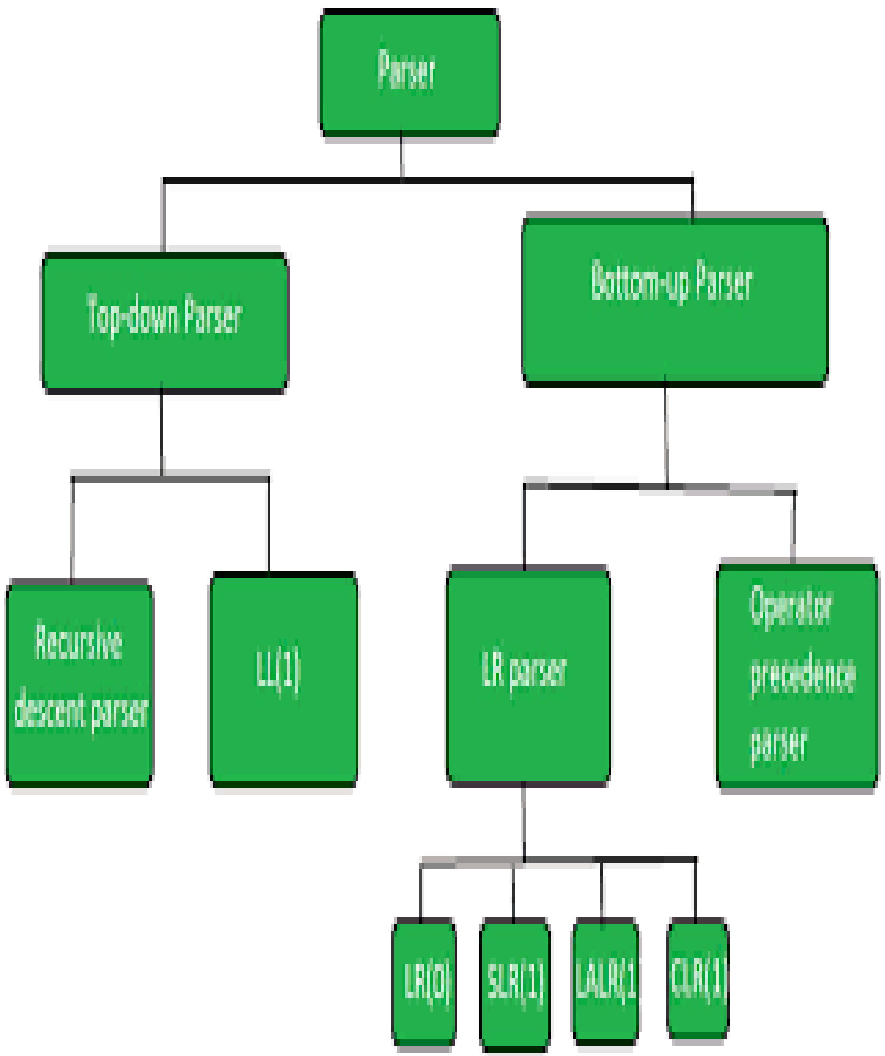
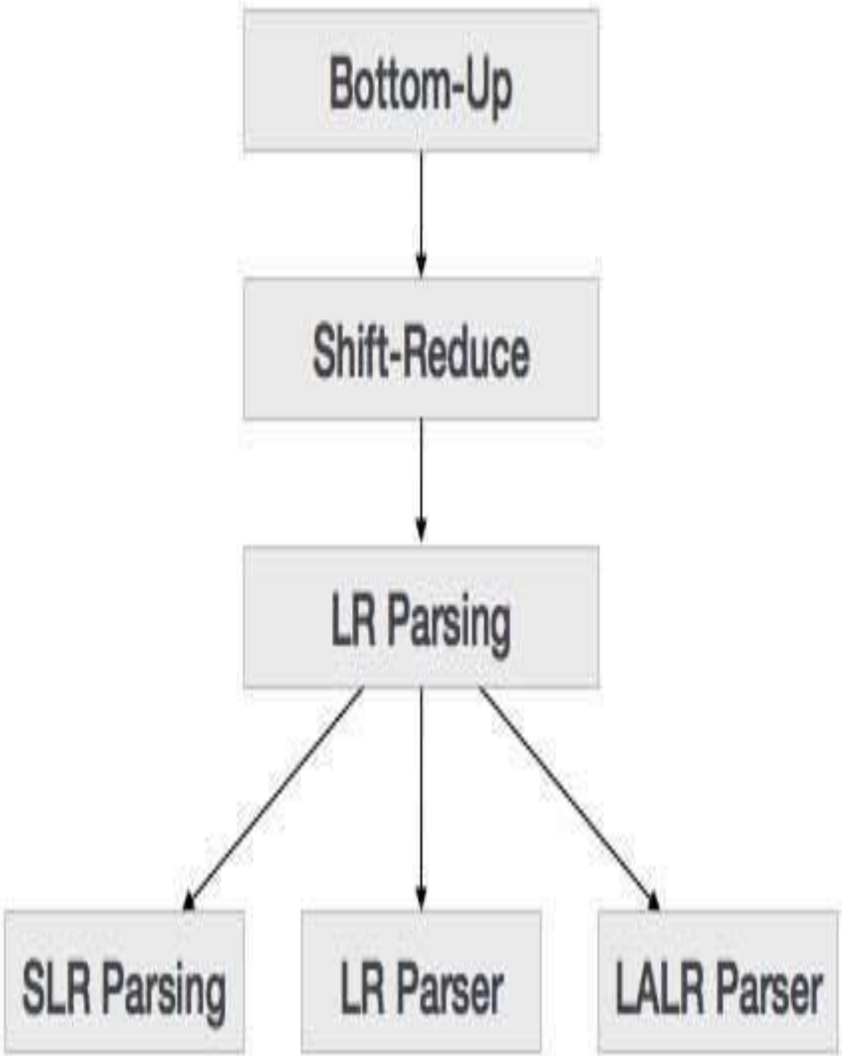
BOTTOM-UP PARSING

Parsing – Types





Bottom up parsing

- Also known as Shift- Reduce parsing.
- It attempts to construct a parse tree from an input string beginning at the leaf nodes and working up towards the root node.
- At each step, a particular substring matching the right side of a production is replaced by a symbol on the left side of that production and if chosen correctly at each step, a rightmost production is traced out in reverse.

Bottom-Up Parsing

- Bottom-Up Parser : Constructs a parse tree for an input string beginning at the leaves(the bottom) and working up towards the root(the top)
- We can think of this process as one of “reducing” a string w to the start symbol of a grammar
- Bottom-up parsing is also known as *shift-reduce parsing* because its two main actions are shift and reduce.
  - At each shift action, the current symbol in the input string is pushed to a stack.
  - At each reduction step, the symbols at the top of the stack (this symbol sequence is the right side of a production) will replaced by the non-terminal at the left side of that production.

Shift-Reduce Parsing

- A shift-reduce parser tries to reduce the given input string into the starting symbol.

  a string $\rightarrow$ the starting symbol
  reduced to

- At each reduction step, a substring of the input matching to the right side of a production rule is replaced by the non-terminal at the left side of that production rule.
- If the substring is chosen correctly, the right most derivation of that string is created in the reverse order.

Rightmost Derivation: $S \overset{*}{\underset{rm}{\Rightarrow}} \omega$

Shift-Reduce Parser finds: $\omega \underset{rm}{\Leftarrow} \ldots \underset{rm}{\Leftarrow} S$

Shift-Reduce Parsing-Example

- Consider the grammar

  $S \longrightarrow aABe$

  $A \longrightarrow Abc \mid b$

  $B \longrightarrow d$

Input string : abbcde

aAbcde

aAde ⇓ reduction

aABe

S

We can scan *abbcde* looking for a substring that matches the right side of some production. The substrings *b* and *d* qualify.Let us choose left most *b* and replace it by *A*, the left side of the production A→b; we thus obtain the string *aAbcde*. Now the substrings *Abc*, *b* and *d* match the right side of some production. Although b is the leftmost substring that matches the right side of the some production, we choose to replace the substring *Abc* by *A*, the left side of the production A→Abc. We obtain *aAde*. Then replacing *d* by *B*, and then replacing the entire string by *S*. Thus, by a sequence of four reductions we are able to reduce *abbcde* to *S*.

Shift-Reduce Parsing-Example

- These reductions in-fact trace out the following right-most derivation in reverse

$$S \underset{rm}{\Rightarrow} aABe \underset{rm}{\Rightarrow} aAde \underset{rm}{\Rightarrow} aAbcde \underset{rm}{\Rightarrow} abbcde$$

Right Sentential Forms

- How do we know which substring to be replaced at each reduction step?

Handle

- Informally, a "handle" of a string is a substring that matches the right side of the production, and whose reduction to nonterminal on the left side of the production represents one step along the reverse of a rightmost derivation
  - But not every substring matches the right side of a production rule is handle. If we have replaced 'b' by A in the second string 'aAbcde' , we would have obtained the string 'aAAcde' , which could not be reduced to S.
- Formally , a "handle" of a right sentential form γ ($\equiv \alpha\beta\omega$) is a production

rule $A \rightarrow \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ.

$$S \overset{*}{\underset{rm}{\Rightarrow}} \alpha A\omega \underset{rm}{\Rightarrow} \alpha\beta\omega$$

then $A \rightarrow \beta$ in the position following α is a handle of $\alpha\beta\omega$.

- The string ω to the right of the handle contains only terminal symbols. If a grammar is unambiguous, then every right sentential form of the grammar has exactly one handle.

- Consider the grammar:
- E →E+E
- E → E*E
- E →(E)
- E → id

parsing of “id+id*id”

E → E+E

E → E+E*E

E → E+E*id

E → E+id*id

E → id+id*id

This give ‘*’ a higher precedence than ‘+’.

- Another parsing strategy for "id+id*id"

$E \rightarrow \underline{E*E}$

$E \rightarrow E*\underline{id}$

$E \rightarrow \underline{E+E}*id$

$E \rightarrow E+\underline{id}*id$

$E \rightarrow \underline{id}+id*id$

This give '+' a higher precedence than '*'.

Example

- Here, *abbcde* is a right sentential form whose handle is $A \rightarrow b$ at position 2. Likewise, *aAbcde* is a right sentential form whose handle is $A \rightarrow Abc$ at position 2.
- Sometimes we say "the substring β is a handle of αβω" if the position of β and the production A→β we have in mind are clear.

Handle Pruning

- A rightmost derivation in reverse can be obtained by “handle pruning”. That is, we start with a string of terminals w that we wish to parse. If ω is a sentence of grammar at hand, then ω = γ, where γ_n is the nth right-sentential form of some as yet unknown rightmost derivation.

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \dots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = \omega$$

Input string

Handle Pruning

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \dots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = \omega$$

- Start from γ_n, find a handle $A_n \rightarrow \beta_n$ in γ_n, and replace β_n in by A_n to get γ_{n-1}.
- Then find a handle $A_{n-1} \rightarrow \beta_{n-1}$ in γ_{n-1}, and replace β_{n-1} in by A_{n-1} to get γ_{n-2}.
- Repeat this, until we reach S.

A Shift-Reduce Parser

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$

Right-Most Derivation of `id+id*id`

$E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*id \Rightarrow E+F*id$
$\Rightarrow E+id*id \Rightarrow T+id*id \Rightarrow F+id*id \Rightarrow id+id*id$

Right-Most Sentential form	HANDLE	Reducing Production
id+id*id	id	F→id
F+id*id	F	T→F
T+id*id	T	E→T
E+id*id	id	F→id
E+F*id	F	T→F
E+T*id	Id	F→id
E+T*F	T*F	T→T*F
E+T	E+T	E→E+T
E		

A Stack Implementation of a Shift-Reduce Parser

- There are four possible actions of a shift-parser action:

    1. Shift : The next input symbol is shifted onto the top of the stack.

    2. Reduce: Replace the handle on the top of the stack by the non-terminal.

    3. Accept: Successful completion of parsing.

    4. Error: Parser discovers a syntax error, and calls an error recovery routine.

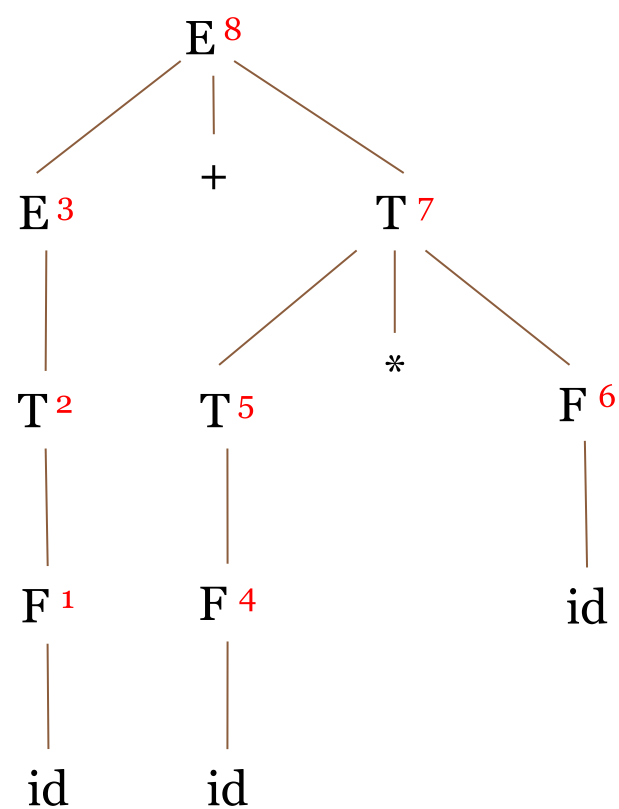
- Initial stack just contains only the end-marker $.

- The end of the input string is marked by the end-marker $.

A Stack Implementation of A Shift-Reduce Parser

Stack	Input	Action
$	id+id*id$shift	
$id	+id*id$	Reduce by F→id
$F	+id*id$	Reduce by T→F
$T	+id*id$	Reduce by E→T
$E	+id*id$	Shift
$E+	Id*id$	Shift
$E+id	*id$	Reduce by F→id
$E+F	*id$	Reduce by T→F
$E+T	*id$	Shift
$E+T*	id$	Shift
$E+T*id	$	Reduce by F→id
$E+T*F	$	Reduce by T→T*F
$E+T	$	Reduce by E →E+T
$E	$	Accept

Parse Tree



Example

- Consider the following grammar-
- $E \rightarrow E+T \mid T$
- $T \rightarrow T * F \mid F$
- $F \rightarrow (E) \mid id$

- Parse the input string id * id using a shift-reduce parser.

Solution

Stack	Input Buffer	Parsing Action
$	id * id $	Shift
$ id	* id $	Reduce F → id
$F	* id $	Reduce T → F
$T	* id $	Shift Reduce Conflict (Handle is T) Shift
$T*	id$	Shift
$T*id	$	Reduce F → id
$T*F	$	Reduce-Reduce conflict (Handles are F, T, T*F) Reduce T → T*F
$T	$	Reduce E → T
$E	$	Accept

Conflicts in Bottom up Parsing

- Two types of conflicts are associated with Bottom up parsing. They are:

1. **<u>Shift Reduce Conflict</u>**.

A *shift-reduce* conflict occurs in a state that requests both a shift action and a reduce action.

2. **<u>Reduce Reduce Conflict</u>**.

A *reduce-reduce* conflict occurs in a state that requests two or more different reduce actions.

- Consider the grammar:

$E \rightarrow 2E2$

$E \rightarrow 3\ E\ 3$

$E \rightarrow 4$

Perform Shift Reduce parsing for input string "32423".

Actions of the parser

Stack	Input Buffer	Parsing Action
$	32423$	Shift
$3	2423$	Shift
$32	423$	Shift
$324	23$	Reduce by E --> 4
$32E	23$	Shift
$32E2	3$	Reduce by E --> 2E2
$3E	3$	Shift
$3E3	$	Reduce by E --> 3E3
$E	$	Accept

Example

- Consider the following grammar-
- $S \rightarrow (L) \mid a$
- $L \rightarrow L , S \mid S$

- Parse the input string (a , (a , a)) using a shift-reduce parser.

Solution:

Stack	Input Buffer	Parsing Action
$	(a , (a , a)) $	Shift
$ (	a , (a , a)) $	Shift
$ (a	, (a , a)) $	Reduce S → a
$ (S	, (a , a)) $	Reduce L → S
$ (L	, (a , a)) $	Shift
$ (L ,	(a , a)) $	Shift
$ (L , (	a , a)) $	Shift
$ (L , (a	, a)) $	Reduce S → a
$ (L , (S	, a)) $	Reduce L → S

Solution (Contd.)

Stack	Input Buffer	Parsing Action
$ (L , (L	, a)) $	Shift
$ (L , (L ,	a)) $	Shift
$ (L , (L , a	)) $	Reduce S → a
$ (L , (L , S)	)) $	Reduce L → L , S
$ (L , (L	)) $	Shift
$ (L , (L)	) $	Reduce S → (L)
$ (L , S	) $	Reduce L → L , S
$ (L	) $	Shift
$ (L)	$	Reduce S → (L)
$ S	$	Accept

Example:

- Consider the following grammar-

$S \rightarrow T\ L$

$T \rightarrow int \mid float$

$L \rightarrow L\ ,\ id \mid id$

- Parse the input string int id , id ; using a shift-reduce parser.

Solution

Stack	Input Buffer	Parsing Action
\$	int id , id ; \$	Shift
\$ int	id , id ; \$	Reduce T → int
\$ T	id , id ; \$	Shift
\$ T id	, id ; \$	Reduce L → id
\$ T L	, id ; \$	Shift
\$ T L ,	id ; \$	Shift
\$ T L , id	; \$	Reduce L → L , id
\$ T L	; \$	Shift
\$ T L ;	\$	Reduce S → T L
\$ S	\$	Accept

COMPILER DESIGN
(CS 1703)

INTRODUCTION TO LR PARSERS

LR PARSERS

- Efficient, Bottom-up syntax analysis that can parse a large class of CFG.
- Technique is called LR(k) parsing: ‘L’ represents Left to Right scanning, ‘R’ represents righty-most derivation in the reverse and ‘k’ represents the number of input symbols of look ahead that can be used for taking parsing decisions. When (k) is omitted, it is assumed to be 1.

Advantages of LR parsers

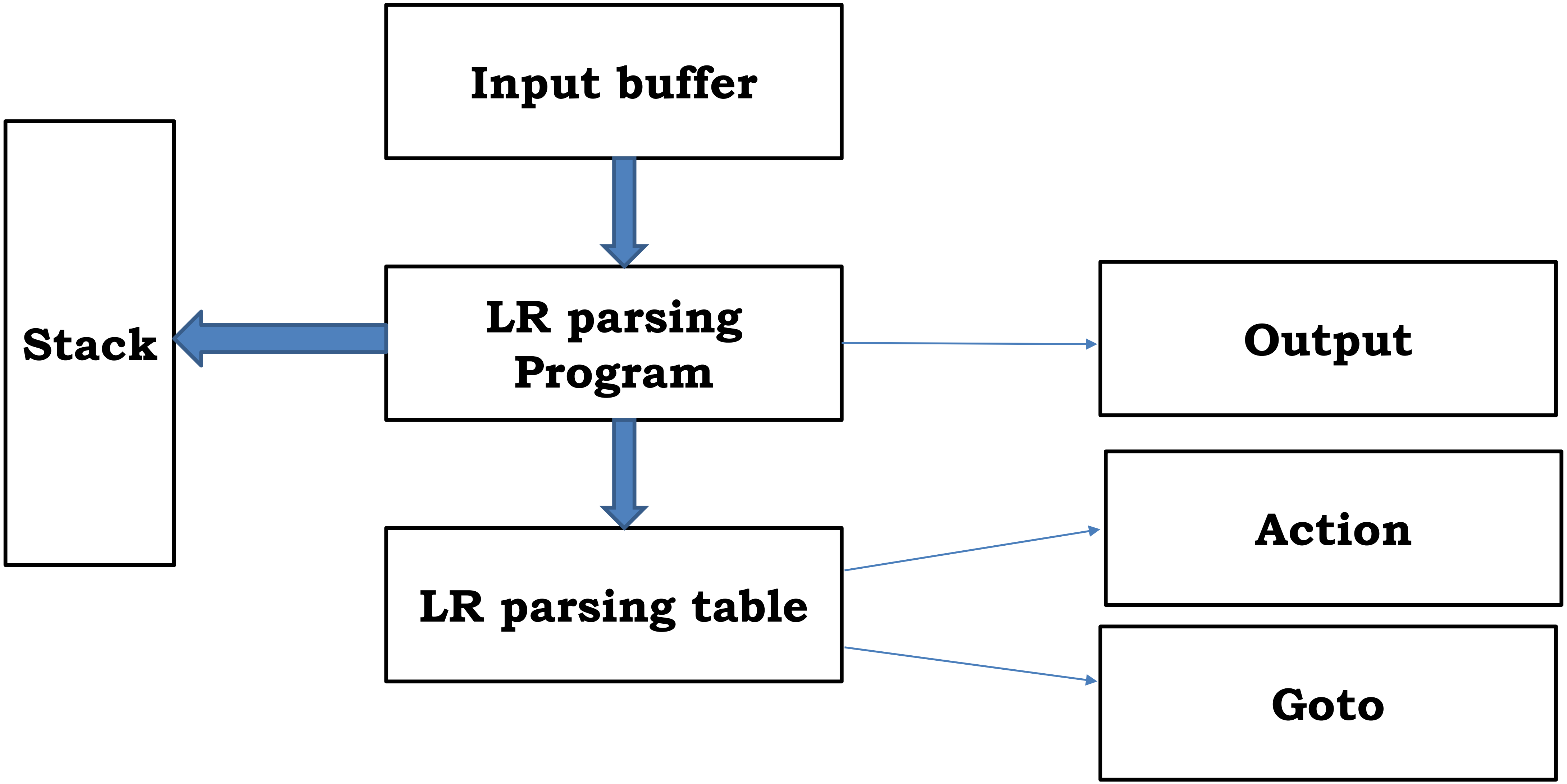
- Can be used for recognizing virtually all programming language constructs for which a CFG can be written.
- It is the most general non back-tracking shift-reduce parsing method known.
- Class of grammars that can be parsed using LR methods is a ***proper superset*** of the class of grammars that can be parsed with predictive parsers.
- Can detect a syntactic error as soon as it is possible to do so on a left-to-right scanning of the input.

- Principal drawback: Too much work needs to be done in order to design an LR parser by hand for a typical programming language grammar. But specialized LR parser generators are available. (Ex: YACC).

Types of LR parser

1. LR(0)
2. SLR : Simple LR- easiest to implement.
3. CLR(1): Canonical LR- Most powerful and most expensive.
4. LALR(1): Look-Ahead LR- Intermediate in power.

Working of LR Parser



Working of LR Parser contd..

- LR parser consists of an input, an output, a stack, a driver program and a parsing table that has two functions
  1. ***Action***
  2. ***Goto***
- The driver program is same for all LR parsers. ***Only the parsing table changes from one parser to another.***

Working of LR Parser contd..

- The parsing program reads character from an input buffer one at a time, where a shift reduces parser would shift a symbol; an LR parser ***shifts a state***.
- Each state summarizes the information contained in the stack.
- ***States represent a set of "items"***

- Program uses a stack to store a string of the form $\mathbf{s_0X_1s_1X_2s_2.......X_ms_m}$, after reading from the input buffer one at a time. Each $\mathbf{X}_i$ is a grammar symbol and $\mathbf{s_i}$ is a state.

- Parser table is divided into 2 parts namely:
- 1. Action.
- 2. Goto.

- Behaviour of the LR parser is as follows:
- it determines $\mathbf{s_m}$ **…** on the top of the stack and $\mathbf{a_i}$, the current input symbol. It then consults action[], the parsing action table entry for state $\mathbf{s_m}$ and $\mathbf{a_i}$, which can have any of the four values:
- 1. Shift s, where s is a state.
- 2. Reduce by a grammar by the rule $A \rightarrow \beta$
- 3. Accept
- 4. Error.

- The Goto function takes a state and grammar symbols as arguments and produces a state.

- A configuration of an LR parser is a pair whose first component is the stack contents and the second component is the unread input represented as:
- $(s_0X_1s_1X_2s_2\ldots\ldots X_ms_{m,}\ a_i\ a_{i+1}\ a_{i+2}\ldots\ldots\ a_n\ \$)$
- It represents the right sentential form:
- $X_1\ X_2\ldots\ldots X_m\ \ a_i\ a_{i+1}\ a_{i+2}\ldots\ldots\ a_n$

LR(0) Parser

- An LR(O)parser is a shift-reduce parser that uses **zero** tokens of look-ahead to determine what action to take (hence the 0).
- This means that in any configuration of the parser, the parser must have an unambiguous action to choose-either it shifts a specific symbol or applies a specific reduction.
- If there are ever two or more choices to make, the parser fails and the grammar is not LR(O).

LR(0) Parser

LR(O) Items:

- An LR(O) item of a grammar G is a production of G with a ***dot*** at some position of the body.

Example:

A → •XYZ

- One collection of set of LR(O) items, called the ***canonical LR(O) collection***, provides finite automaton that is used to make parsing decisions.
- Such an automaton is called an LR(O) automaton.

LR(0) Items

$A \rightarrow \bullet XYZ$

$A \rightarrow X \bullet YZ$

$A \rightarrow XY \bullet Z$

$A \rightarrow XYZ \bullet$

$A \rightarrow \varepsilon$ will generate only one item, $A \rightarrow \bullet$

LR(0) Parser

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 I to **CLOSURE(I)**.
  2. If A —> α • Bβ is in CLOSURE(I) and B —> ɣ is a production, then add the item B —> • ɣ to CLOSURE(I), if it is not already there. Apply this rule until no more items can be added to CLOSURE (I).

LR(0) Parser

$S \rightarrow AA$

$A \rightarrow aA \mid b$

LR(0) Parser

1. Add a production grammar: Augment the grammar

S' → S

S→AA
A→aA|b

LR(0) Parser

2. Generate the items and closures:

S' → • S

Include productions of S closure

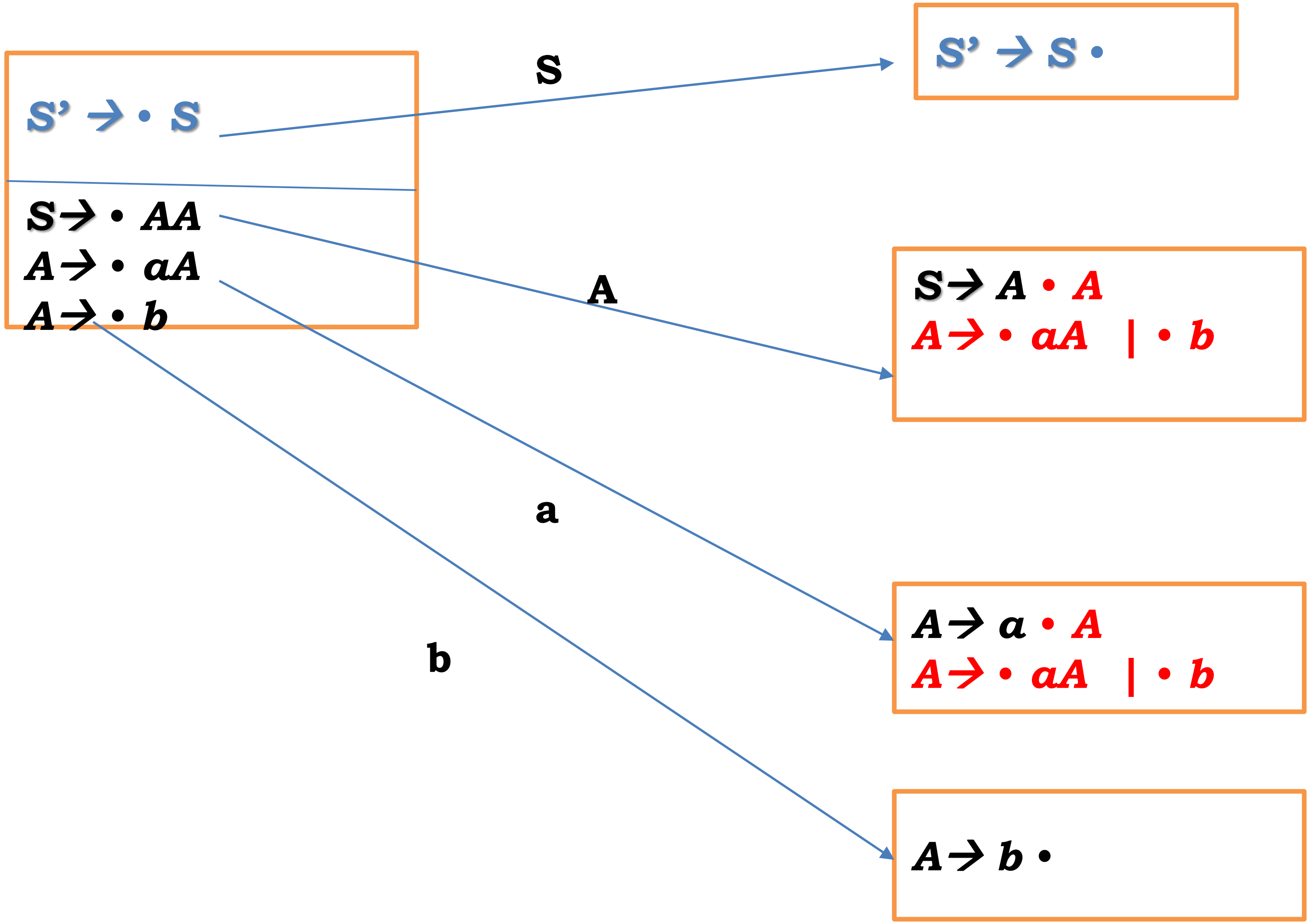
– S→ • AA

Include production of A closure

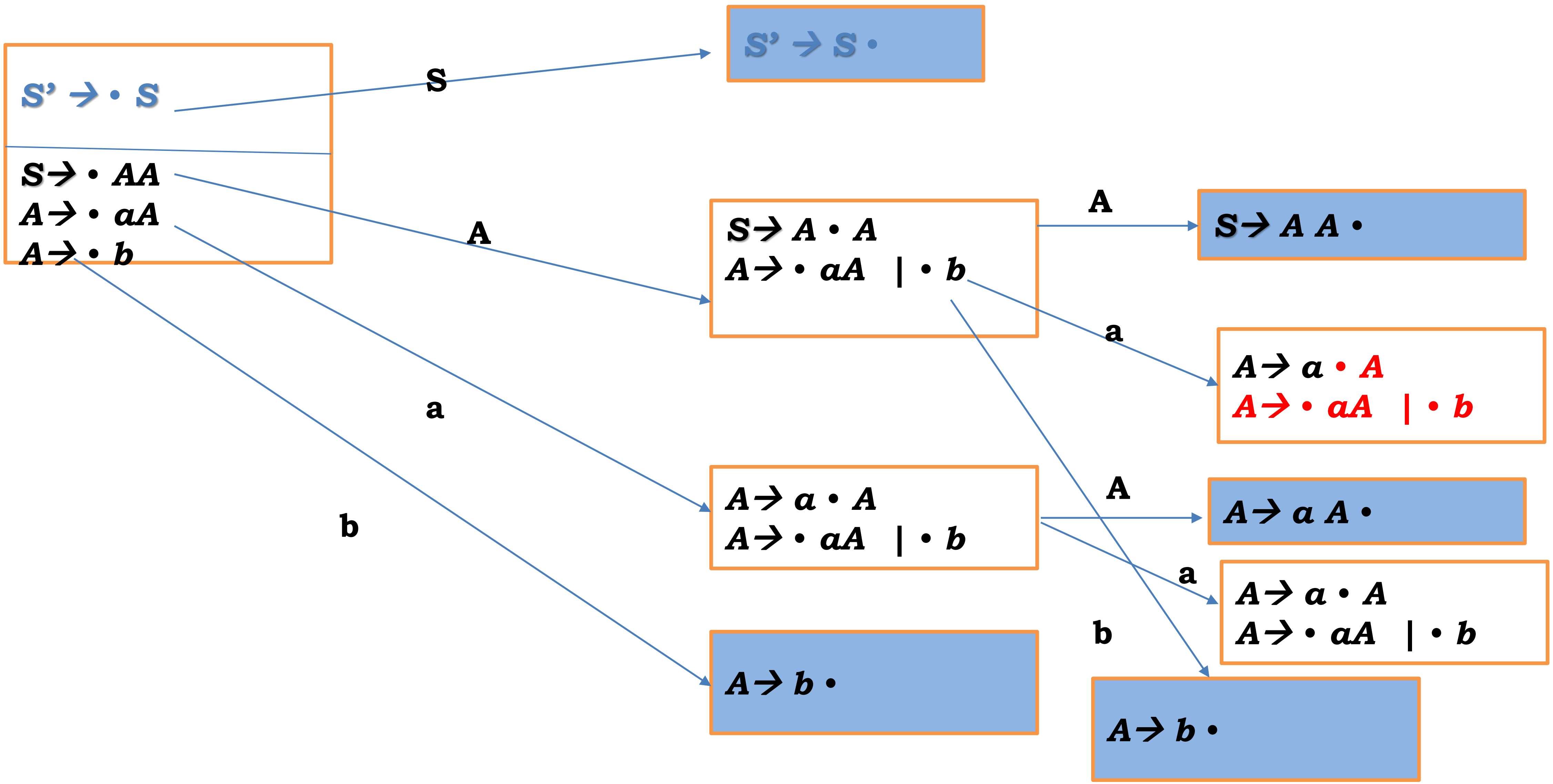
– A→ • aA | • b

S→AA
A→aA|b

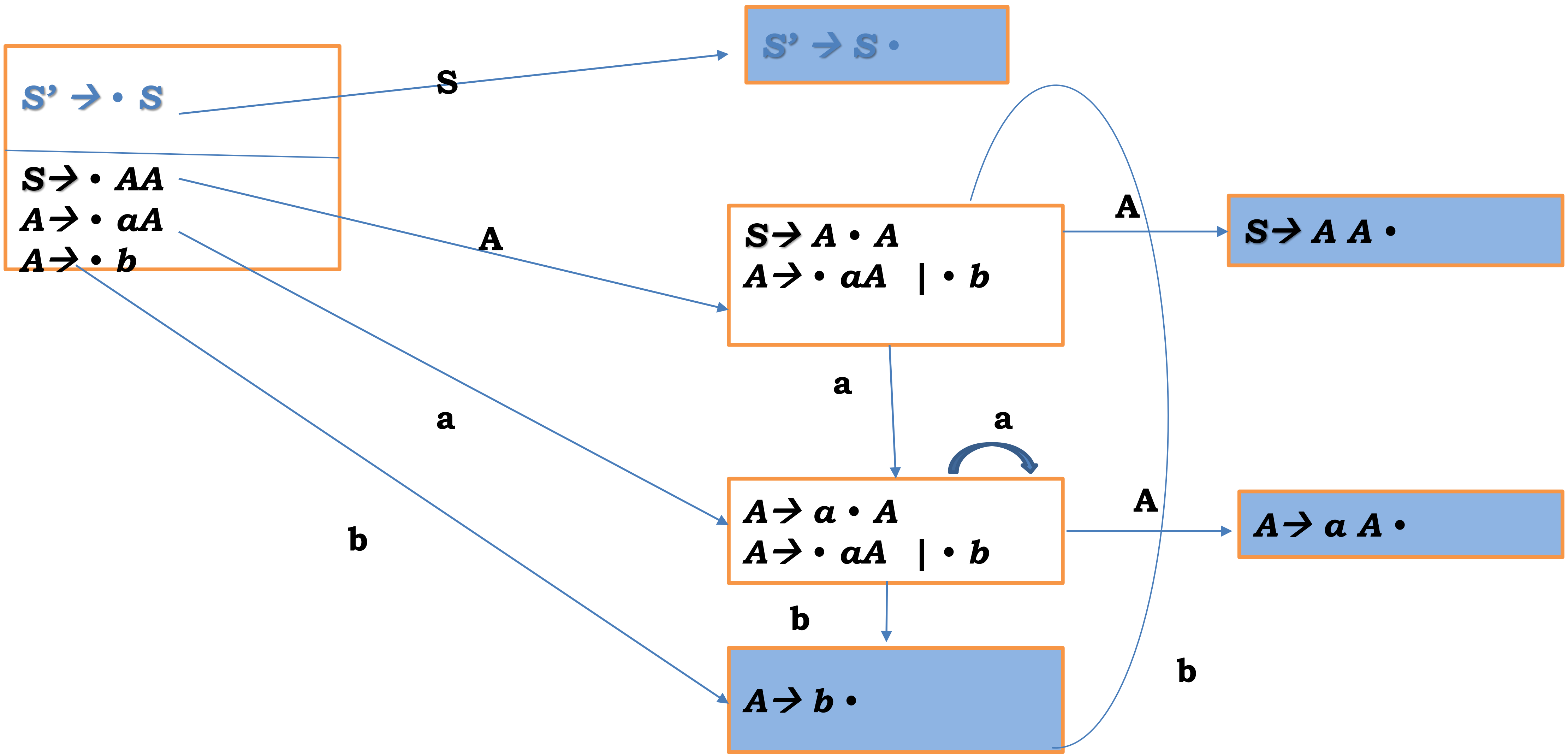
LR(0) Parser



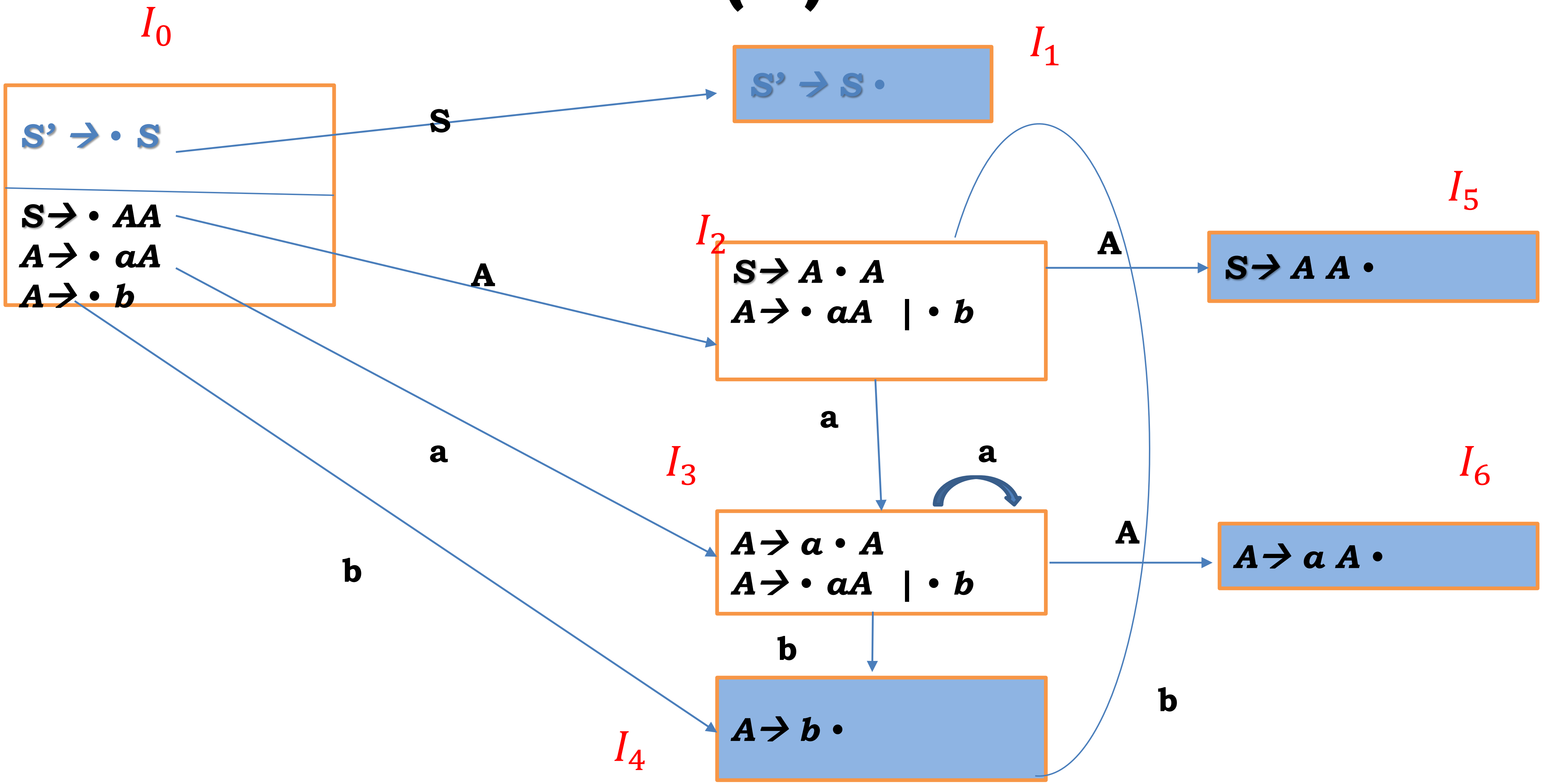
LR(0) Parser


S' → • S
S → • AA
A → • aA
A → • b
S
S' → S •
A
S → A • A
A → • aA | • b
A
S → A A •
a
A → a • A
A → • aA | • b
a
A → a • A
A → • aA | • b
A
A → a A •
a
A → a • A
A → • aA | • b
b
A → b •
b
A → b •

LR(0) Parser

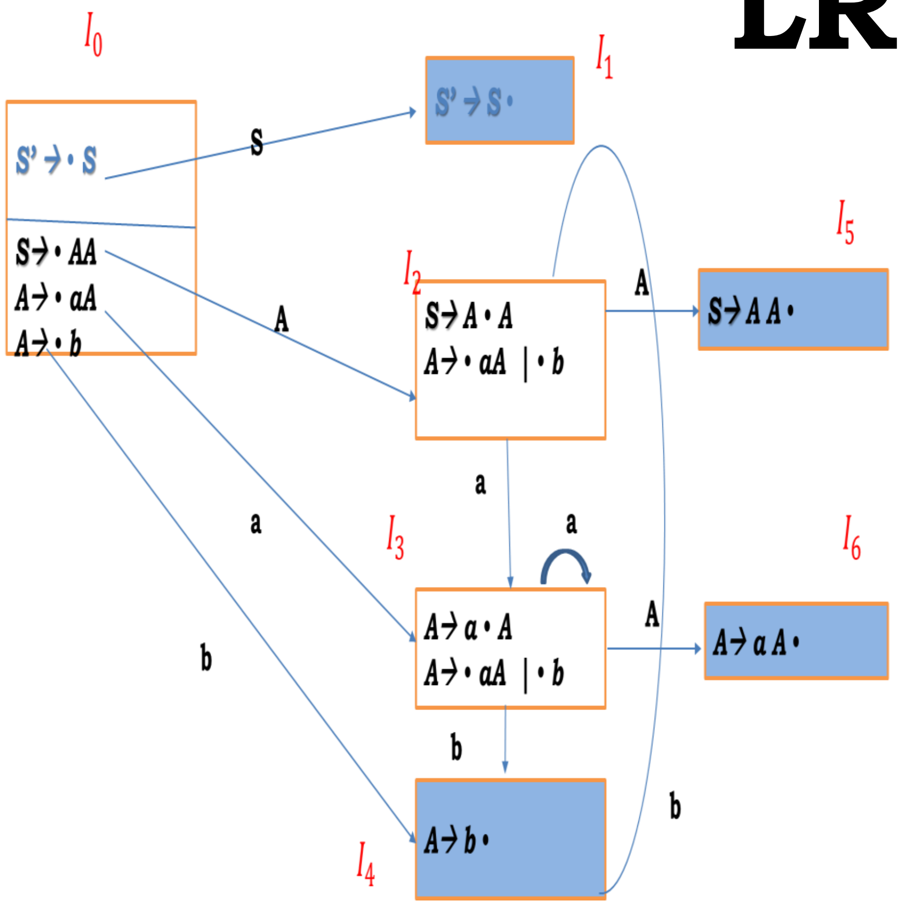


LR(0) Parser



I_0
S' → • S
S→ • AA
A→ • aA
A→ • b
S
I_1
S' → S •
A
I_2
S→ A • A
A→ • aA | • b
A
I_5
S→ A A •
a
a
a
I_3
A→ a • A
A→ • aA | • b
A
I_6
A→ a A •
b
b
b
I_4
A→ b •

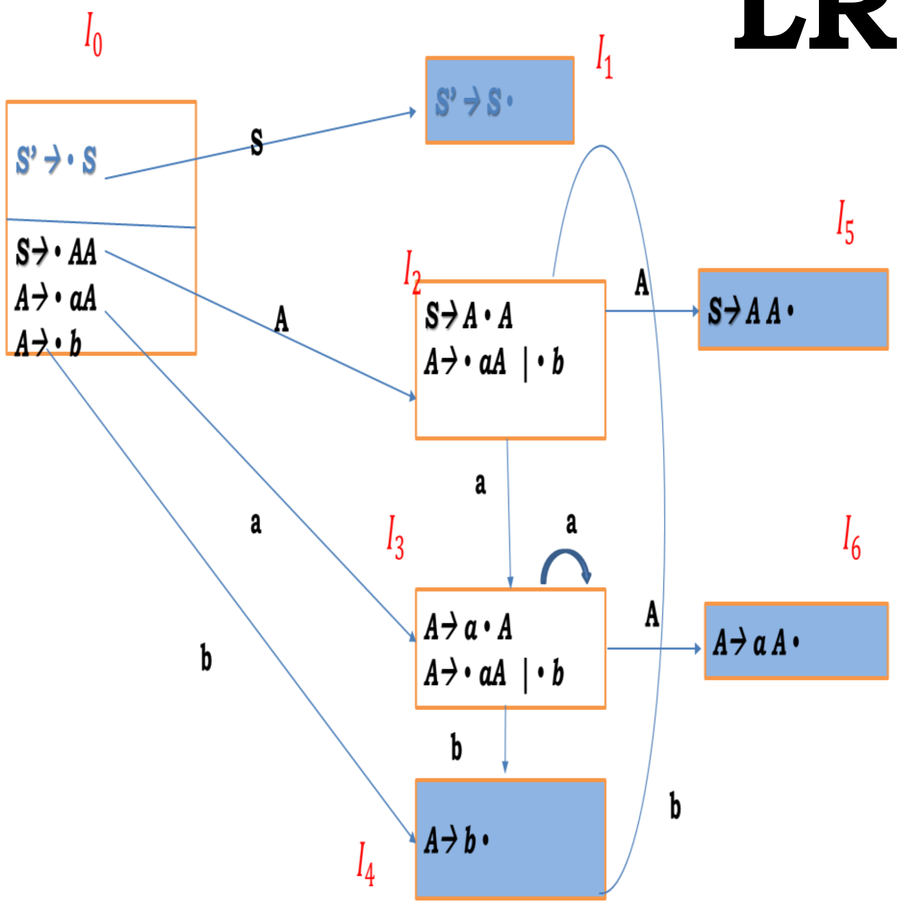
LR(0) Parsing Table



	Action			Goto	
States	a	b	$	A	S
0					
1					
2					
3					
4					
5					
6					

1. S→AA
2. A→aA
3. A→b

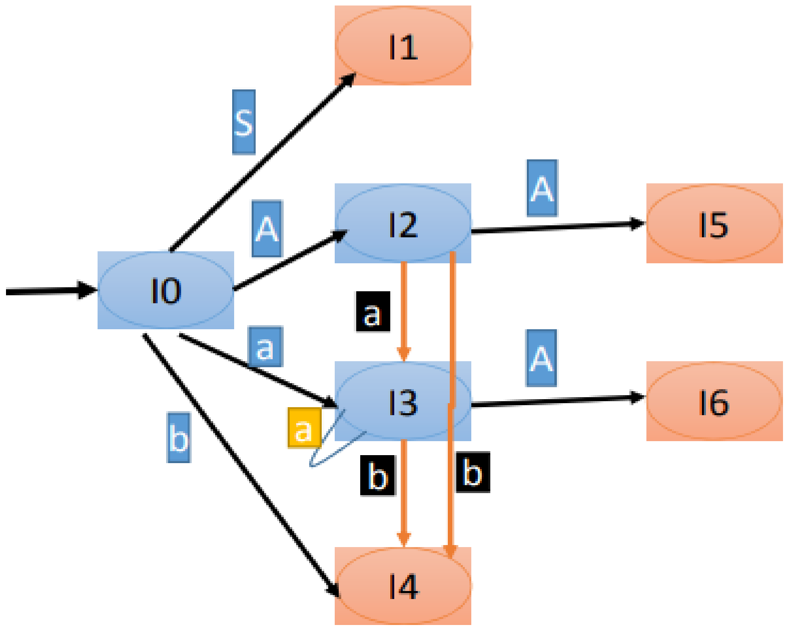
LR(0) Parsing Table



	Action			Goto	
States	**a**	**b**	**$**	**A**	**S**
0	S3	S4		2	1
1			Accept		
2	S3	S4		5	
3	S3	S4		6	
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

1. S→AA
2. A→aA
3. A→b

SLR(1) Parsing Table



	a	b	$	S	A
	ACTION			GOTO	
I0	S3	S4		1	2
I1			Accept		
I2	S3	S4			5
I3	S3	S4			6
I4	r3	r3	r3		
I5			r1		
I6	r2	r2	r2		

Follow(S) = $
Follow(A) = {a,b,$}

Rules for construction of parsing table from Canonical collections of LR(0) items

Action part: For Terminal Symbols

- If $A \rightarrow \alpha.a\beta$ is state Ix in Items and **goto(Ix,a)=Iy** then set action [Ix,a]=Sy (represented as shift to state Iy]
- If $A \rightarrow \alpha.$ is in Ix, then set **action[Ix,f]** to reduce **$A \rightarrow \alpha$** for all symbols "f" where "f" is in Follow(A) (Use rule number)
- If $S' \rightarrow S.$ is in Ix then set action[Ix,$]=accept.

Go To Part: For Non Terminal Symbols

- If **goto(Ix, A) = Iy**, then goto(Ix,A) in table **= Y**
- It is numeric value of state Y.
- All other entries are considered as error.
- Initial state is $S' \rightarrow .S$

Steps in processing of the string ***abb***

Stack	Input	Action
\$0	abb\$	Shift 3
\$0a3	bb\$	Shift 4
\$0a3b4	b\$	Reduce by A→b
\$0a3A	b\$	Goto 6 (Intermediate Step)
\$0a3A6	b\$	Reduce by A→aA
\$0A	b\$	Goto 2 (Intermediate Step)
\$0A2	b\$	Shift 4
\$0A2b4	b\$	Reduce by A→b
\$0A2A5	\$	Reduce by S→AA
\$0S1	\$	Accept

- Consider the grammar below:

A → (A)

A → a

The corresponding Augmented grammar can be written as:

A' → A -----→ 1

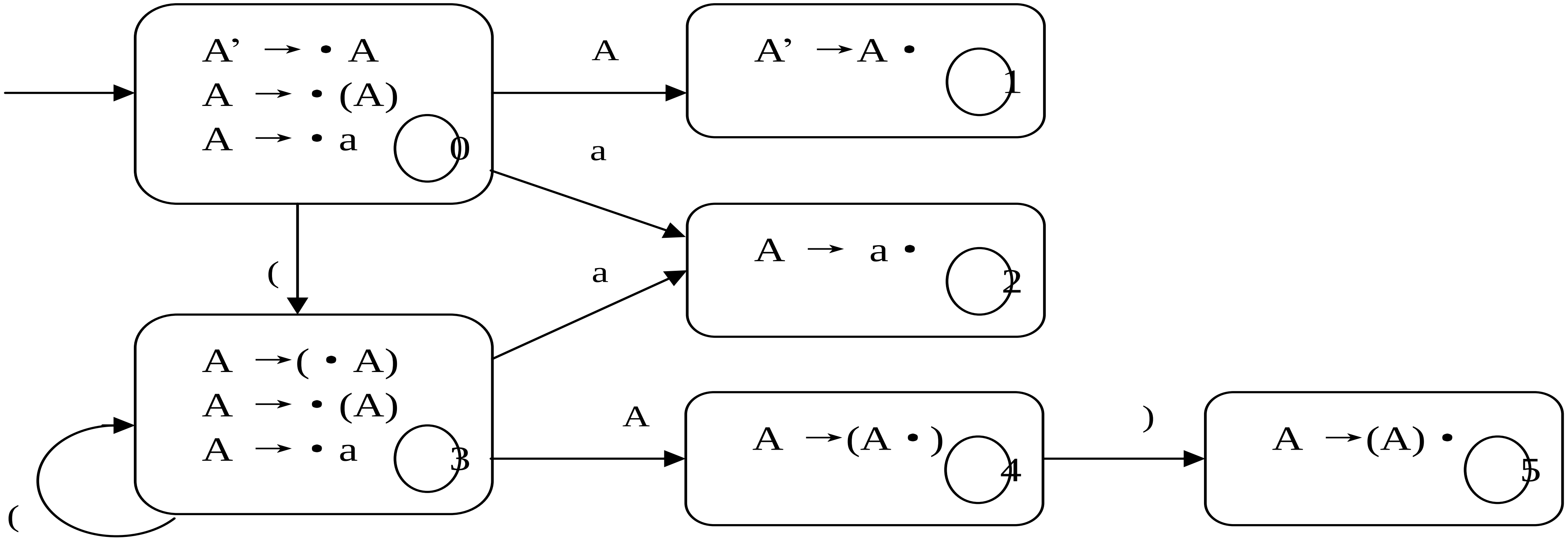
A → .(A) ---→ 2

A → .a ----→ 3

This grammar has eight items in its closure set:

$$A' \rightarrow \cdot A$$
$$A' \rightarrow A\cdot$$
$$A \rightarrow \cdot(A)$$
$$A \rightarrow (\cdot A)$$
$$A \rightarrow (A\cdot)$$
$$A \rightarrow (A)\cdot$$
$$A \rightarrow \cdot a$$
$$A \rightarrow a\cdot$$

DFA construction of closure sets



LR(0) Parsing Table

	Action				Goto
States	**(**	**a**	**)**	**$**	**A**
0	s3	s2			1
1				Accept	
2	r3	r3	r3		
3	s3	s2			4
4			S5		
5	r2	r2	r2		

Augmented Grammar:

A’ → .A --→ 1
A → .(A) -→ 2
A → a ---→ 3

SLR(1), called simple LR(1) parsing, uses the DFA of sets of LR(0) items .

SLR(1) increases the power of LR(0) parsing significant by using the next token in the input string.

- First, it consults the input token *before* a shift to make sure that an appropriate DFA transition exists.
- Second, it uses the Follow set of a non-terminal to decide if a reduction should be performed.

Definition of The SLR(1) parsing algorithm(1)

Let s be the current state,

actions are defined as follows: .

1.If state s contains any item of form $A \rightarrow \alpha \cdot X\beta$

where X is a terminal, and

X is the next token in the input string,

then to shift the current input token onto the stack,

and push the new state containing the item

$A \rightarrow \alpha X \cdot \beta$

2. If state s contains the complete item $A \rightarrow \gamma \cdot$,

and the next token in input string is in Follow(A)

then to reduce by the rule $A \rightarrow \gamma$

Definition of The SLR(1) parsing algorithm(2)

2. (Continue)

A reduction by the rule $S' \rightarrow S$, is equivalent to acceptance;

- This will happen only if the next input token is \$.

In all other cases, Remove the string γ and a corresponding states from the parsing stack

- Correspondingly, back up in the DFA to the state from which the construction of γ began.
- This state must contain an item of the form $B \rightarrow \alpha \cdot A\beta$.

Push A onto the stack, and the state containing the item

$B \rightarrow \alpha A \cdot \beta$.

Definition of The SLR(1) parsing algorithm(3)

3. If the next input token is such that neither of the above two cases applies,
   - an error is declared

A grammar is an SLR(l) grammar if the application of the above SLR(1) parsing rules results in *no ambiguity*

A grammar is SLR(1) if and only if, for any state s, the following two conditions are satisfied:

- For any item $A \rightarrow \alpha \cdot X\beta$ in s with X a terminal,
  There is no complete item $B \rightarrow \gamma.$ in s with X in Follow(B).
- For any two complete items $A \rightarrow \alpha\cdot$ and $B \rightarrow \beta\cdot$ in s, Follow(A) ∩ Follow(B) is empty.

A violation of the first of these conditions represents a **shift-reduce conflict.**

A violation of the second of these conditions represents a **reduce-reduce conflict.**

SLR(1) Parsing Table

	Action				Goto
States	**(**	**a**	**)**	**$**	**A**
0	s3	s2			1
1				Accept	
2			r3	r3	
3	s3	s2			4
4			S5		
5			r2	r2	

Augmented Grammar:

A’ → .A → 1
A → .(A) → 2
A → a → 3

FOLLOW (A’)={$}
FOLLOW (A)={),$}

Steps in processing of the string *((a))*

Stack	Input	Action
$0	((a))$	Shift
$0(3	(a))$	Shift
$0(3(3	a))$	Shift
$0(3(3a2	))$	Reduce by A →a
$0(3(3A4	))$	Shift
$0(3(3A4)5	)$	Reduce by A→(A)
$0(3A4	)$	Shift
$0(3A4)5	$	Reduce by A→(A)
$0A1	$	Accept

Semantic Analysis

Semantic Analysis

- Semantic Analyzer
- Attribute Grammars
- Top-Down Translators
- Bottom-Up Translators
- Recursive Evaluators
- Type Checking

Semantic Analysis in Compiler Design

- Third phase of compiler Design.

- Makes sure that declarations and statements are semantically correct.

- Collection of procedures called by the parser as and when required by the grammar.

- Syntax tree and symbol table are used to check the consistency of the given code.

- Information is subsequently used by the compiler during the intermediate code generation.

Semantic Errors:

1. Type mismatch.

2. Undeclared variables.

3. Reserved identifier misuse.

4. Multiple declaration of a variable in a scope.

5. Accessing an out of scope variable.

6. Actual and formal parameter mismatch.

Functions of Semantic Analysis

1. Type Checking:

   Ensures that data types are used in a way that is consistent with their definition.

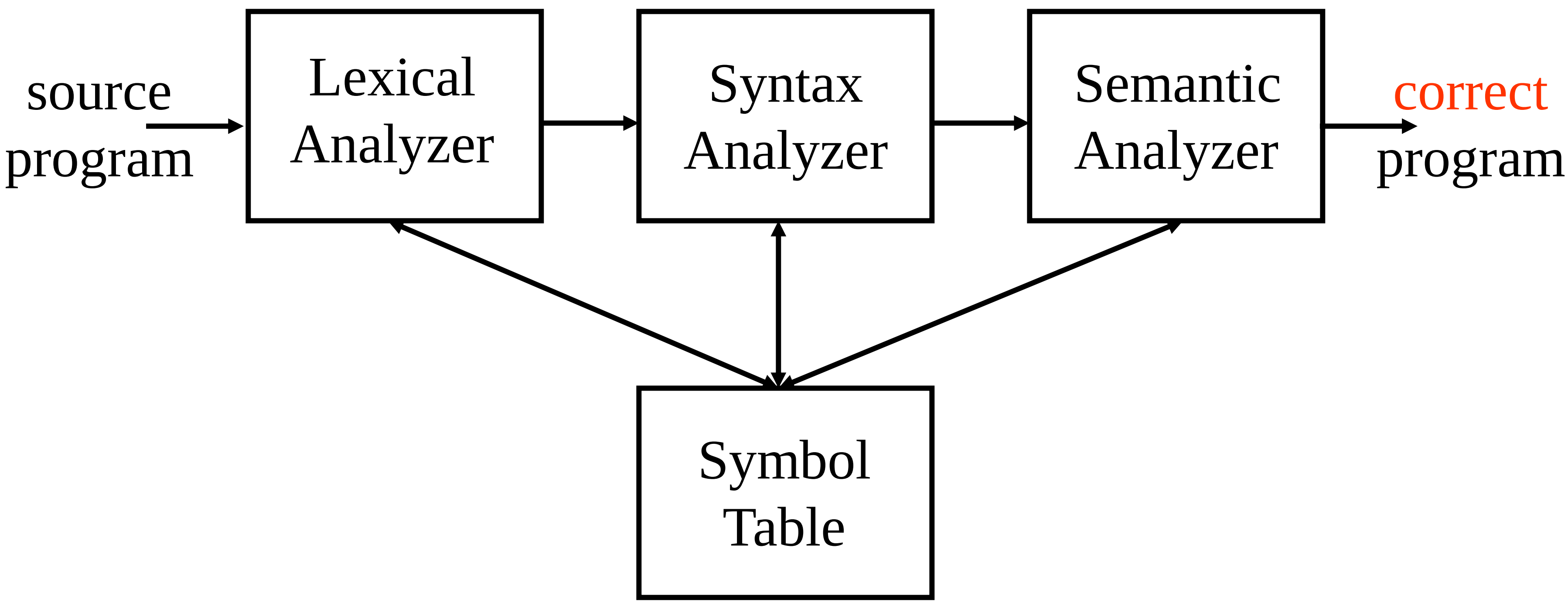
2. Label checking.

   A program should contain labels and references.

3. Flow-control check.

   Keeps a check that control structures are used in a proper manner.

Semantic Analyzer



Semantics

- *Type* of each construct
- *Interpretation* of each construct
- *Translation* of each construct

Attribute Grammars

- An *attribute grammar* is a context free grammar with associated *attributes* and *semantic rules*
- Each *grammar symbol* is associated with a set of *attributes*
- Each *production* is associated with a set of *semantic rules* for computing attributes

Attribute grammars (Contd.)

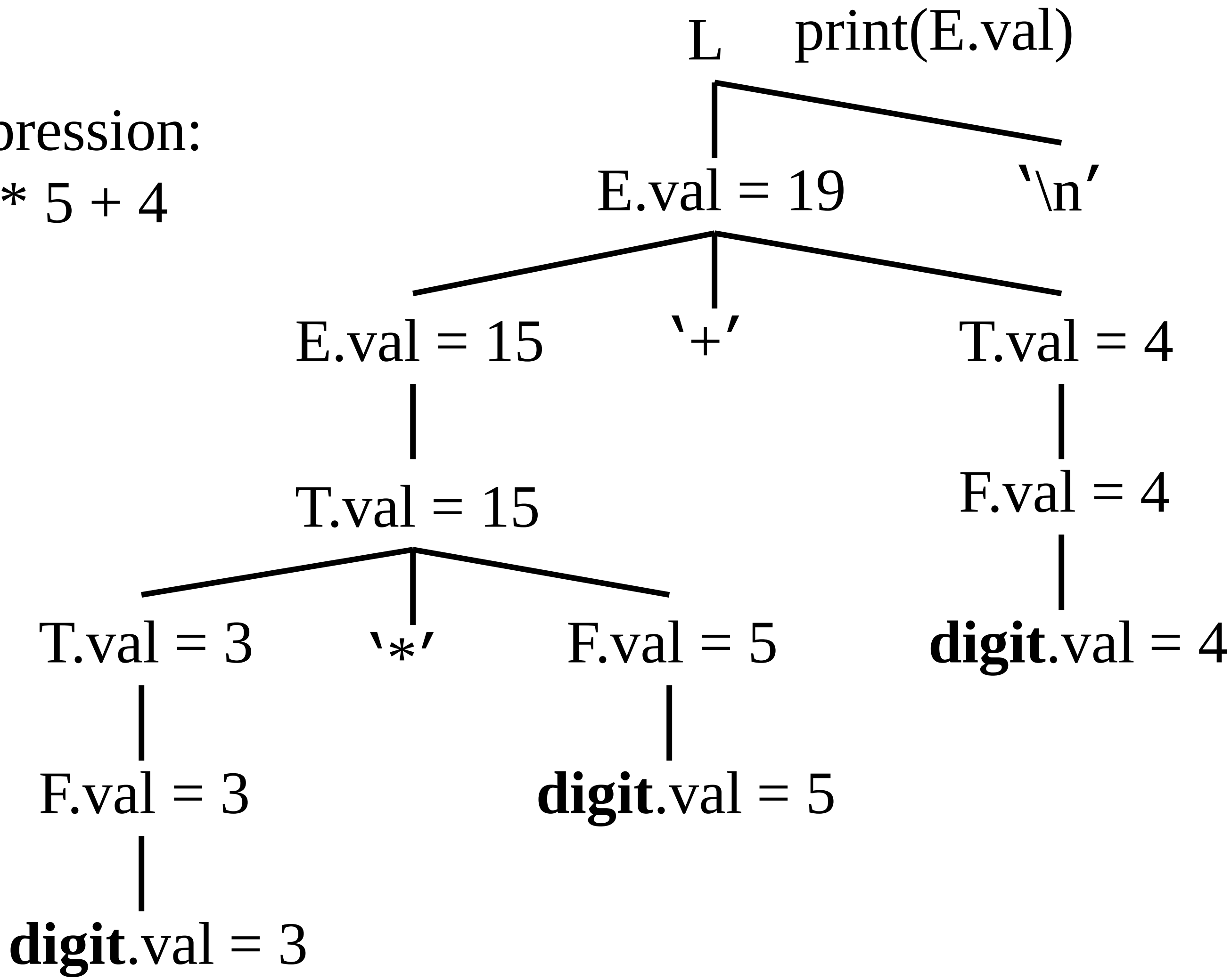
- Each attribute has well defined domain of values such as integer, float, character, string and expressions.

- Attribute grammar can pass values or information among the nodes in a parse tree.

Production	Semantic Rules
$L \rightarrow E$ '\n'	print(E.val)
$E \rightarrow E1 + T$	E.val:=E1.val + T.val
$E \rightarrow T$	E.val:=T.val
$T \rightarrow T1 * F$	T.val:=T1.val * F.val
$T \rightarrow F$	T.val:=F.val
$F \rightarrow (E)$	F.val:=E.val
$F \rightarrow digit$	F.val:=digit.val

Annotated Parse Trees

Expression:
3 * 5 + 4



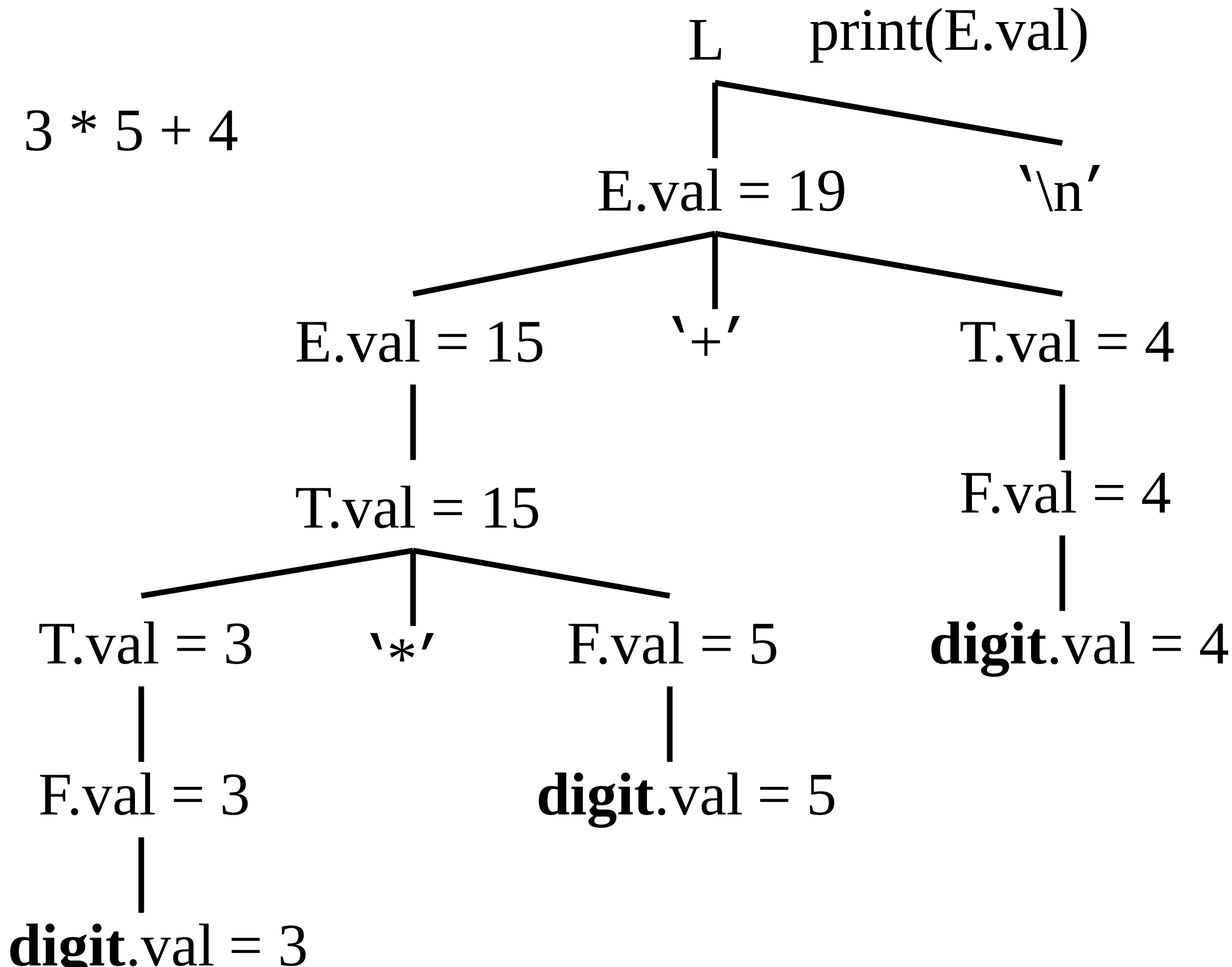
Attributes

- An attribute of a node (grammar symbol) in the parse tree is *synthesized* if its value is computed from that of its children
- An attribute of a node in the parse tree is *inherited* if its value is computed from that of its parent and siblings

Production	Semantic Rules
L → E '\n'	print(E.val)
E → E1 + T	E.val:=E1.val + T.val
E → T	E.val:=T.val
T → T1 * F	T.val:=T1.val * F.val
T → F	T.val:=F.val
F → (E)	F.val:=E.val
F → digit	F.val:=digit.val

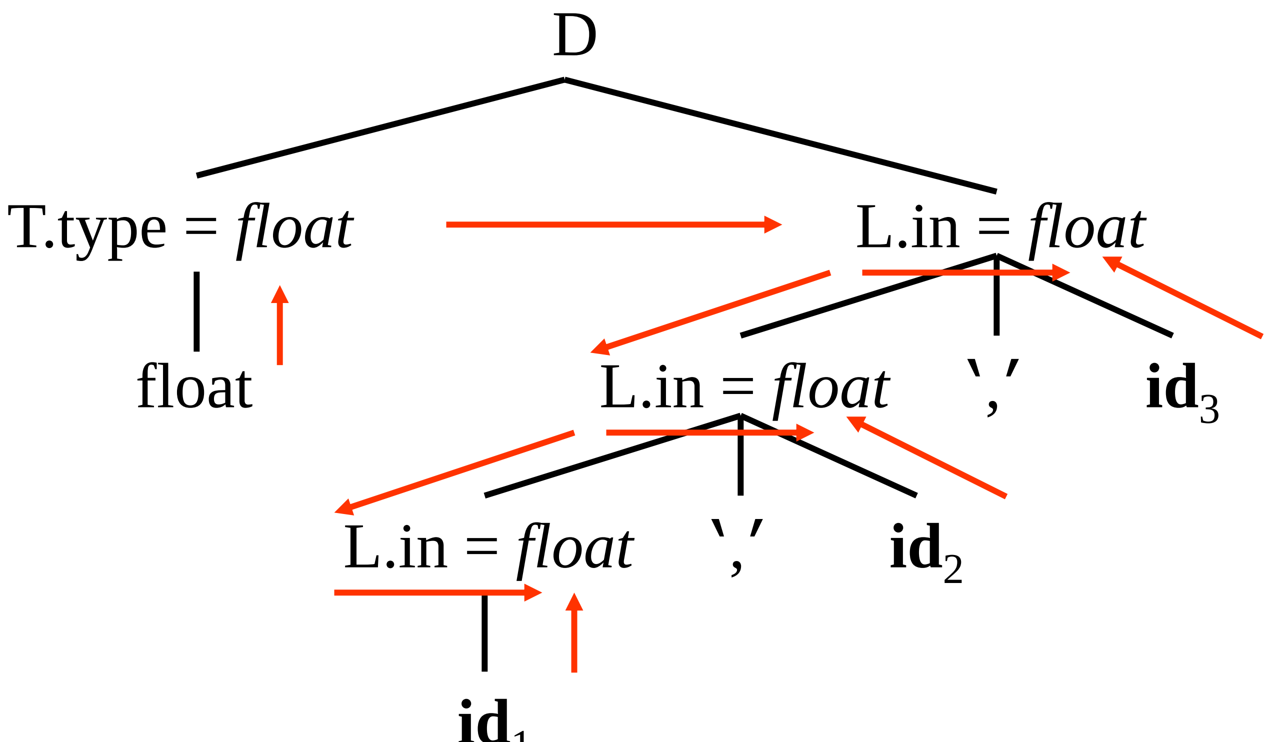
Synthesized Attributes

3 * 5 + 4



Production	Semantic Rules
$D \rightarrow TL$	L.in:=T.type
$T \rightarrow$ int	T.type:=integer
$T \rightarrow$ float	T.type:=integer
$L \rightarrow L1$ ‘,’ id	L1.in:=L.in addtype(id.entry, L.in)
$L \rightarrow$ id	addtype(id.entry, L.in)

Inherited Attributes



Two Notations

- Syntax-Directed Definitions
- Translation Schemes

Syntax-Directed Definitions

- Each grammar production $A \rightarrow \alpha$ is associated with a set of semantic rules of the form

$$b:=f(c_1,c_2,\ldots,c_k)$$

where f is a function and

1. b is a *synthesized* attribute of A and c_1, c_2, …, c_k are attributes of A or grammar symbols in α,

or

2. b is an *inherited* attribute of one of the grammar symbols in α and c_1, c_2, …, c_k are attributes of A or grammar symbols in α.

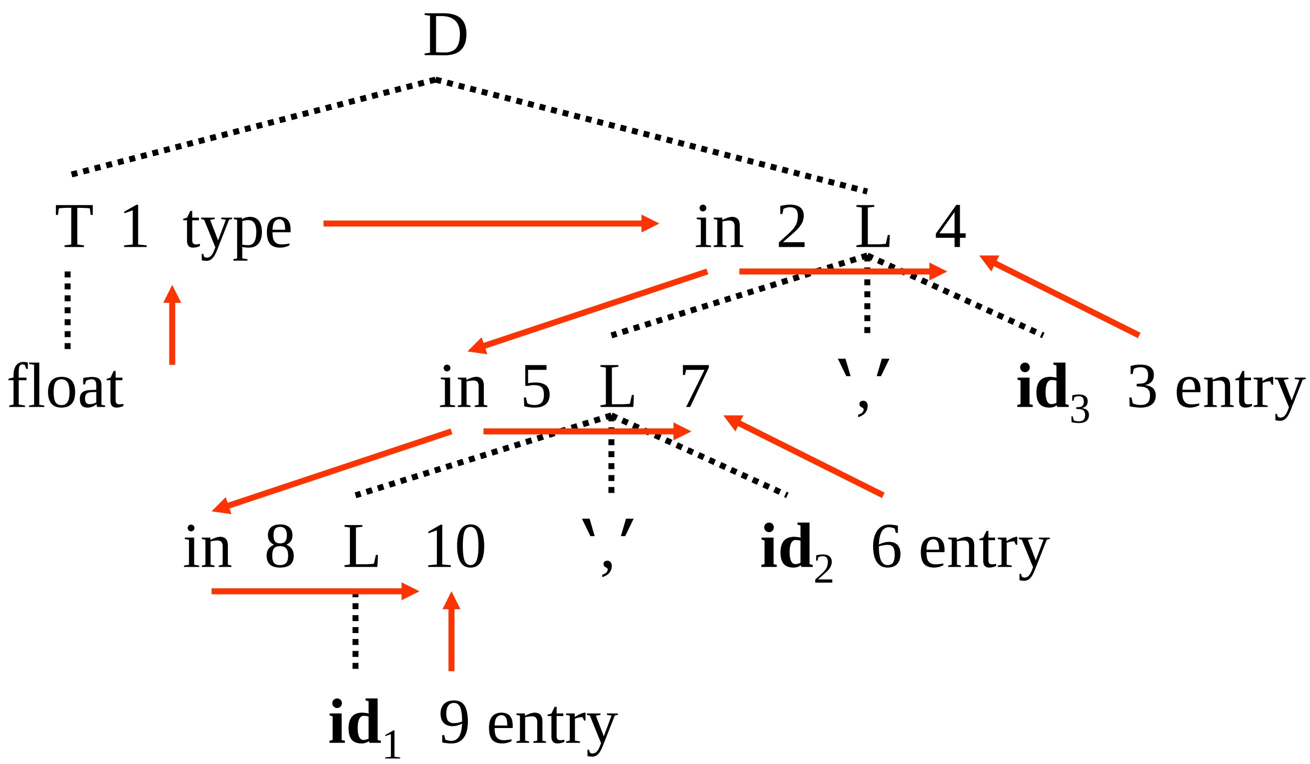
Dependencies of Attributes

- In the semantic rule

  $$b := f(c_1, c_2, \ldots, c_k)$$

  we say b *depends on* $c_1, c_2, \ldots, c_k$
- The semantic rule for b must be evaluated *after* the semantic rules for $c_1, c_2, \ldots, c_k$
- The dependencies of attributes can be represented by a directed graph called *dependency graph*

Dependency Graphs



Evaluation Order

Apply *topological sort* on dependency graph

```
a1 := float
a2 := a1
addtype(a3, a2)      /* a4 */
a5 := a2
addtype(a6, a5)      /* a7 */
a8 := a5
addtype(a9, a8)      /* a10 */
```

```
a1 := float
a2 := a1
a5 := a2
a8 := a5
addtype(a9, a8)      /* a10 */
addtype(a6, a5)      /* a7 */
addtype(a3, a2)      /* a4 */
```

S-Attributed Definitions

- A syntax-directed definition is *S-attributed* if it uses synthesized attributes *exclusively*

Production	Semantic Rules
L → E ‘\n’	print(E.val)
E →E1 ‘+’ T	E.val:=E1.val + T.val
E →T	E.val:=T.val
T →T1 ‘*’ F	T.val:=T1.val * F.val
T →F	T.val:=F.val
F → ‘(‘ E ‘)’	F.val:=E.val
F →digit	F.val:=digit.val

L-Attributed Definitions

- A syntax-directed definition is *L-attributed* if each attribute in each semantic rule for each production

  $$A \rightarrow X_1\ X_2 \dots X_n$$

  is a synthesized attribute, or an inherited attribute of X_j, $1 \leq j \leq n$, depending only on
  1. the attributes of $X_1, X_2, \dots, X_{j\text{-}1}$
  2. the inherited attributes of A

Production	Semantic Rules
D → TL	L.in:=T.type
T → int	T.type:=integer
T → float	T.type:= float
L → L1 ‘,’ id	L1.in:=L.in addtype(id.entry, L.in)
L → id	addtype(id.entry, L.in)

Production	Semantic Rules
$A \longrightarrow LM$	L.i := l(A.i) M.i:= m(L.s) A.s := f(M.s)
$A \longrightarrow QR$	R.i := r(A.i) Q.i := q(R.s) A.s := f(Q.s)

Translation Schemes

- A *translation scheme* is an attribute grammar in which semantic rules are enclosed between braces { and }, and are inserted within the right sides of productions
- The value of an attribute must be *available* when a semantic rule refers to it

An Example

$D \rightarrow T$ {L.in := T.type} L

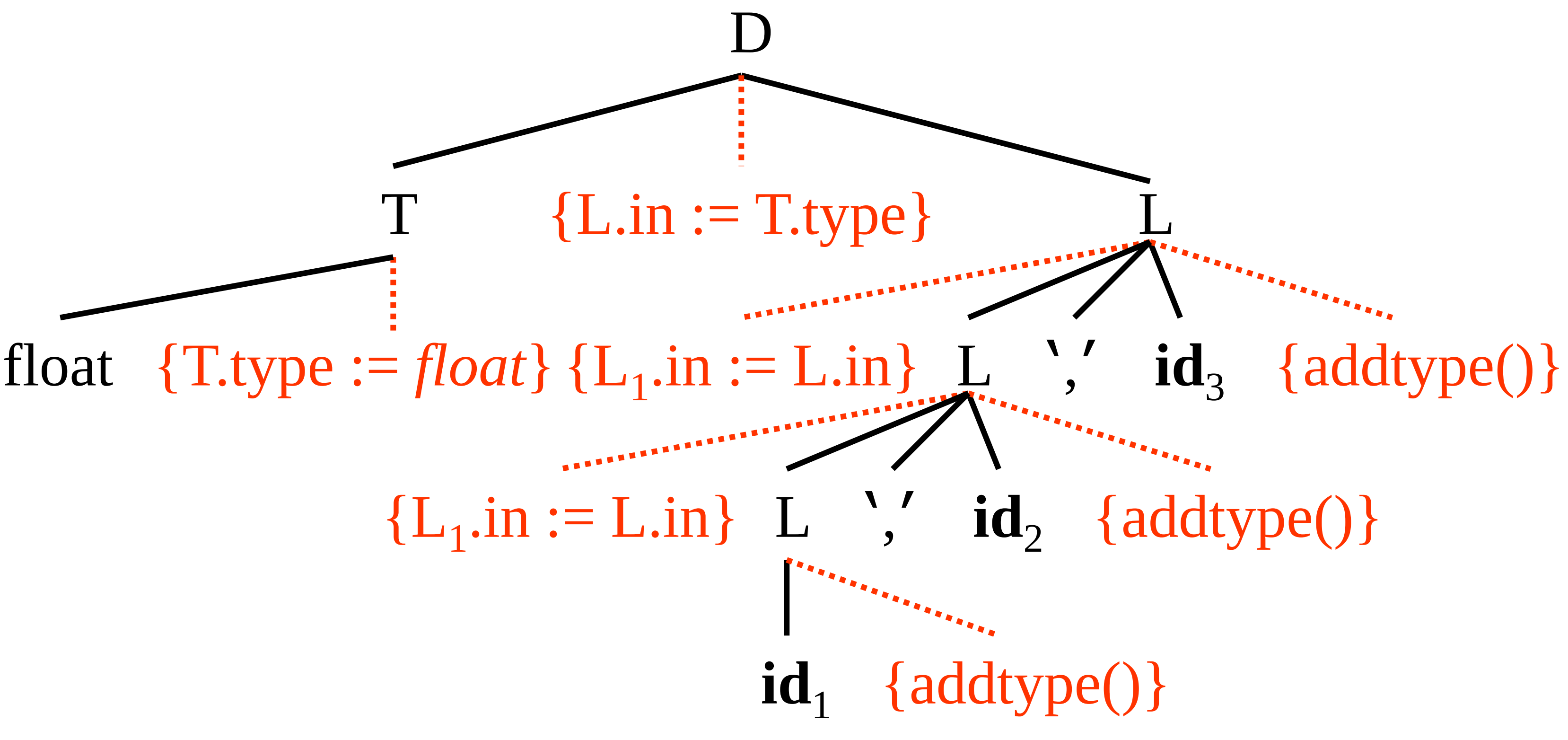
$T \rightarrow$ **int** {T.type := *integer*}

$T \rightarrow$ **float** {T.type := *float*}

$L \rightarrow$ {L_1.in := L.in} L_1 ',' **id** {addtype(**id**.entry, L.in)}

$L \rightarrow$ **id** {addtype(**id**.entry, L.in)}

An Example



D
T
{L.in := T.type}
L
float
{T.type := *float*}
{L_1.in := L.in}
L
','
id$_3$
{addtype()}
{L_1.in := L.in}
L
','
id$_2$
{addtype()}
id$_1$
{addtype()}

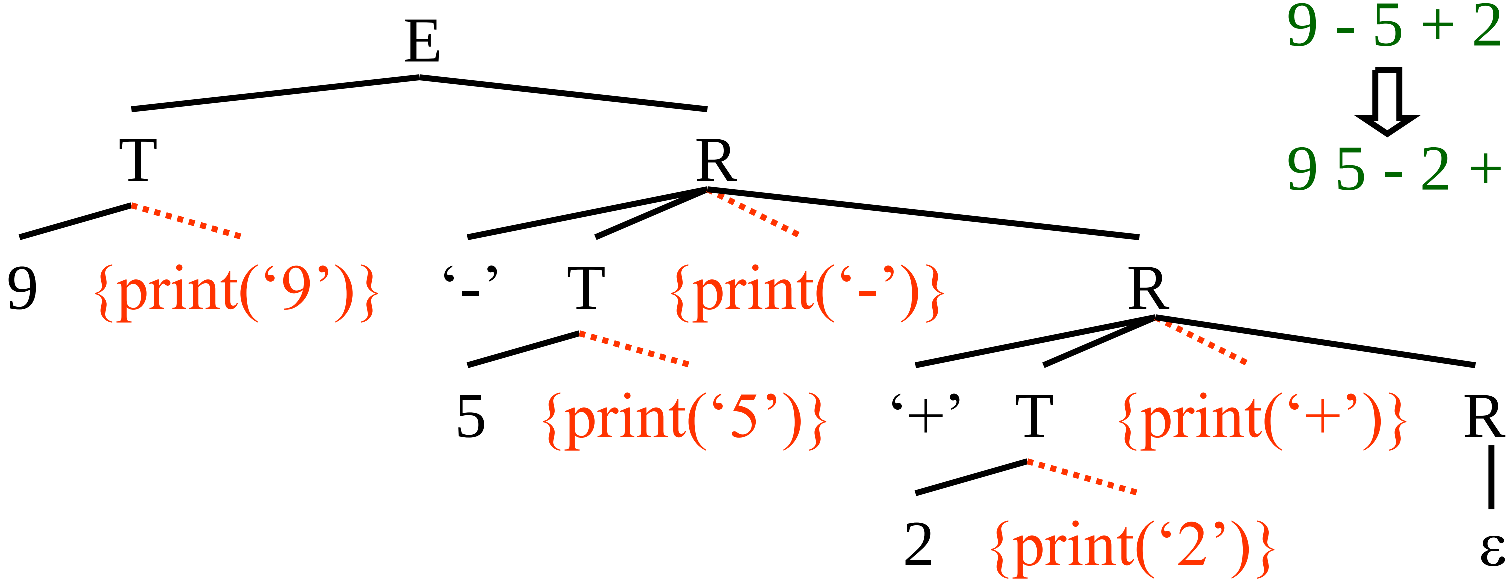
An Example

$E \rightarrow T\ R$

$T \rightarrow$ num {print(num.val)}

$R \rightarrow$ addop T {print(addop.lexeme)} R

$R \rightarrow \varepsilon$



9 - 5 + 2

⇓

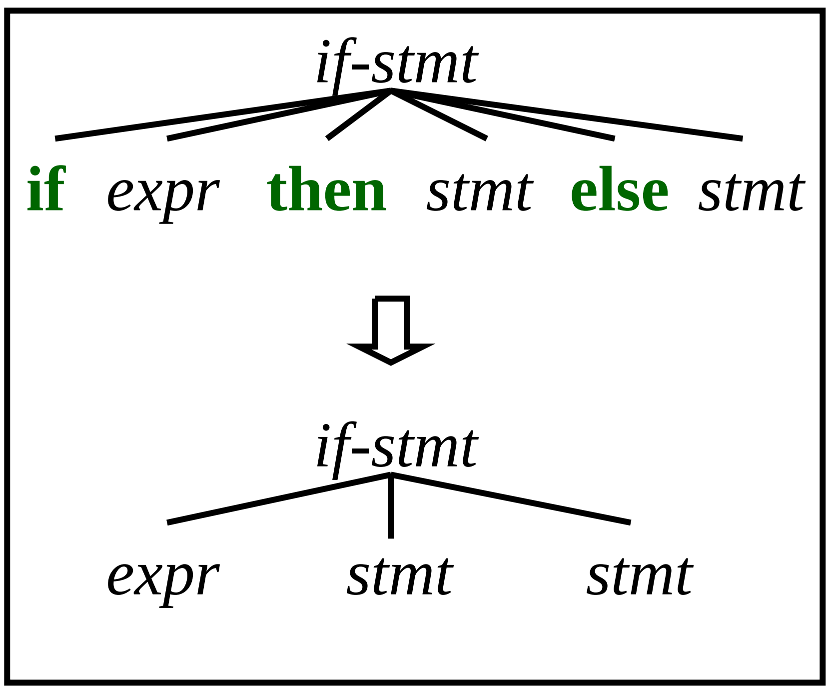
9 5 - 2 +

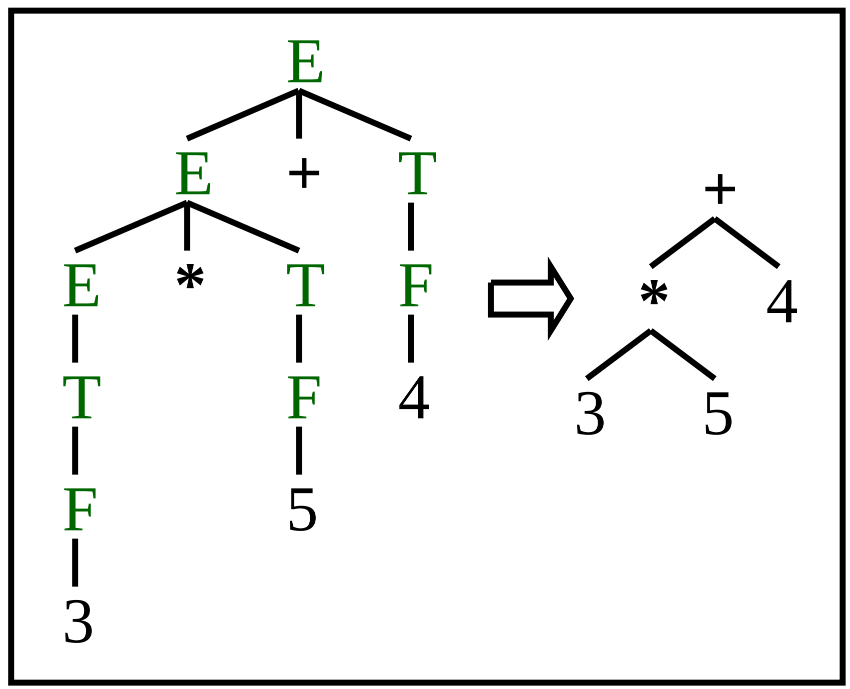
Restrictions on Translation Schemes

- An *inherited attribute* for a symbol on the right side must be computed in a semantic rule before that symbol
- A semantic rule must not refer to a *synthesized attribute* for a symbol to its right
- A *synthesized attribute* for the symbol on the left can be computed after all attributes it depends on have been computed

Construction of Syntax Trees

- An *abstract syntax tree* is a condensed form of *parse tree* useful for representing constructs



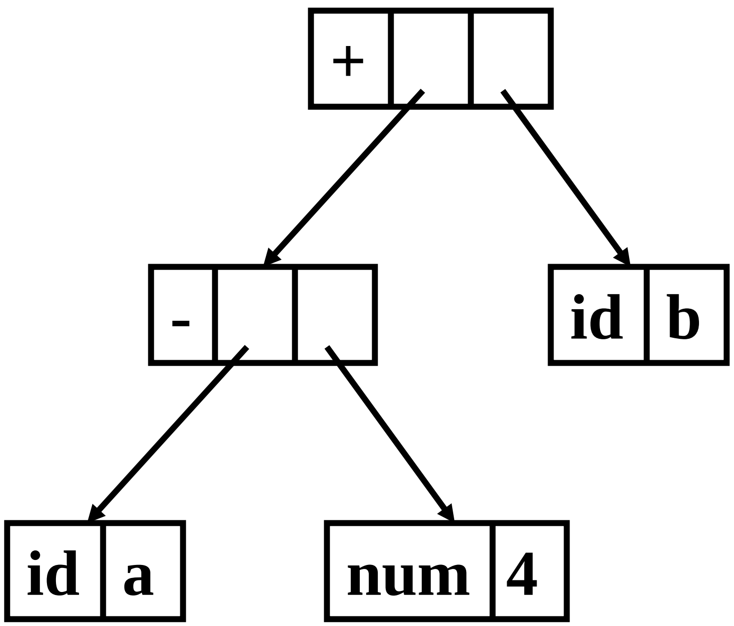


Syntax Trees for Expressions

- Interior nodes are *operators*
- Leaves are *identifiers* or *numbers*
- Functions for constructing nodes
  - mknode(op, left, right)
  - mkleaf(id, entry)
  - mkleaf(num, value)

An Example

a - 4 + b



p1 := mkleaf(id, $entry_a$);
p2 := mkleaf(num, 4);
p3 := mknode('-', p1, p2);
p4 := mkleaf(id, $entry_b$);
p5 := mknode('+', p3, p4);

Top-Down Translators

- For each nonterminal A,
  - inherited attributes $\rightarrow$ formal parameters
  - synthesized attributes $\rightarrow$ returned values
- For each production,
  - for each terminal X with synthesized attribute x,
    save X.x; match(X); advance input;
  - for nonterminal B, c := B(b_1, b_2, …, b_k);
  - for each semantic rule, copy the rule to the parser

An Example

$E \rightarrow T$ { R.i := T.nptr }
$\quad\quad R$ { E.nptr := R.s }

$R \rightarrow$ **addop**
$\quad\quad T$ { R_1.i := mknode(**addop**.lexeme, R.i, T.nptr) }
$\quad\quad R_1$ { R.s := R_1.s }
$R \rightarrow \varepsilon$ { R.s := R.i }

$T \rightarrow$ "(" E ")" { T.nptr := E.nptr }
$T \rightarrow$ **num** { T.nptr := mkleaf(num, **num**.value) }

An Example

```
syntax_tree_node *E(  );
syntax_tree_node *R( syntax_tree_node * );
syntax_tree_node *T(  );

syntax_tree_node *E(  ) {
    syntax_tree_node *enptr, *tnptr, *ri, *rs;
    tnptr = T( );
    ri = tnptr;              /* R.i := T.nptr */
    rs = R(ri);
    enptr = rs;              /* E.nptr := R.s */
    return enptr;
}
```

An Example

```
syntax_tree_node *R(syntax_tree_node * i) {
    syntax_tree_node *nptr, *i1, *s1, *s;
    char addoplexeme;
    if  (lookahead == addop) {
   addoplexeme = lexval;
   match(addop);
   nptr = T();
   i1 = mknode(addoplexeme, i, nptr);
          /* R1.i := mknode(addop.lexeme, R.i, T.nptr) */
   s1 = R(i1);
   s = s1;                    /* R.s := R1.s */
    } else s = i;                        /* R.s := R.i */
    return s;
}
```

An Example

```
syntax_tree_node *T( ) {
    syntax_tree_node *tnptr, *enptr;
    int numvalue;
    if  (lookahead == '(' ) {
        match('(');  enptr = E( );  match(')');
        tnptr = enptr;                    /* T.nptr := E.nptr */
    } else if (lookahead == num ) {
        numvalue = lexval;  match(num);
        tnptr = mkleaf(num, numvalue);
           /* T.nptr := mkleaf(num, num.value) */
    } else error( );
    return tnptr;
}
```

Bottom-Up Translators

- Keep the values of *synthesized attributes* on the parser stack

$A \to X\ Y\ Z$ A.a := f(X.x, Y.y, Z.z);

	symbol	val	
	...	...	
	X	X.x	val[top-2]
	Y	Y.y	val[top-1]
top →	Z	Z.z	val[top]

$A \to X\ Y\ Z$ val[ntop] := f(val[top-2], val[top-1], val[top]);

Evaluation of Synthesized Attributes

- When a token is shifted onto the stack, its attribute value is placed in val[top]
- Code for semantic rules are executed *just before* a reduction takes place
- If the left-hand side symbol has a synthesized attribute, code for semantic rules will place the value of the attribute in val[ntop]

Production	Code Fragment
$L \rightarrow En$	print(val [top])
$E \rightarrow E1 + T$	val[ntop]:=val[top-2] + val[top-1]
$E \rightarrow T$	val[top]:=val[top]
$T \rightarrow T1 * F$	val[ntop]:=val[top-2] * val[top]
$T \rightarrow F$	val[top]:=val[top]
$F \rightarrow (E)$	val[ntop]:=val[top-1]
$F \rightarrow digit$	val[top]:=digit.val

An Example

Input	*symbol*	*val*	*production used*
3*5+4n			
*5+4n	**digit**	3	
*5+4n	F	3	F $\rightarrow$ **digit**
*5+4n	T	3	T $\rightarrow$ F
5+4n	T *	3 _	
+4n	T * **digit**	3 _ 5	
+4n	T * F	3 _ 5	F $\rightarrow$ **digit**
+4n	T	15	T $\rightarrow$ T * F
+4n	E	15	E $\rightarrow$ T

An Example

Input	*symbol*	*val*	*production used*
+4n	E	15	E → T
4n	E +	15 _	
n	E + **digit**	15 _ 4	
n	E + F	15 _ 4	F → **digit**
n	E + T	15 _ 4	T → F
n	E	19	E → E + T
	E n	19 _	
L	_		L → E n

Evaluation of Inherited Attributes

- Removing embedding actions from translation scheme by introducing *marker* nonterminals

$E \rightarrow T\ R$
$R \rightarrow$ "+" T {print('+')} R | "-" T {print('-')} R | ε
$T \rightarrow$ **num** {print(**num**.val)}

$E \rightarrow T\ R$
$R \rightarrow$ "+" $T\ M\ R$ | "-" $T\ N\ R$ | ε
$T \rightarrow$ **num** {print(**num**.val)}
$M \rightarrow \varepsilon$ {print('+')}
$N \rightarrow \varepsilon$ {print('-')}

Evaluation of Inherited Attributes

- Inheriting synthesized attributes on the stack

$$A \rightarrow X \ \{Y.i := X.s\} \ Y$$

symbol	val
...	...
X	X.s
top → Y	

An Example

$D \rightarrow T$ {L.in := T.type}

L

$T \rightarrow$ **int**   {T.type := *integer*}

$T \rightarrow$ **float**   {T.type := *float*}

$L \rightarrow$ {L_1.in := L.in}

L_1 ‘,’ **id**   {addtype(**id**.entry, L.in)}

$L \rightarrow$ **id**   {addtype(**id**.entry, L.in)}

$D \rightarrow T\ L$

$T \rightarrow$ **int**   {val[ntop] := *integer*}

$T \rightarrow$ **float**   {val[ntop] := *float*}

$L \rightarrow L_1$ ‘,’ **id**   {addtype(val[top], val[top-3])}

$L \rightarrow$ **id**   {addtype(val[top], val[top-1])}

An Example

Input	*symbol*	*val*	*production used*
int p,q,r			
p,q,r	**int**	_	
p,q,r	T	*i*	T → **int**
,q,r	T **id**	*i* *e*	
,q,r	T L	*i* _	L → **id**
q,r	T L ,	*i* _ _	
,r	T L , **id**	*i* _ _ *e*	
,r	T L	*i* _	L → L "," **id**
r	T L ,	*i* _ _	
	T L , **id**	*i* _ _ *e*	
	T L	*i* _	L → L "," **id**
	D		

Evaluation of Inherited Attributes

- Simulating the evaluation of inherited attributes

  Inheriting the value of a synthesized attribute works only if the grammar allows the position of the attribute value to be predicted

An Example

$S \rightarrow a\ A\ C$ $\{C.i := A.s\}$
$S \rightarrow b\ A\ B\ C$ $\{C.i := A.s\}$
$C \rightarrow c$ $\{C.s := g(C.i)\}$

$S \rightarrow a\ A\ C$ $\{C.i := A.s\}$
$S \rightarrow b\ A\ B\ M\ C$ $\{M.i := A.s; C.i := M.s\}$
$C \rightarrow c$ $\{C.s := g(C.i)\}$
$M \rightarrow \varepsilon$ $\{M.s := M.i\}$

Another Example

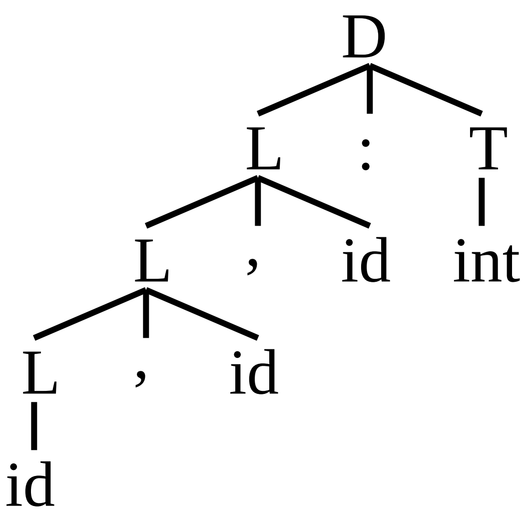
$S \to a\ A\ C$ $\{C.i := f(A.s)\}$

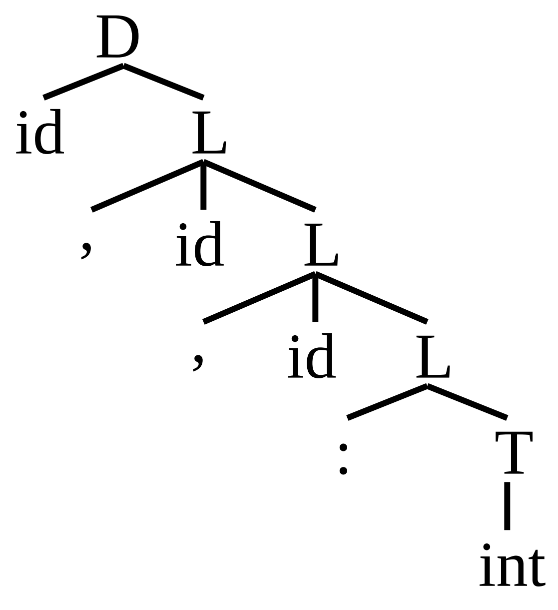
$S \to a\ A\ N\ C$ $\{N.i := A.s;\ C.i := N.s\}$
$N \to \varepsilon$ $\{N.s := f(N.i)\}$

From Inherited to Synthesized

D → L ":" T
L → L "," **id** | **id**
T → **integer** | **char**

D → **id** L
L → "," **id** L | ":" T
T → **integer** | **char**





Bison

```
%token DIGIT
%%
line :  expr '\n'  {printf("%d\n", $1);}
        ;
expr:  expr '+' term  {$$ = $1 + $3;}
        |  term
        ;
term:  term '*' factor  {$$ = $1 * $3;}
        |  factor
        ;
factor: '(' expr ')'  {$$ = $2;}
        |  DIGIT
        ;
```

Bison

```
%union {
        char op_type;   int value;
   }
%token <value> DIGIT
%type  <op_type>  op
%type  <value>  expr  factor
%%
expr: expr op factor {$$ = $2 == ‘+’ ? $1 + $3 : $1 - $3;}
       | factor
       ;
op: + {$$ = ‘+’;}
   |  - {$$ = ‘-’;}
   ;
factor: DIGIT ;
```

Recursive Evaluators

- The parser constructs a parse tree *explicitly*
- A *recursive evaluator* is a function that traverses the parse tree and evaluates attributes
- A recursive evaluator can traverse the parse tree *in any order*
- A recursive evaluator can traverse the parse tree *multiple times*

An Example

$S \rightarrow \quad \{\ B.ps := 10\ \}$

$\quad B \quad \{\ S.ht := B.ht\ \}$

$B \rightarrow \quad \{\ B_1.ps := B.ps\ \}$

$\quad B_1 \quad \{\ B_2.ps := B.ps\ \}$

$\quad B_2 \quad \{\ B.ht := \max(B_1.ht,\ B_2.ht)\ \}$

$B \rightarrow \quad \{\ B_1.ps := B.ps\ \}$

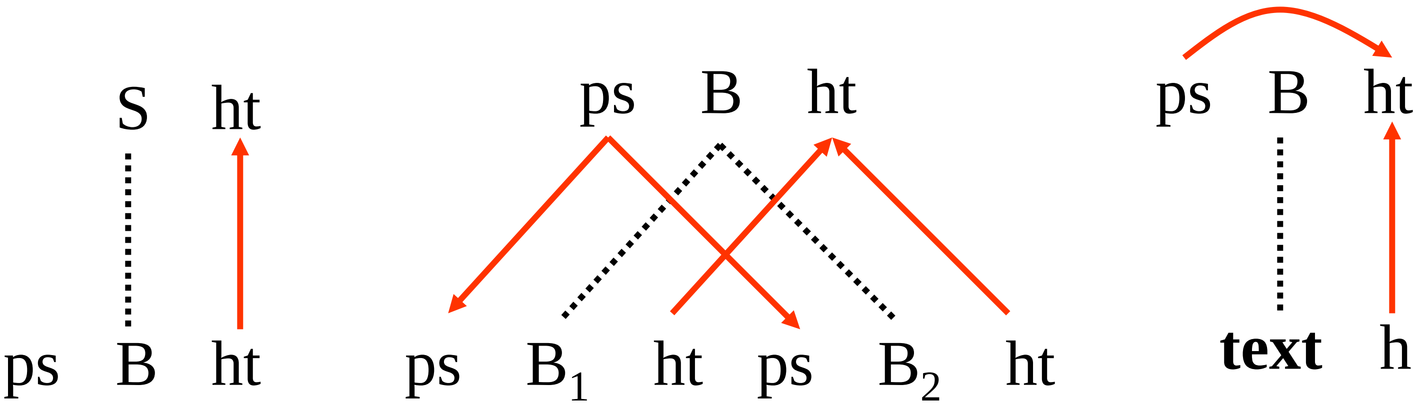
$\quad B_1$

$\quad \mathbf{sub} \quad \{\ B_2.ps := \mathrm{shrink}(B.ps)\ \}$

$\quad B_2 \quad \{\ B.ht := \mathrm{disp}(B_1.ht,\ B_2.ht)\ \}$

$B \rightarrow \mathbf{text} \quad \{\ B.ht := \mathbf{text}.h \times B.ps\ \}$

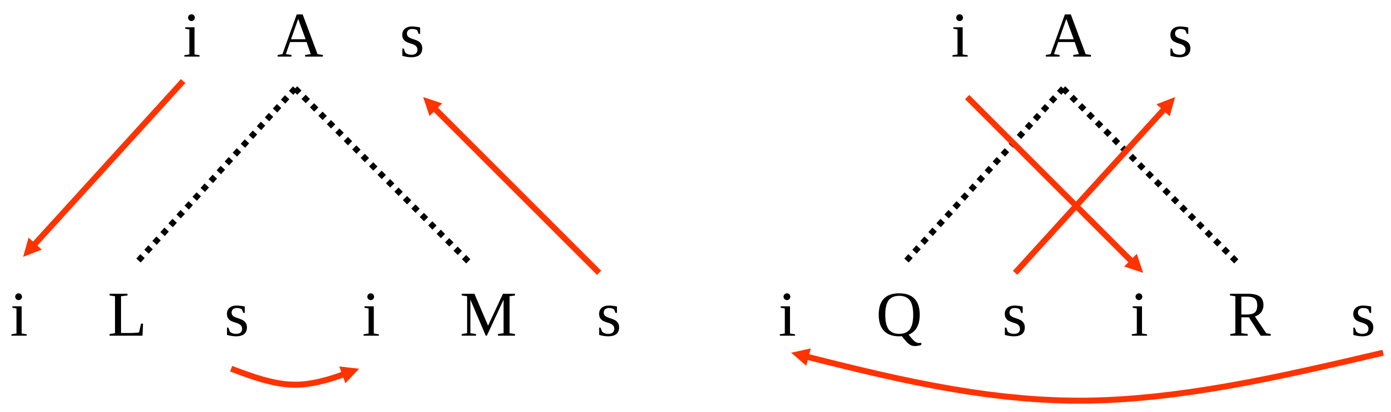
An Example



An Example

```
function B(n, ps);
    var  ps1, ps2, ht1, ht2;
begin
    case  production at node n  of
    'B → B1 B2':
        ps1 :=  ps;  ht1 :=  B(child(n, 1), ps1);  ps2 :=  ps;
        ht2 :=  B(child(n, 2), ps2);  return max(ht1, ht2);
    'B → B1 sub B2':
        ps1 :=  ps;  ht1 :=  B(child(n, 1), ps1);  ps2 :=  shrink(ps);
        ht2 :=  B(child(n, 3), ps2);  return disp(ht1, ht2);
    'B → text':
        return ps  ×  text.h;
    default:  error  end
end;
```

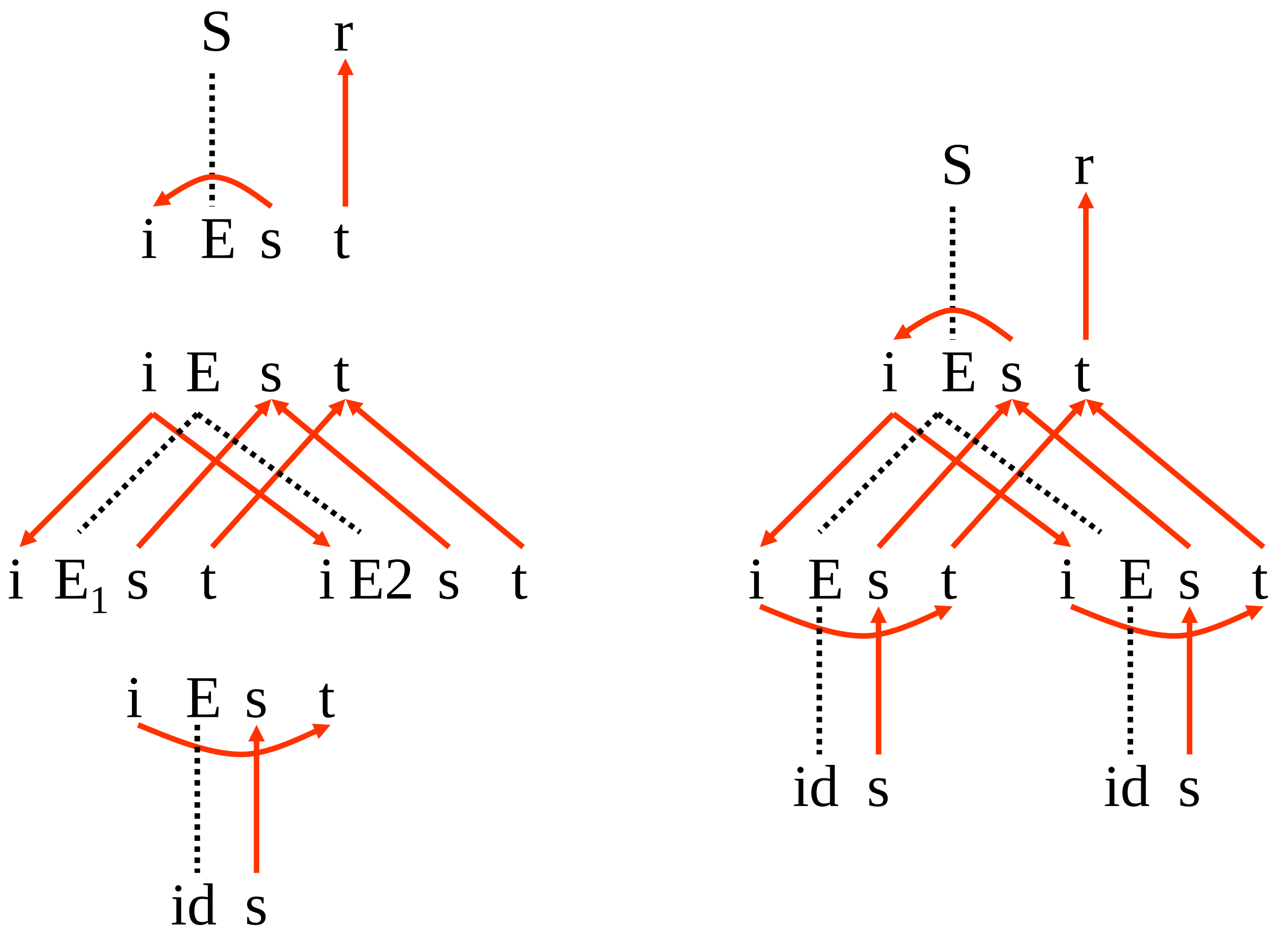
An Example



An Example

```
function  A(n, ai);
   var  li, ls, mi, ms, ri, rs, qi, qs;
begin
   case  production at node n  of
   'A → L  M':
      li := l(ai);  ls :=  L(child(n, 1), li);  mi :=  m(ls);
      ms :=  M(child(n, 2), mi);  return f(ms);
   'A → Q  R':
      ri := r(ai);  rs :=  R(child(n, 2), ri);  qi :=  q(rs);
      qs :=  Q(child(n, 1), qi);  return f(qs);
   default:  error
   end
end;
```

An Example



An Example

```
function  Es(n);
   var  s1, s2;
begin
   case  production at node n  of
   'E → E1 E2':
      s1 :=  Es(child(n, 1));  s2 :=  Es(child(n, 2));
      return  fs(s1, s2);
   'E → id':
      return  id.s;
   default:  error
   end
end;
```

An Example

```
function  Et(n, i);
    var  i1, t1, i2, t2;
begin
    case  production at node n  of
    'E → E1 E2':
        i1 := fi1(i);  t1 :=  Et(child(n, 1), i1);
        i2 := fi2(i);  t2 :=  Et(child(n, 2), i2);
        return  ft(t1, t2);
    'E → id':
        return  h(i);
    default:  error
    end
end;
```

An Example

```
function  Sr(n);
    var  s, i, t;
begin
    s :=  Es(child(n, 1));
    i :=  g(s);
    t :=  Et(child(n, 1), i);
    return  t
end;
```

Type Systems

- A *type system* is a collection of rules for assigning types to the various parts of a program
- A *type checker* implements a type system
- Types are represented by *type expressions*

Type Expressions

- A *basic type* is a type expression
  - boolean, char, integer, real, void, type_error
- A *type constructor* applied to type expressions is a type expression
  - array: array(I, T)
  - product: $T_1 \times T_2$
  - record: $record((N_1 \times T_1) \times (N_2 \times T_2))$
  - pointer: pointer(T)
  - function: $D \rightarrow R$

Type Declarations

$P \rightarrow$ D ";" E

$D \rightarrow$ D ";" D

| **id** ":" T { addtype(**id**.entry, T.type) }

$T \rightarrow$ **char** { T.type := char }

$T \rightarrow$ **integer** { T.type := integer }

$T \rightarrow$ "*" T_1 {T.type := pointer(T_1.type) }

$T \rightarrow$ **array** "[" **num** "]" **of** T_1

{ T.type := array(**num**.value, T_1.type) }

Type Checking of Expressions

$E \rightarrow$ **literal** {E.type := char}

$E \rightarrow$ **num** {E.type := int}

$E \rightarrow$ **id** {E.type := lookup(**id**.entry)}

$E \rightarrow E_1$ **mod** E_2

{E.type := if E_1.type = int and E_2.type = int
then int else type_error}

$E \rightarrow E_1$ "[" E_2 "]"

{E.type := if E_1.type = array(s, t) and E_2.type = int
then t else type_error}

$E \rightarrow$ "*" E_1

{E.type := if E_1.type = pointer(t)
then t else type_error}

Type Checking of Statements

$P \rightarrow D$ ";" S

$S \rightarrow$ **id** ":=" E

{S.type := if lookup(**id**.entry) = E.type
then void else type_error}

$S \rightarrow$ **if** E **then** S_1

{S.type := if E.type = boolean
then S_1.type else type_error}

$S \rightarrow$ **while** E **do** S_1

{S.type := if E.type = boolean
then S_1.type else type_error}

$S \rightarrow S_1$ ";" S_2

{S.type := if S_1.type = void and S_2.type = void
then void else type_error}

Type Checking of Functions

$T \to T_1$ "$\to$" T_2

$\{T.type := T_1.type \;\to\; T_2\,.type\}$

$E \to E_1$ "(" E_2 ")"

$\{E.type :=$ if $E_1.type = s \to t$ and $E_2.type = s$

then t else type_error}

It says that in an expression formed by applying E_1 to E_2 , the type of E_1 must be a function $s \to t$ from the type s of E_2 to some range type t; the type of $E_1(E_2)$ is t.