′ → E** to the given grammar **G**, we get the augmented grammar **G′** as shown below:

E′ → E

E → E + T | T

T → T * F | F

F → (E)| id

Items and the LR(0) Automaton

Closure of Item Sets:

If **I** is a set of *items* for a grammar **G**, then **CLOSURE**(**I**) is constructed using the following two rules:

Rule 1: Initially, add every *item* in **I** to **CLOSURE**(**I**).

Rule 2: If **A → α.Bβ** is in **CLOSURE**(**I**) and **B → γ** is a production and **B → .γ** is not there in **CLOSURE**(**I**), then add **B → .γ** to **CLOSURE**(**I**).

Rule 3: Apply **rule 2** repeatedly until no more new items are added to **CLOSURE**(**I**).

Items and the LR(0) Automaton

Closure of Item Sets:

Example:

Consider the following augmented grammar:

E′ → E

E → E + T | T

T → T * F | F

F → (E)| id

a) If **I = {E′ → .E}** then find **CLOSURE(I)**

b) If **I = {F → (.E)}** then find **CLOSURE(I)**

Items and the LR(0) Automaton

Closure of Item Sets:

Example: ***Solution***

The **CLOSURE(I)** can be obtained using the given productions as shown below:

a) Since **I = {E′ → .E}** add **E′ → .E** to **CLOSURE**(**I**) and compute **CLOSURE**(**I**) as shown below:

Items and the LR(0) Automaton

Closure of Item Sets:

Example: ***Solution***

CLOSURE(E′ → .E) =

{

E′ → .E placed initially. After **dot(.)**, **E** is present. So, add **E**-productions beginning with **dot** as shown below:

E → .E + T

E → .T

Items and the LR(0) Automaton

Closure of Item Sets:

Example: ***Solution***

CLOSURE(E′ → .E) =

{

E → .T After **dot**(**.**), **T** is present. So, add **T**-productions beginning with **dot** as shown below:

T → .T * F

T → .F

Items and the LR(0) Automaton

Closure of Item Sets:

Example: ***Solution***

CLOSURE(E′ → .E) =

{

T → .F After **dot**(**.**), **F** is present. So, add **F**-productions beginning with **dot** as shown below:

F → .(E)

F → .id After **dot**(**.**) no variable (or non-terminal) is present. So, stop at this point.

}

Items and the LR(0) Automaton

Closure of Item Sets:

Example: ***Solution***

b) Since **I = {F → (.E)}** add **F → (.E)** to **CLOSURE**(**I**) as shown below:

CLOSURE(**F → (.E)**) **=**

{

F → (.E) After **dot**(**.**), **E** is present. So, add **E**-productions beginning with **dot** as shown below:

E → .E + T

E → .T

Items and the LR(0) Automaton

Closure of Item Sets:

Example: ***Solution***

b) Since **I = {F → (.E)}** add **F → (.E)** to **CLOSURE**(**I**) as shown below:

CLOSURE(**F → (.E)**) **=**

{

E → .T After **dot**(**.**), **T** is present. So, add **T**-productions beginning with **dot** as shown below:

T → .T * F

T → .F

Items and the LR(0) Automaton

Closure of Item Sets:

Example: ***Solution***

b) Since **I = {F → (.E)}** add **F → (.E)** to **CLOSURE**(**I**) as shown below:

CLOSURE(**F → (.E)**) **=**

{

T → .F After **dot**(**.**), **F** is present. So, add **F**-productions beginning with **dot** as shown below:

F → .(E)

F → .id

Items and the LR(0) Automaton

Closure of Item Sets:

Example: ***Solution***

b) Since **I = {F → (.E)}** add **F → (.E)** to **CLOSURE**(**I**) as shown below:

CLOSURE(**F → (.E)**) **=**

{

F → .id After **dot**(**.**) no variable (or non-terminal) is present. So, stop at this point.

}

Items and the LR(0) Automaton

Algorithm to find CLOSURE(I):

Function **CLOSURE**(**I**)

{

J = I

repeat

for each item **A → α.Bβ** is in **J** and for each production **B → γ** in **G**

if **B → .γ** is not in **J** then

add **B → .γ** to **J**

return **J**

}

Items and the LR(0) Automaton

GOTO function:

The **GOTO** (**I, X**) is a function which accepts set of items **I** as the *first argument* and **X** which is a *grammar symbol* (it can be either a terminal or a non-terminal) as the *second argument*. It is formally defined as shown below:

If **A → α.Xβ** in **I**

GOTO (I, X) = CLOSURE (A → αX.β)

Items and the LR(0) Automaton

GOTO function:

The **GOTO** function is used to define the *transition* from *one set of items* to *other set of items*. Using these transitions we can easily obtain the parsing table consisting of **ACTION** and **GOTO**. **I** is a *set of items* for a grammar **G**.

Items and the LR(0) Automaton

Algorithm to find GOTO (I, X):

Function **GOTO** (**I, X**)

{

if **(A → α.Xβ** is in **I)**

return **CLOSURE** (**A → αX.β**)

}

Items and the LR(0) Automaton

GOTO function:

Example:

Consider the following augmented grammar:

E′ → E

E → E + T | T

T → T * F | F

F → (E)| id

If **I = {[E → T.], {[E → E.+T] }** then compute **GOTO(I, +)**

Items and the LR(0) Automaton

GOTO function:

Example: ***Solution***

Given that **I = {[E → T.], {[E → E.+T] }**.

E → T. is already completed item because **dot** (**.**) is placed at the end.

In **E → E.+T** Immediately after **dot** (**.**) there is a symbol '**+**'. So, to compute **GOTO** (**I**, **+**):

- Let us consider the item: **E → E.+T**
- Shift **dot** (**.**) to next position: **E → E+.T**
- Now, it is necessary to compute: **CLOSURE** (**E → E+.T**)

Items and the LR(0) Automaton

GOTO function:

Example: ***Solution***

GOTO (**I**, **+**) = **CLOSURE** (**E → E+.T**) **=**

{

E → E+.T Placed initially. After **dot** (**.**) **T** is present. So, add **T**-productions beginning with **dot** (**.**) as shown below:

T → .T * F

T → .F

Items and the LR(0) Automaton

GOTO function:

Example: ***Solution***

GOTO (**I**, **+**) = **CLOSURE** (**E → E+.T**) **=**

{

T → .F After **dot** (**.**) **F** is present. So, add **F**-productions beginning with **dot** (**.**) as shown below:

F → .(E)

F → .id After **dot** (**.**) no non-terminal is present. So, stop at this point.

}

Items and the LR(0) Automaton

Canonical collection of sets of LR(0) items:

The canonical collection of sets of **LR(0)** *items* are computed using the following algorithm:

Function **ITEMS** (**G′**)

{

C = CLOSURE (**S′ → .S**)

repeat

for each set of items **I** in **C**

Items and the LR(0) Automaton

Canonical collection of sets of LR(0) items:

Function **ITEMS** (**G′**)

{

C = CLOSURE (**S′ → .S**)

repeat

for each set of items **I** in **C**

for each grammar symbol **X**

if (**GOTO (I, X)** is not empty and not in **C**

add **GOTO (I, X)** to **C**

until no more sets of items are added to **C**;

return **C**

}

Items and the LR(0) Automaton

Canonical collection of sets of LR(0) items:

Example:

Consider the following augmented grammar:

E → E + T | T

T → T * F | F

F → (E)| id

Obtain **LR(0)** items (or) compute canonical collection of sets of **LR(0)** items.

Items and the LR(0) Automaton

Canonical collection of sets of LR(0) items:

Example: ***Solution***

Given grammar:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Augmented grammar:

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Items and the LR(0) Automaton

Canonical collection of sets of LR(0) items:

Example: The various items obtained along with **GOTO**'s are obtained by constructing the initial item with **CLOSURE** ***(E′ → .E)*** as shown below:

I_0:

E′ → .E

E → .E + T

E → .T

T → .T * F

T → .F

F → .(E)

F → .id

Items and the LR(0) Automaton

Canonical collection of sets of LR(0) items:

I_1: GOTO (I_0, E)

E′ → E.

E → E. + T

I_2: GOTO (I_0, T)

E → T.

T → T. * F

I_3: GOTO (I_0, F)

T → F.

I_4: GOTO (I_0, ()

F → (.E)

E → .E + T

E → .T

T → .T * F

T → .F

F → .(E)

F → .id

Items and the LR(0) Automaton

Canonical collection of sets of LR(0) items:

I_5: GOTO (I_0, id)

F → id.

I_6: GOTO (I_1, +)

E → E +. T

T → .T * F

T → .F

F → .(E)

F → .id

I_7: GOTO (I_2, *)

T → T *.F

F → .(E)

F → .id

I_8: GOTO (I_4, E)

F → (E.)

E → E. + T

Items and the LR(0) Automaton

Canonical collection of sets of LR(0) items:

I_9: GOTO (I_4, T) = I_2

E → T.

T → T. * F

I_{10}: GOTO (I_4, F) = I_3

T → F.

I_{11}: GOTO (I_4, () = I_4

F → (.E)

E → .E + T

E → .T

T → .T * F

T → .F

F → .(E)

F → .id

Items and the LR(0) Automaton

Canonical collection of sets of LR(0) items:

I_{12}: GOTO (I_4, id) = I_5

F → id.

I_{13}: GOTO (I_6, T)

E → E + T.

T → T. * F

I_{14}: GOTO (I_6, F) = I_3

T → F.

I_{15}: GOTO (I_6, id) = I_5

F → id.

I_{16}: GOTO (I_6, () = I_4

F → (.E)

E → .E + T

E → .T

T → .T * F

T → .F

F → .(E)

F → .id

Items and the LR(0) Automaton

Canonical collection of sets of LR(0) items:

I_{17}: GOTO (I_7, F)
T → T * F.

I_{18}: GOTO (I_7, () = I_4
F → (.E)
E → .E + T
E → .T
T → .T * F
T → .F
F → .(E)
F → .id

I_{19}: GOTO (I_7, id) = I_5
F → id.

I_{20}: GOTO (I_8,))
F → (E).

I_{21}: GOTO (I_8, +) = I_6
E → E +. T
T → .T * F
T → .F
F → .(E)
F → .id

Items and the LR(0) Automaton

Canonical collection of sets of LR(0) items:

I_{22}: GOTO (I_{13}, *) = I_7

T → T *.F

F → .(E)

F → .id

Items and the LR(0) Automaton

Construction of DFA for Canonical collection of sets of LR(0) items:

Items and the LR(0) Automaton

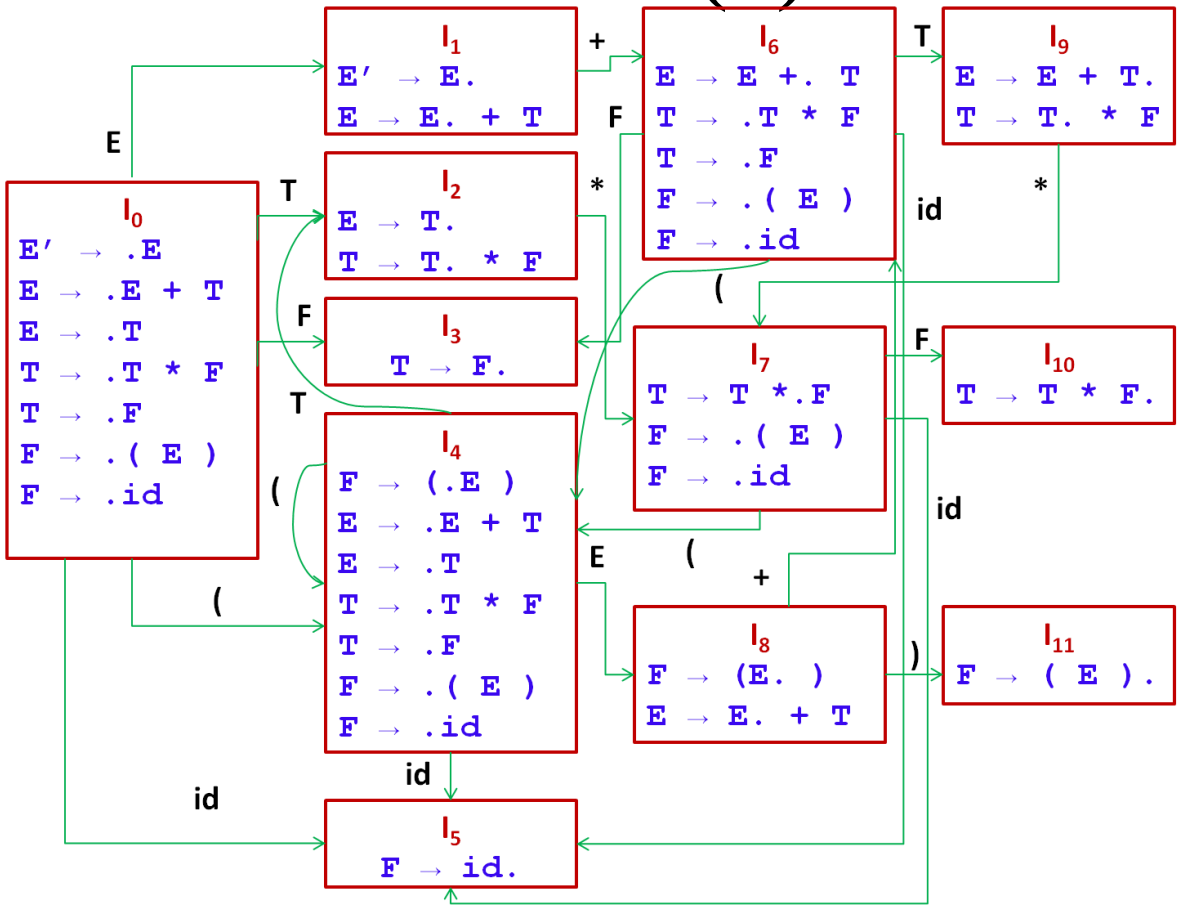

Fig: DFA for LR(0) items

Items and the LR(0) Automaton

Note: All the items called **LR(0)** items in the above Automaton (**DFA**) are divided into two classes:

1. **Kernel items:** The initial item, **S′ → .S** and all the items that are not starting with **dot** (**.**) are called **kernel items**.
2. **Non-kernel items:** All the items with **dot** (**.**) at the beginning except the item **S′ → .S** are called **non-kernel items**.

Items and the LR(0) Automaton

Canonical collection of sets of LR(0) items:

Practice problem:

Construct the LR(0) items for the following grammar:

S → L = R | R
L → * R | id
R → L

Check whether the above grammar is LR(0) grammar or not

[JNTUH: March/April – 2021]

Summary...

Bottom-Up Parsing:

- Introduction
- Why LR Parsers
- Items and the LR(0) Automaton

Reading: Aho2, Section 4.6.1 & 4.6.2

Next Lecture: The LR-Parsing Algorithm