Course Outcomes

CO 1: Acquire fundamental understanding of the core concepts in automata theory and Theory of Computation

CO 2: Design and develop lexical analyzers, parsers and code generators

CO 3: Design Grammars and Automata (recognizers) for different language classes and become knowledgeable about restricted models of Computation (Regular, Context Free) and their relative powers.

CO 4: Acquire fundamental understanding of the structure of a Compiler and Apply concepts automata theory and Theory of Computation to design Compilers

CO 5: Design computations models for problems in Automata theory and adaptation of such model in the field of compilers

Institution Vision

To produce top-quality engineers who are groomed for attaining excellence in their profession and competitive enough to help in the growth of nation and global society.

Institution Mission

M1: To offer affordable high-quality graduate program in engineering with value education and make the students socially responsible.

M2: To support and enhance the institutional environment to attain research excellence in both faculty and students and to inspire them to push the boundaries of knowledge base.

M3: To identify the common areas of interest amongst the individuals for the effective industry-institute partnership in a sustainable way by systematically working together.

M4: To promote the entrepreneurial attitude and inculcate innovative ideas among the engineering professionals.

Department Vision

To be a center of excellence in Information Science & Engineering education, research and training to meet the growing needs of the industry and society.

Department Mission

M1: To impart theoretical and practical knowledge through the concepts and technologies in Information Science and Engineering

M2: To foster research, collaboration and higher education with premier institutions and industries.

M3: Promote innovation and entrepreneurship to fulfill the needs of the society and industry

Program Educational Objectives

PEO1: Analyse, design and implement solutions to the real-world problems in the field of Information Science and Engineering with multidisciplinary setup

PEO2: Keep abreast with the technology, innovation and pursue higher education with high standards of social and professional ethics

PEO3: Develop professional and entrepreneurship skills to work effectively as an individual and in a team to meet the ever-changing goals of the organization

Program Outcomes

PO1: **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.

PO2: **Problem Analysis:** Identify, formulate, review research literature, and analyse complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural science and engineering sciences.

PO3: **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal and environmental considerations.

PO4: **Conduct investigations of complex problems:** Use research based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: **Modern tool usage:** Create, select and apply appropriate techniques, resources and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations

PO6: **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice

PO7: **Environment sustainability:** Understand the impact of the professional engineering solutions in the societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and team work: Function effectively as an individual and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10: **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions

PO11: **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to ones own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: **Lifelong learning:** Recognize the need for, and have the preparation and ability to engage in independent and lifelong learning in the broader context of technological change.

Program Specific Outcomes

PSO1: Design, implement and maintain the information systems that fulfill the current needs of the industry and society

PSO2: Apply computational theory, storage and networking concepts to solve the day to day problems of the world

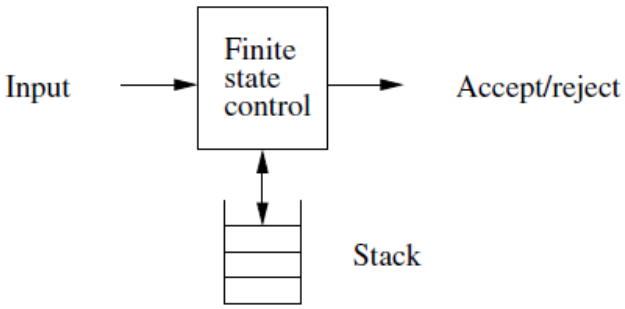
PUSH DOWN AUTOMATA

The context free languages have a type of automaton that defines them. This automaton, called a pushdown automaton, is an extension of the nondeterministic finite automaton with ε-transitions, which is one of the ways to define the regular languages.

The pushdown automaton is essentially an ε-NFA with the addition of a stack. The stack can be read, pushed, and popped only at the top, just like the "stack" data structure.

Definition of pushdown Automata:

The pushdown automaton is in essence a nondeterministic finite automaton with ε-transitions permitted and one additional capability: a stack on which it can store a string of "stack symbols". The presence of a stack means that, unlike the finite automaton, the pushdown automaton can "remember" an in finite amount of information. However, unlike a general purpose computer, which also has the ability to remember arbitrarily large amounts of information, the pushdown automaton can only access the information on its stack in a last in first out way. As a result, there are languages that could be recognized by some computer program, but are not recognizable by any pushdown automaton. In fact, push down automata recognize all and only the context free languages.



As Fig. indicates, a pushdown automaton consists of three components: 1) an input tape, 2) a control unit and 3) a stack structure. The input tape consists of a linear configuration of cells each of which contains a character from an alphabet. This tape can be moved one cell at a time to the left. The stack is also a sequential structure that has a first element and grows in either direction from the other end. Contrary to the tape head associated with the input tape, the head positioned over the current stack element can read and write special stack characters from that position. The current stack element is always the top element of the stack, hence the name ``stack". The control unit contains both tape heads and finds itself at any moment in a particular state.

Our formal notation for a pushdown automaton "PDA" involves seven components. We write the specification of a PDA P as follows:

$$P = \{Q, \sum, \Gamma, \delta, q_0, Z_0, F\}$$

The components have the following meanings:

Q: A finite set of states, like the states of a finite automaton.

∑: A finite set of input symbols, also analogous to the corresponding component of a finite automaton.

Γ: A finite stack alphabet. This component, which has no finite-automaton analog, is the set of symbols that we are allowed to push onto the stack.

δ: The transition function. As for a finite automaton, δ governs the behaviour of the automaton. Formally, δ takes as argument a triple δ (q, a, X), where:

1. q is a state in Q.
2. a is either an input symbol in ∑ or a = ε the empty string, which is assumed not to be an input symbol.
3. X is a stack symbol, that is, a member of Γ.

The output of δ is a finite set of pairs (p, γ), where p is the new state and γ is the string of stack symbols that replaces X at the top of the stack.

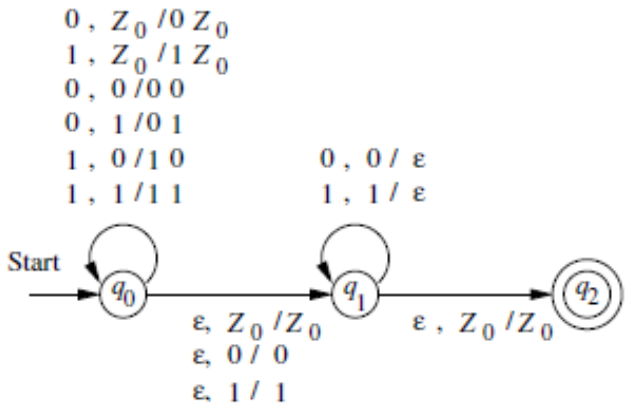
For instance, if γ = ε then the stack is popped, if γ = X then the stack is unchanged, and if γ =YZ, then X is replaced by Z and Y is pushed onto the stack.

Q_0: The start state, The PDA is in this state before making any transitions.

Z_0: The start symbol, Initially, the PDA's stack consists of one instance of this symbol, and nothing else.

F: The set of accepting states, or final states.

Example: Let us design a PDA P to accept the language L_{wwr}



$$P = (\{q_0, q_1, q_2\}, \{0,1\}, \{0,1,Z_0\}, \delta, q_0, Z_0, \{q_2\})$$

where δ is defined by the following rules:

1. $\delta(q_0, 0, Z_0) = \{(q_0, 0Z_0)\}$ and $\delta(q_0, 1, Z_0) = \{(q_0, 1Z_0)\}$. One of these rules applies initially, when we are in state q_0 and we see the start symbol Z_0 at the top of the stack. We read the first input, and push it onto the stack, leaving Z_0 below to mark the bottom.

2. $\delta(q_0, 0, 0) = \{(q_0, 00)\}$, $\delta(q_0, 0, 1) = \{(q_0, 01)\}$, $\delta(q_0, 1, 0) = \{(q_0, 10)\}$, and $\delta(q_0, 1, 1) = \{(q_0, 11)\}$. These four, similar rules allow us to stay in state q_0 and read inputs, pushing each onto the top of the stack and leaving the previous top stack symbol alone.

3. $\delta(q_0, \epsilon, Z_0) = \{(q_1, Z_0)\}$, $\delta(q_0, \epsilon, 0) = \{(q_1, 0)\}$, and $\delta(q_0, \epsilon, 1) = \{(q_1, 1)\}$. These three rules allow P to go from state q_0 to state q_1 spontaneously (on ϵ input), leaving intact whatever symbol is at the top of the stack.

4. $\delta(q_1, 0, 0) = \{(q_1, \epsilon)\}$, and $\delta(q_1, 1, 1) = \{(q_1, \epsilon)\}$. Now, in state q_1 we can match input symbols against the top symbols on the stack, and pop when the symbols match.

5. $\delta(q_1, \epsilon, Z_0) = \{(q_2, Z_0)\}$. Finally, if we expose the bottom-of-stack marker Z_0 and we are in state q_1, then we have found an input of the form ww^R. We go to state q_2 and accept.

Instantaneous Descriptions of a PDA

The Instantaneous description is called as an informal notation, and explains how a Push down automata (PDA) computes the given input string and makes a decision that the given string is accepted or rejected.

- The PDA involves both state and content of the stack.
- Stack is often one of the important parts of PDA at any time.
- So, we make a convenient notation for describing the successive configurations of PDA for string processing.
- The factors of PDA notation by triple (q, w, γ) were
    - q is the current state.
    - w is the remaining input alphabet.
    - γ is the current contents of the PDA stack.

Generally, the leftmost symbol indicates the top of the stack γ and the bottom at the right end. This type of triple notation is called an instantaneous description or ID of the pushdown automata.

A move from one instantaneous description to another is denoted by the symbol '⊢'

Therefore,

$(q0, aw, z0) \vdash (q1, w, yz0)$

is possible if and only if

$\delta(q0, a, z0) \in (q1, yz0)$.

Example

Consider an example as given below −

Show the IDs or moves for input string w = "aabb" of PDA where,

M = ({q0, q1, q2}, {a, b}, {a, b, Z0}, δ, q0, Z0, {q2}),

Where δ is defined as follows −

$\delta(q0, a, Z0) = \{(q0, aZ0)\}$ Rule (1)

$\delta(q0, a, a) = \{(q0, aa)\}$ Rule (2)

$\delta(q0, b, a) = \{(q1, \varepsilon)\}$ Rule (3)

$\delta(q1, b, a) = \{(q1, \varepsilon)\}$ Rule (4)

$\delta(q1, \varepsilon, Z0) = \{(q2, \varepsilon)\}$ Rule (5)

$\delta(q0, \varepsilon, Z0) = \{(q2, \varepsilon)\}$ Rule (6)

And we need to find out whether string w is accepted by PDA or not.

Solution

The Instantaneous Description for the string w = "aabb". It is explained below −

(q0, aabb, Z0)

|- (q0, abb, aZ0) based on Rule (1)

|- (q0, bb, aaZ0) based on Rule (2)

|- (q1, b, aZ0 based on Rule (3)

|- (q1, ε, Z0) based on Rule (3)

|- (q2, ε, ε) based on Rule (5)

Therefore, PDA reaches a configuration of (q2, ε, ε) i.e. PDA stack is empty and it has reached a final state. So the string 'w' is accepted.

The language of a PDA

A language can be accepted by Pushdown automata using two approaches:

1. Acceptance by Final State: The PDA is said to accept its input by the final state if it enters any final state in zero or more moves after reading the entire input.

Let P =(Q, ∑, Γ, δ, q0, Z, F) be a PDA. The language acceptable by the final state can be defined as:

$$L(PDA) = \{w \mid (q0, w, Z) \vdash^* (p, \varepsilon, \varepsilon), q \in F\}$$

2. Acceptance by Empty Stack: On reading the input string from the initial configuration for some PDA, the stack of PDA gets empty.

Let P =(Q, ∑, Γ, δ, q0, Z, F) be a PDA. The language acceptable by empty stack can be defined as:

$$N(PDA) = \{w \mid (q0, w, Z) \vdash^* (p, \varepsilon, \varepsilon), q \in Q\}$$

Equivalence of Acceptance by Final State and Empty Stack

- If L = N(P1) for some PDA P1, then there is a PDA P2 such that L = L(P2). That means the language accepted by empty stack PDA will also be accepted by final state PDA.
- If there is a language L = L (P1) for some PDA P1 then there is a PDA P2 such that L = N(P2). That means language accepted by final state PDA is also acceptable by empty stack PDA.

Example:

Construct a PDA that accepts the language L over {0, 1} by empty stack which accepts all the string

of 0's and 1's in which a number of 0's are twice of number of 1's.

Solution:

There are two parts for designing this PDA:

- If 1 comes before any 0's
- If 0 comes before any 1's.

We are going to design the first part i.e. 1 comes before 0's. The logic is that read single 1 and push two 1's onto the stack. Thereafter on reading two 0's, POP two 1's from the stack. The δ can be

1. $\delta(q0, 1, Z) = (q0, 11, Z)$ Here Z represents that stack is empty

2. $\delta(q0, 0, 1) = (q0, \varepsilon)$

Now, consider the second part i.e. if 0 comes before 1's. The logic is that read first 0, push it onto the stack and change state from q0 to q1. [Note that state q1 indicates that first 0 is read and still second 0 has yet to read].

Being in q1, if 1 is encountered then POP 0. Being in q1, if 0 is read then simply read that second 0 and move ahead. The δ will be:

1. $\delta(q0, 0, Z) = (q1, 0Z)$

2. $\delta(q1, 0, 0) = (q1, 0)$

3. $\delta(q1, 0, Z) = (q0, \varepsilon)$(indicate that one 0 and one 1 is already read, so simply read the second 0)

4. $\delta(q1, 1, 0) = (q1, \varepsilon)$

Now, summarize the complete PDA for given L is:

1. $\delta(q0, 1, Z) = (q0, 11Z)$

2. $\delta(q0, 0, 1) = (q1, \varepsilon)$

3. $\delta(q0, 0, Z) = (q1, 0Z)$

4. $\delta(q1, 0, 0) = (q1, 0)$

5. $\delta(q1, 0, Z) = (q0, \varepsilon)$

6. $\delta(q0, \varepsilon, Z) = (q0, \varepsilon)$ ACCEPT state

3. From Empty Stack to Final State

The classes of languages that are L(P) for some PDA P is the same as the class of languages that are N(P) for some PDA P. This class is also exactly the context free languages. The first construction shows how to take a PDA P_N that accepts a language L by empty stack and construct a PDA P_F that accepts L by final state.

Theorem: If $L = N(P_N)$ for some PDA $P_N = P = \{Q, \sum, \Gamma, \delta , q_0, Z_0, F\}$ then there is a PDA P_F such that $L = L(P_F)$.

PROOF: The idea behind the proof is in Fig. We use a new symbol X_0, which must not be a symbol of Γ; X_0 is both the start symbol of P_F and a marker on the bottom of the stack that lets us know when P_N has reached an empty stack. That is, if P_F sees X_0 on top of its stack, then it knows that P_N would empty its stack on the same input.

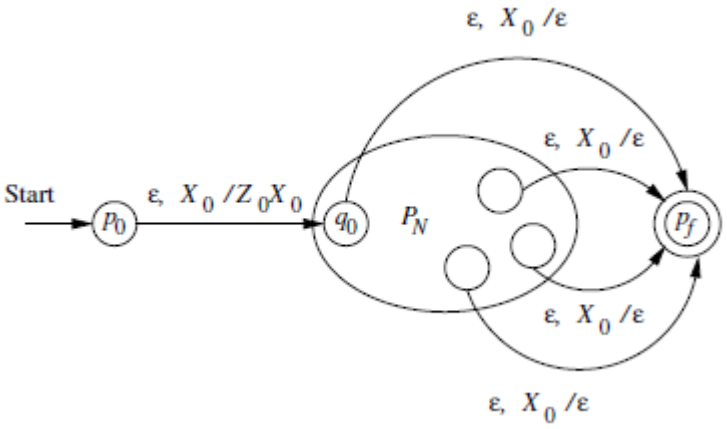


Figure: P_F simulates P_N and accepts if P_N empties its stack

Theorem: If L = N(Pn) for some PDA $P_n = (Q_n, \Sigma, \Gamma_n, \delta_n, q_0, Z_0, F_n)$, then there is a PDA $P_f = (Q_f, \Sigma, \Gamma_f, \delta_f, p_0, X_0, F_f)$ such that $L = L(P_f)$.

Proof: The idea behind the proof is in Figure. We use a new symbol X_0, which must not be a symbol of Γ_n; X_0 is both the start symbol of P_f and a marker on the bottom of the stack that lets us know when P_n has reached an empty stack. That is, if P_f sees X_0 on top of the stack, then it knows that P_n would empty its stack on the same input.

We also need a new start state, p_0, whose sole function is to push Z_0, the start state of P_n, onto the top of the stack and enter state q_0, the start state of P_n. Then, P_f simulates P_n, until the stack of P_n is empty, which P_f detects because it sees X_0 on the top of the stack. Finally, we need another new state, p_f, which is the accepting state of P_f; this PDA transfers to state p_f whenever it discover that P_n would have emptied its stack.

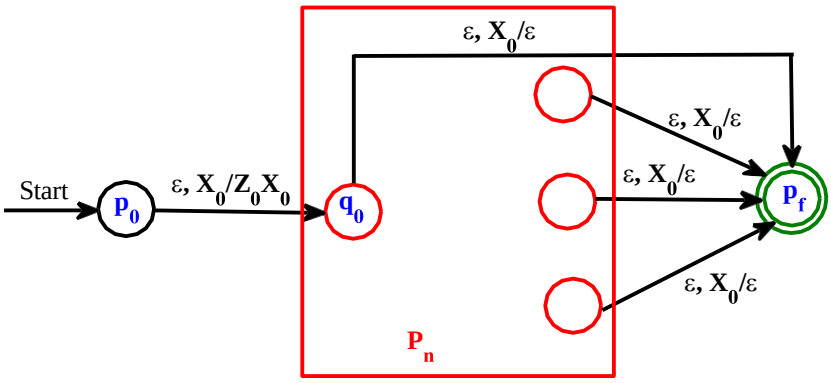


Figure: P_f simulates P_n and accepts if P_n empties its stack.

The specification of Pf is as follows:

$Q_f = Q_n \cup \{p_0, p_f\}$.

$\Gamma_f = \Gamma_n \cup \{X_0\}$.

$F_f = \{p_f\}$.

δ_f is defined by

1. $\delta_f (p_0, \epsilon, X_0) = \{(q_0, Z_0X_0)\}$. In its start state, P_f makes a spontaneous transition to the start state of P_n, pushing its start symbol Z_0 onto the stack.
2. For all state $q \in Q_n$, inputs $a \in \Sigma_n$ or $a = \epsilon$, and stack symbol $Y \in \Gamma_n$, $\delta_f (q, a, Y)$ contains all the pairs in $\delta_n(q, a, Y)$.
3. In addition to rule (2), $\delta_f (q, \epsilon, X_0)$ contains (p_f , ϵ) for every state $q \in Q_n$.

We must show that w is in L(Pf) if and only if w is in N(Pn).

(If) We are given that $(q0, w, Z0) \vdash^*_{Pn} (q, \epsilon, \epsilon)$ for some state q. Insert X_0 at the bottom of the stack and conclude $(q_0, w, Z_0X_0) \vdash^*_{Pn} (q, \epsilon, X_0)$. Since by rule (2) above, P_f has all the moves of P_n, we may also conclude that $(q_0, w, Z_0X_0) \vdash^*_{Pf} (q, \epsilon, X0)$. If we put this sequence of moves with the initial and final moves from rules (1) and (3) above, we get:

$$(p_0, w, X_0) \vdash^*_{pf} (q_0, w, Z_0X_0) \vdash^*_{Pf} (q, \epsilon, X_0) \vdash^*_{pf} (p_f , \epsilon, \epsilon) \ldots (A)$$

Thus, P_f accepts w by final state.

Only-if The converse requires only that we observe the additional transitions of rules (1) and (3) gives us very limited ways to accept w by final state. We must use rule (3) at the last step, and we can only use that rule if the stack of Pf contains only X0. No X0's ever appear on the stack except at the bottommost position. Further, rule (1) is only used at the first step, and it must be used at the first step.

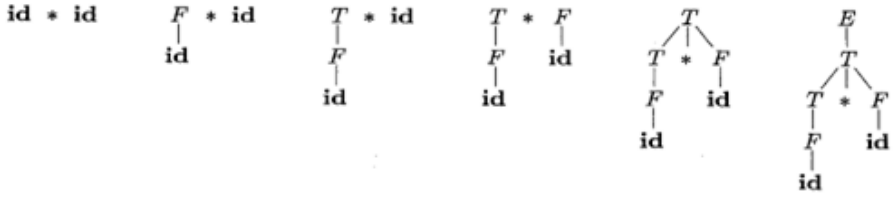
Thus, any computation of Pf that accept w must look like sequence (A). Moreover, the middle of the computation - all but the first and last steps - must also be a computation of Pn with X0 below the stack. The reason is that, except for the first and last steps, Pf cannot use any transition that is not also a transition of Pn, and X0 cannot be exposed or the computation would end at the next step. We conclude that $(q0, w, Z0) \vdash^* Pn (q, \epsilon, \epsilon)$. That is $w \in N(Pn)$.

Bottom-Up Parsers

Bottom-up parse for the input string id*id for the grammar

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

- **A bottom-up parse** corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).
- It is convenient to describe parsing as the process of building parse trees, although a front end may in fact carry out a translation directly without building an explicit tree.
- We can think of bottom-up parsing as the process of "reducing" a string w to the start symbol of the grammar.
- At each *reduction* step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.
- The key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds.



A bottom-up parse for **id * id**

Handle Pruning

Bottom-up parsing during a left-to-right scan of the input constructs a rightmost derivation in reverse. Informally, a ***"handle"*** is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \rightarrow \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \rightarrow F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \rightarrow \mathbf{id}$
$T * F$	$T * F$	$E \rightarrow T * F$

Handles during a parse of $\mathbf{id}_1 * \mathbf{id}_2$

Shift reduce parsing

- Shift-reduce parsing are the form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.
- The handle always appears at the top of the stack just before it is identified as the handle.
- $ is used to mark the bottom of the stack and also the right end of the input.

While the primary operations are shift and reduce, there are actually four possible actions a shift-reduce parser can make: **(1) shift, (2) reduce, (3) accept, and (4) error.**

1. *Shift.* Shift the next input symbol onto the top of the stack.

2. *Reduce.* The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.

3. *Accept.* Announce successful completion of parsing.

4. *Error.* Discover a syntax error and call an error recovery routine.

Figure shows steps through the actions a shift-reduce parser might take in parsing the input string idl *id2 according to the expression grammar

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	shift
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	reduce by $F \rightarrow \mathbf{id}$
$ F	$* \mathbf{id}_2$ $	reduce by $T \rightarrow F$
$ T	$* \mathbf{id}_2$ $	shift
$ $T *$	$\mathbf{id}_2$ $	shift
$ $T * \mathbf{id}_2$	$	reduce by $F \rightarrow \mathbf{id}$
$ $T * F$	$	reduce by $T \rightarrow T * F$
$ T	$	reduce by $E \rightarrow T$
$ E	$	accept

Configurations of a shift-reduce parser on input $\mathbf{id}_1 * \mathbf{id}_2$

Conflicts during Shift-Reduce Parsing.

1. shift/reduce conflict (to shift or to reduce):

There are context-free grammars for which shift-reduce parsing cannot be used. Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack contents and the next input symbol, cannot decide whether to shift or to reduce.

Example:

Example : An ambiguous grammar can never be LR. For example, consider the dangling-else grammar

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \textbf{other}
\end{array}
$$

If we have a shift-reduce parser in configuration

STACK	INPUT
$\cdots$ **if** *expr* **then** *stmt*	**else** $\cdots$ \$

- We cannot tell whether if *expr* **then** *stmt* is the handle, no matter what appears below it on the stack. Here there is a shift / reduce conflict.
- Depending on what follows the **else** on the input, it might be correct to reduce if *expr* **then** *stmt* to *stmt*, or it might be correct to shift **else** and then to look for another *stmt* to complete the alternative if *expr* **then** *stmt* **else** *stmt.*

2. *Reduce/reduce conflict:*

There are context-free grammars for which shift-reduce parsing cannot be used. Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack contents and the next input symbol, cannot decide which of several reductions to make a *reduce.*

Example:

(1)	*stmt*	$\rightarrow$	**id** (*parameter_list*)
(2)	*stmt*	$\rightarrow$	*expr* := *expr*
(3)	*parameter_list*	$\rightarrow$	*parameter_list* , *parameter*
(4)	*parameter_list*	$\rightarrow$	*parameter*
(5)	*parameter*	$\rightarrow$	**id**
(6)	*expr*	$\rightarrow$	**id** (*expr_list*)
(7)	*expr*	$\rightarrow$	**id**
(8)	*expr_list*	$\rightarrow$	*expr_list* , *expr*
(9)	*expr_list*	$\rightarrow$	*expr*

Productions involving procedure calls and array references

STACK	INPUT
$\cdots$ **id** (**id**	, **id**) $\cdots$

- It is evident that the **id** on top of the stack must be reduced, but by which production? The correct choice is production (5) if **p** is a procedure, but production (7) if p is an array.
- The stack does not tell which; information in the symbol table obtained from the declaration of **p** must be used.

Introduction to LR Parsing: Simple LR

Introduction to LR Parsing: Simple LR

> The most prevalent type of bottom-up parser today is based on a concept called LR(k) parsing; the "L" is for left-to-right scanning of the input, the "R" for constructing a rightmost derivation in reverse, and 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 $k \leq 1$ here. When (k) is omitted, k is assumed to be 1.

Why LR Parsers?

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. Non-LR context-free grammars exist, but these can generally be avoided for typical programming-language constructs.
- 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 (see the bibliographic notes).
- 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.
- The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive or LL methods. For a grammar to be LR(k), we must be able to recognize the occurrence of the right side of a production in a right-sentential form, with k input symbols of lookahead. This requirement is far less stringent than that for LL(k) grammars where we must be able to recognize the use of a production seeing only the first k symbols of what its right side derives. Thus, it should not be surprising that LR grammars can describe more languages than LL grammars.

The principal drawback of the LR method is that it is too much work to construct an LR parser by hand for a typical programming-language grammar. A specialized tool, an LR parser generator, is needed. Fortunately, many such generators are available, and one of the most commonly used ones, YACC.

Items and the LR (0) Automaton

An LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in a parse. States represent sets of "items." An *LR(0) item* (*item* for short) of a grammar G is a production of G with a dot at some position of the body. Thus, production $A \to XYZ$ yields the four items

$$A \to \cdot XYZ$$
$$A \to X \cdot YZ$$
$$A \to XY \cdot Z$$
$$A \to XYZ \cdot$$

The production $A \to \epsilon$ generates only one item, $A \to \cdot$.

Intuitively, an item indicates how much of a production we have seen at a given point in the parsing process. For example, the item $A \to \cdot XYZ$ indicates that we hope to see a string derivable from XYZ next on the input. Item $A \to X \cdot YZ$ indicates that we have just seen on the input a string derivable from X and that we hope next to see a string derivable from YZ. Item $A \to XYZ \cdot$ indicates that we have seen the body XYZ and that it may be time to reduce XYZ to A.

One collection of sets of LR(0) items, called the *canonical* LR(0) collection, provides the basis for constructing a deterministic finite automaton that is used to make parsing decisions. Such an automaton is called an *LR(0) automaton.*[3] In particular, each state of the LR(0) automaton represents a set of items in the canonical LR(0) collection. The automaton for the expression grammar (4.1), shown in Fig. 4.31, will serve as the running example for discussing the canonical LR(0) collection for a grammar.

To construct the canonical LR(0) collection for a grammar, we define an augmented grammar and two functions, CLOSURE and GOTO. If G is a grammar with start symbol S, then G', the *augmented grammar* for G, is G with a new start symbol S' and production $S' \to S$. The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input. That is, acceptance occurs when and only when the parser is about to reduce by $S' \to S$.

Closure of Item Sets

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

1. Initially, add every item in I to CLOSURE(I).
2. If $A \to \alpha \cdot B\beta$ is in CLOSURE(I) and $B \to \gamma$ is a production, then add the item $B \to \cdot\gamma$ to CLOSURE(I), if it is not already there. Apply this rule until no more new items can be added to CLOSURE(I).

A convenient way to implement the function *closure* is to keep a boolean array *added*, indexed by the nonterminals of G, such that *added*[B] is set to **true** if and when we add the item $B \to \cdot\gamma$ for each B-production $B \to \gamma$.

```
SetOfItems CLOSURE(I) {
    J = I;
    repeat
        for ( each item A → α·Bβ in J )
            for ( each production B → γ of G )
                if ( B → ·γ is not in J )
                    add B → ·γ to J;
    until no more items are added to J on one round;
    return J;
}
```

Computation of CLOSURE

We divide all the sets of items of interest into two classes:

1. *Kernel items*: the initial item, $S' \to \cdot S$, and all items whose dots are not at the left end.

2. *Nonkernel items*: all items with their dots at the left end, except for $S' \to \cdot S$.

The Function GOT0

The second useful function is GOTO(I, X) where I is a set of items and X is a grammar symbol. GOTO(I, X) is defined to be the closure of the set of all items $[A \to \alpha X \cdot \beta]$ such that $[A \to \alpha \cdot X\beta]$ is in I. Intuitively, the GOTO function is used to define the transitions in the LR(0) automaton for a grammar. The states of the automaton correspond to sets of items, and GOTO(I, X) specifies the transition from the state for I under input X.

If I is the set of two items $\{[E' \to E\cdot],\ [E \to E\cdot + T]\}$, then GOTO$(I, +)$ contains the items

$$
\begin{aligned}
&E \to E + \cdot T\\
&T \to \cdot T * F\\
&T \to \cdot F\\
&F \to \cdot(E)\\
&F \to \cdot\mathbf{id}
\end{aligned}
$$

We computed GOTO$(I, +)$ by examining I for items with $+$ immediately to the right of the dot. $E' \to E\cdot$ is not such an item, but $E \to E\cdot + T$ is. We moved the dot over the $+$ to get $E \to E + \cdot T$ and then took the closure of this singleton set. □

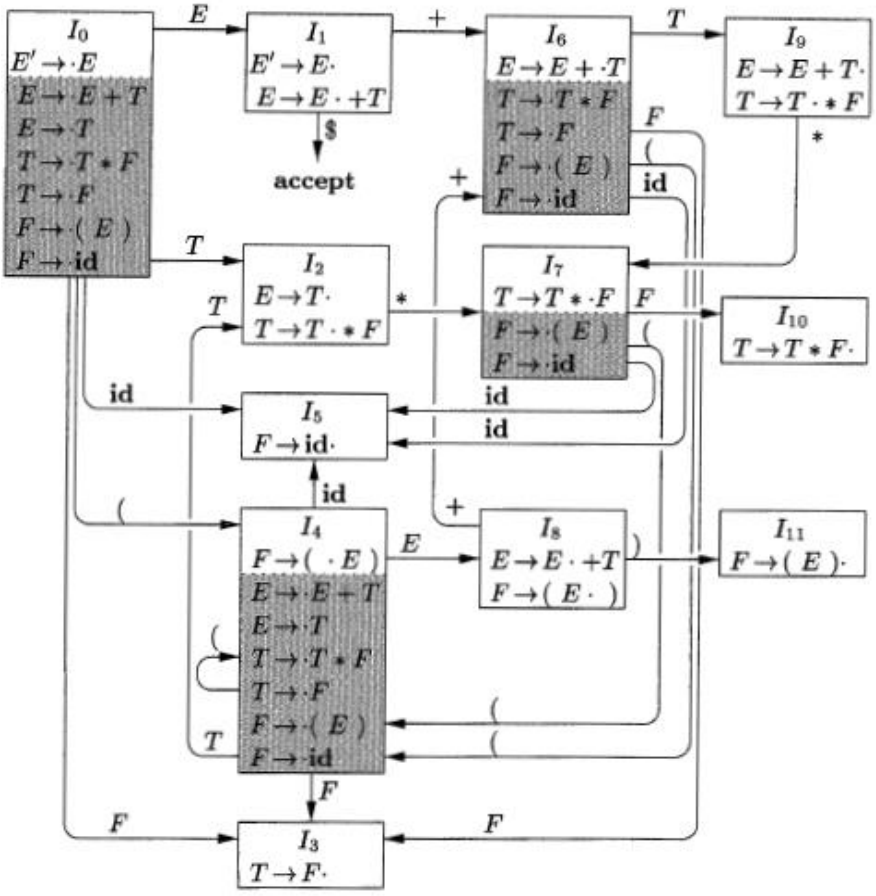
We are now ready for the algorithm to construct C, the canonical collection of sets of LR(0) items for an augmented grammar G' — the algorithm is shown in

```
void 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 new sets of items are added to C on a round;
}
```

Computation of the canonical collection of sets of LR(0) items

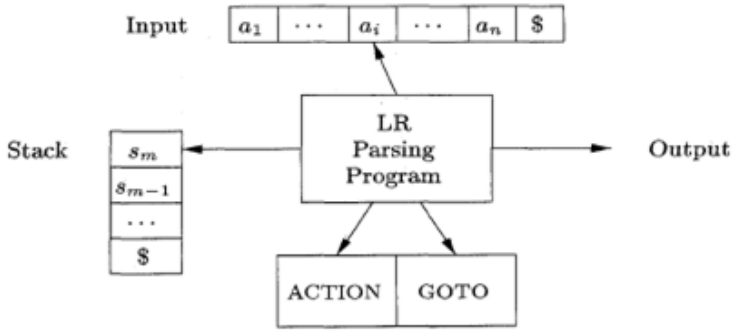
Canonical collection of set of LR (0) items and SLR parsing table for the following grammar.

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$



LR(0) automaton for the expression grammar

A schematic of an LR parser



Model of an LR parser

Structure of the LR Parsing Table

The parsing table consists of two parts: a parsing-action function ACTION and a goto function GOTO.

1. The ACTION function takes as arguments a state i and a terminal a (or \$, the input endmarker). The value of ACTION$[i, a]$ can have one of four forms:
   (a) Shift j, where j is a state. The action taken by the parser effectively shifts input a to the stack, but uses state j to represent a.
   (b) Reduce $A \rightarrow \beta$. The action of the parser effectively reduces β on the top of the stack to head A.
   (c) Accept. The parser accepts the input and finishes parsing.
   (d) Error. The parser discovers an error in its input and takes some corrective action. We shall have more to say about how such error-recovery routines work in Sections 4.8.3 and 4.9.4.
2. We extend the GOTO function, defined on sets of items, to states: if GOTO$[I_i, A] = I_j$, then GOTO also maps a state i and a nonterminal A to state j.

Behavior of the LR Parser

The next move of the parser from the configuration above is determined by reading a_i, the current input symbol, and s_m, the state on top of the stack, and then consulting the entry ACTION$[s_m, a_i]$ in the parsing action table. The configurations resulting after each of the four types of move are as follows

1. If ACTION$[s_m, a_i]$ = shift s, the parser executes a shift move; it shifts the next state s onto the stack, entering the configuration

$$(s_0 s_1 \cdots s_m s,\ a_{i+1} \cdots a_n \$)$$

   The symbol a_i need not be held on the stack, since it can be recovered from s, if needed (which in practice it never is). The current input symbol is now a_{i+1}.

2. If ACTION$[s_m, a_i]$ = reduce $A \rightarrow \beta$, then the parser executes a reduce move, entering the configuration

$$(s_0 s_1 \cdots s_{m-r} s,\ a_i a_{i+1} \cdots a_n \$)$$

   where r is the length of β, and s = GOTO$[s_{m-r}, A]$. Here the parser first popped r state symbols off the stack, exposing state s_{m-r}. The parser then pushed s, the entry for GOTO$[s_{m-r}, A]$, onto the stack. The current input symbol is not changed in a reduce move. For the LR parsers we shall construct, $X_{m-r+1} \cdots X_m$, the sequence of grammar symbols corresponding to the states popped off the stack, will always match β, the right side of the reducing production.

   The output of an LR parser is generated after a reduce move by executing the semantic action associated with the reducing production. For the time being, we shall assume the output consists of just printing the reducing production.

3. If ACTION$[s_m, a_i]$ = accept, parsing is completed.

4. If ACTION$[s_m, a_i]$ = error, the parser has discovered an error and calls an error recovery routine.

The LR-parsing algorithm is summarized below. All LR parsers behave in this fashion; the only difference between one LR parser and another is the information in the ACTION and GOTO fields of the parsing table.

Algorithm 4.44: LR-parsing algorithm.

INPUT: An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G.

OUTPUT: If w is in $L(G)$, the reduction steps of a bottom-up parse for w; otherwise, an error indication.

METHOD: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the program

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
    let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
            push t onto the stack;
            let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
            pop |β| symbols off the stack;
            let state t now be on top of the stack;
            push GOTO[t, A] onto the stack;
            output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

Constructing SLR-Parsing Tables

Algorithm Constructing an SLR-parsing table.

INPUT: An augmented grammar G'.

OUTPUT: The SLR-parsing table functions ACTION and GOTO for G'.

METHOD:

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(0) items for G'.

2. State i is constructed from I_i. The parsing actions for state i are determined as follows:

   (a) If $[A \to \alpha \cdot a\beta]$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to "shift j." Here a must be a terminal.

   (b) If $[A \to \alpha \cdot]$ is in I_i, then set $\text{ACTION}[i, a]$ to "reduce $A \to \alpha$" for all a in $\text{FOLLOW}(A)$; here A may not be S'.

   (c) If $[S' \to S \cdot]$ is in I_i, then set $\text{ACTION}[i, \$]$ to "accept."

   If any conflicting actions result from the above rules, we say the grammar is not SLR(1). The algorithm fails to produce a parser in this case.

3. The goto transitions for state i are constructed for all nonterminals A using the rule: If $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}[i, A] = j$.

4. All entries not defined by rules (2) and (3) are made "error."

5. The initial state of the parser is the one constructed from the set of items containing $[S' \to \cdot S]$.

STATE	ACTION						GOTO		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Parsing table for expression grammar

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		**id** * **id** + **id** $	shift
(2)	0 5	**id**	* **id** + **id** $	reduce by $F \rightarrow$ **id**
(3)	0 3	F	* **id** + **id** $	reduce by $T \rightarrow F$
(4)	0 2	T	* **id** + **id** $	shift
(5)	0 2 7	T *	**id** + **id** $	shift
(6)	0 2 7 5	T * **id**	+ **id** $	reduce by $F \rightarrow$ **id**
(7)	0 2 7 10	$T * F$	+ **id** $	reduce by $T \rightarrow T * F$
(8)	0 2	T	+ **id** $	reduce by $E \rightarrow T$
(9)	0 1	E	+ **id** $	shift
(10)	0 1 6	E +	**id** $	shift
(11)	0 1 6 5	E + **id**	$	reduce by $F \rightarrow$ **id**
(12)	0 1 6 3	$E + F$	$	reduce by $T \rightarrow F$
(13)	0 1 6 9	$E + T$	$	reduce by $E \rightarrow E + T$
(14)	0 1	E	$	accept

Moves of an LR parser on **id** * **id** + **id**

More Powerful LR Parsers

The extension of the previous LR parsing techniques to use one symbol of lookahead on the input. There are two different methods:

1. The "canonical-LR" or just "LR" method, which makes full use of the lookahead symbol(s). This method uses a large set of items, called the LR(1) items.
2. The "lookahead-LR" or "LALR" method, which is based on the LR(0) sets of items, and has many fewer states than typical parsers based on the LR(1) items. By carefully introducing lookaheads into the LR(0) items, we can handle many more grammars with the LALR method than with the SLR method, and build parsing tables that are no bigger than the SLR tables. LALR is the method of choice in most situations.

Canonical LR(1) Items

We shall now present the most general technique for constructing an LR parsing table from a grammar. Recall that in the SLR method, state i calls for reduction by A → α if the set of items Ii contains item [A → α] and input symbol I_i is in FOLLOW(A). In some situations, however, when state i appears on top of the stack, the viable prefix βα on the stack is such that βA cannot be followed by α in any right-sentential form. Thus, the reduction by A → α should be invalid on input a.

It is possible to carry more information in the state that will allow us to rule out some of these invalid reductions by $A \to \alpha$. By splitting states when necessary, we can arrange to have each state of an LR parser indicate exactly which input symbols can follow a handle α for which there is a possible reduction to A.

The extra information is incorporated into the state by redefining items to include a terminal symbol as a second component. The general form of an item becomes $[A \to \alpha \cdot \beta, a]$, where $A \to \alpha\beta$ is a production and a is a terminal or the right endmarker \$. We call such an object an *LR(1) item*. The 1 refers to the length of the second component, called the *lookahead* of the item.[6] The lookahead has no effect in an item of the form $[A \to \alpha\cdot\beta, a]$, where β is not ϵ, but an item of the form $[A \to \alpha\cdot, a]$ calls for a reduction by $A \to \alpha$ only if the next input symbol is a. Thus, we are compelled to reduce by $A \to \alpha$ only on those input symbols a for which $[A \to \alpha\cdot, a]$ is an LR(1) item in the state on top of the stack. The set of such a's will always be a subset of FOLLOW(A), but it could be a proper subset.

Formally, we say LR(1) item $[A \to \alpha\cdot\beta, a]$ is *valid* for a viable prefix γ if there is a derivation $S \underset{rm}{\overset{*}{\Rightarrow}} \delta A w \underset{rm}{\Rightarrow} \delta\alpha\beta w$, where

1. $\gamma = \delta\alpha$, and
2. Either a is the first symbol of w, or w is ϵ and a is \$.

Constructing LR(1) Sets of Items

SetOfItems CLOSURE(I) {
    repeat
        for (each item $[A \to \alpha \cdot B\beta, a]$ in I)
            for (each production $B \to \gamma$ in G')
                for (each terminal b in FIRST(βa))
                    add $[B \to \cdot\gamma, b]$ to set I;
    until no more items are added to I;
    return I;
}

SetOfItems GOTO(I, X) {
    initialize J to be the empty set;
    for (each item $[A \to \alpha \cdot X\beta, a]$ in I)
        add item $[A \to \alpha X \cdot \beta, a]$ to set J;
    return CLOSURE(J);
}

void *items*(G') {
    initialize C to $\{$CLOSURE($\{[S' \to \cdot S, \$]\}$)$\}$;
    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 new sets of items are added to C;
}

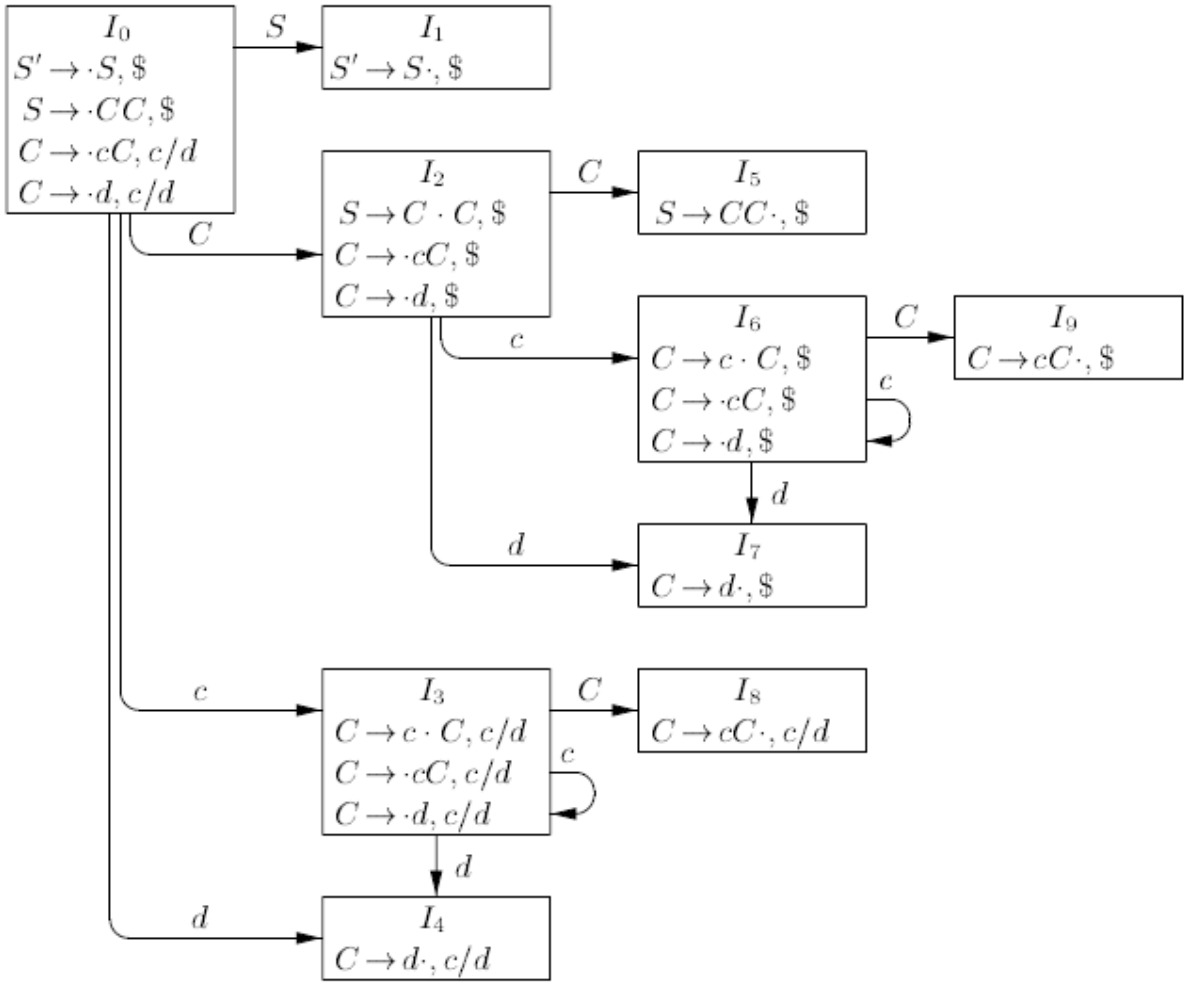


Figure The GOTO graph for grammar

Algorithm Construction of the sets of LR(1) items.

INPUT: An augmented grammar G'.

OUTPUT: The sets of LR(1) items that are the set of items valid for one or more viable prefixes of G'.

METHOD: The procedures CLOSURE and GOTO and the main routine *items* for constructing the sets of items were shown

Canonical LR(1) Parsing Tables

Algorithm Construction of canonical-LR parsing tables.

INPUT: An augmented grammar G'.

OUTPUT: The canonical-LR parsing table functions ACTION and GOTO for G'.

METHOD:

1. Construct $C' = \{I_0, I_1, \cdots, I_n\}$, the collection of sets of LR(1) items for G'.

2. State i of the parser is constructed from I_i. The parsing action for state i is determined as follows.

   (a) If $[A \to \alpha \cdot a\beta, b]$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to "shift j." Here a must be a terminal.

   (b) If $[A \to \alpha \cdot, a]$ is in I_i, $A \neq S'$, then set $\text{ACTION}[i, a]$ to "reduce $A \to \alpha$."

   (c) If $[S' \to S \cdot, \$]$ is in I_i, then set $\text{ACTION}[i, \$]$ to "accept."

   If any conflicting actions result from the above rules, we say the grammar is not LR(1). The algorithm fails to produce a parser in this case.

3. The goto transitions for state i are constructed for all nonterminals A using the rule: If $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}[i, A] = j$.

4. All entries not defined by rules (2) and (3) are made "error."

5. The initial state of the parser is the one constructed from the set of items containing $[S' \to \cdot S, \$]$.

STATE	ACTION			GOTO	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Canonical parsing table for grammar

Constructing LALR Parsing Tables

We now introduce our last parser construction method, the LALR (lookahead- LR) technique. This method is often used in practice, because the tables obtained by it are considerably smaller than the canonical LR tables, yet most common syntactic constructs of programming languages can be expressed conveniently by an LALR grammar. The same is almost true for SLR grammars, but there are a few constructs that cannot be conveniently handled by SLR techniques.

Algorithm An easy, but space-consuming LALR table construction.

INPUT: An augmented grammar G'.

OUTPUT: The LALR parsing-table functions ACTION and GOTO for G'.

METHOD:

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(1) items.
2. For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by their union.

3. Let $C' = \{J_0, J_1, \ldots, J_m\}$ be the resulting sets of LR(1) items. The parsing actions for state i are constructed from J_i in the same manner as in Algorithm 4.56. If there is a parsing action conflict, the algorithm fails to produce a parser, and the grammar is said not to be LALR(1).

4. The GOTO table is constructed as follows. If J is the union of one or more sets of LR(1) items, that is, $J = I_1 \cup I_2 \cup \cdots \cup I_k$, then the cores of $\text{GOTO}(I_1, X)$, $\text{GOTO}(I_2, X), \ldots, \text{GOTO}(I_k, X)$ are the same, since $I_1, I_2, \ldots, I_k$ all have the same core. Let K be the union of all sets of items having the same core as $\text{GOTO}(I_1, X)$. Then $\text{GOTO}(J, X) = K$.

Example Again consider grammar (4.55) whose GOTO graph was shown in Fig. 4.41. As we mentioned, there are three pairs of sets of items that can be merged. I_3 and I_6 are replaced by their union:

$$\begin{array}{lll} I_{36}: & C \to c \cdot C, & c/d/\$ \\ & C \to \cdot cC, & c/d/\$ \\ & C \to \cdot d, & c/d/\$ \end{array}$$

I_4 and I_7 are replaced by their union:

$$I_{47}: \quad C \to d \cdot, \ c/d/\$$$

and I_8 and I_9 are replaced by their union:

$$I_{89}: \quad C \to cC \cdot, \ c/d/\$$$

The LALR action and goto functions for the condensed sets of items are shown in Fig

STATE	ACTION			GOTO	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

University Questions/Question bank:

1. Define PDA. Design a PDA to accept the following language. L={a^n b^n ; n>=0}. Draw the transition diagram for the constructed PDA. Show the ID's for the string aaabbb.
2. Construct a PDA to accept the language L= {ww^R | wϵ{a,b}* }.Draw the graphical representation of this PDA.Show the moves made by this PDA for the string aabbaa.
3. Obtain a PDA to accept the language L(M) = {wCw^R|wt(a+b)*},where W^R is reverse of W by a final state.
4. Discuss the language accepted by a PDA Design a PDA to accept the following languages L= {0^{2n} 1^n ; n≥1} Draw the Transition diagram for the constructed PDA. Also the moves made by PDA for the string "000011"
5. Design a PDA for accepting a^{2n} b^n.
6. What is Syntax analysis? With a neat diagram explain the role of parser.
7. Explain different error recovery strategies.
8. Give the formal definition of CFG with an example.
9. What is derivation? Consider the following context free grammar

   $E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \mathbf{id}$ and the input string – (id + id).

   Give LMD and RMD and Parse tree. Is the grammar ambiguous? why
10. Define Ambiguity. Is the following grammar ambiguous? If yes remove the ambiguity and re-write the grammar.

$$\begin{array}{lcl} stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\ & \mid & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\ & \mid & \textbf{other} \end{array}$$

11. What is left recursion? How left recursion can be eliminated from the grammars? Write down the simple arithmetic expression grammar and re-write the grammar after eliminating the left recursion.
12. What is left factoring? How left factoring can be eliminated from the grammars? Explain with suitable example.
13. Give the Rules for constructing the FIRST and FOLLOW set.
14. Give an algorithm to construct predictive parsing table. Construct the predictive parsing the table by making the necessary changes to the following grammar and show the parsing of the string id + id *id.

E→E+T|T

T→T*F|F

F→(E)|id

15. Explain with a neat diagram, the model of a table driven predictive parser (Non-recursive Predictive Parsing).
16. List and explain the different Error Recovery in Predictive Parsing.
17. What is handle pruning? Give bottom-up parse for the input string id*id for the grammar.
18. What is shift reduce parser? Explain its actions and conflicts by taking an example.
19. Write the schematic of LR parser (SLR parser).
20. Design SLR parser for the following grammar by computing LR(0) items and show the parsing of string ((a))

    A→(A)|a
21. Write the canonical collection of set of LR (0) items and SLR parsing table for the following grammar.

    S→CC

    C→cC|d
22. Write an algorithm to compute LR(1) sets of items.
23. Write the canonical collection of set of LR (0) items and SLR parsing table for the following grammar.

    S→AA

    A→Aa|b
24. How LALR parsing table is constructed? Develop an algorithm for the same.
25. Construct the LALR parser for

    S→AA

    A→Aa|b