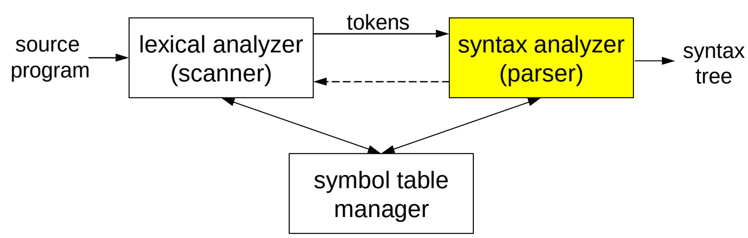
CSc 453
Syntax Analysis
(Parsing)

Saumya Debray
The University of Arizona
Tucson

Overview



Main Task: Take a token sequence from the scanner and verify that it is a syntactically correct program.

Secondary Tasks:

- Process declarations and set up symbol table information accordingly, in preparation for semantic analysis.
- Construct a syntax tree in preparation for intermediate code generation.

Context-free Grammars

- A *context-free grammar* for a language specifies the syntactic structure of programs in that language.
- Components of a grammar:
  - a finite set of tokens (obtained from the scanner);
  - a set of variables representing "related" sets of strings, e.g., *declarations*, *statements*, expressions.
  - a set of rules that show the structure of these strings.
  - an indication of the "top-level" set of strings we care about.

Context-free Grammars: Definition

Formally, a context-free grammar *G* is a 4-tuple *G* = (V, T, P, S), where:

- V is a finite set of *variables* (or *nonterminals*). These describe sets of "related" strings.
- T is a finite set of *terminals* (i.e., tokens).
- P is a finite set of *productions*, each of the form

  $A \rightarrow \alpha$

  where $A \in$ V is a variable, and $\alpha \in (V \cup T)^*$ is a sequence of terminals and nonterminals.
- S $\in$ V is the *start symbol*.

Context-free Grammars: An Example

A grammar for palindromic bit-strings:

G = (V, T, P, S), where:

- V = { S, B }
- T = {0, 1}
- P = { $S \to B$,
  $S \to \varepsilon$,
  $S \to 0\ S\ 0$,
  $S \to 1\ S\ 1$,
  $B \to 0$,
  $B \to 1$
  }

Context-free Grammars: Terminology

- *Derivation*: Suppose that
  - α and β are strings of grammar symbols, and
  - $A \to \gamma$ is a production.

  Then, $\alpha A \beta \Rightarrow \alpha\gamma\beta$ ("$\alpha A \beta$ *derives* $\alpha\gamma\beta$").

- $\Rightarrow$: "derives in one step"

  $\Rightarrow^*$: "derives in 0 or more steps"

  $\alpha \Rightarrow^* \alpha$ (0 steps)

  $\alpha \Rightarrow^* \beta$ if $\alpha \Rightarrow \gamma$ and $\gamma \Rightarrow^* \beta$ ($\geq$ 1 steps)

Derivations: Example

- Grammar for palindromes: G = (V, T, P, S),
  - V = {S},
  - T = {0, 1},
  - P = { S → 0 S 0 | 1 S 1 | 0 | 1 | ε }.
- A derivation of the string 10101:

S
⇒ 1 S 1 (using S → 1S1)
⇒ 1 0S0 1 (using S → 0S0)
⇒ 10101 (using S → 1)

Leftmost and Rightmost Derivations

- A *leftmost derivation* is one where, at each step, the leftmost nonterminal is replaced.
  (analogous for *rightmost derivation*)
- *Example*: a grammar for arithmetic expressions:
  E → E + E | E * E | **id**
  - *Leftmost derivation*:
    E ⇒ E * E ⇒ E + E * E ⇒ **id** + E * E ⇒ **id** + **id** * E ⇒ **id** + **id** * **id**
  - *Rightmost derivation*:
    E ⇒ E + E ⇒ E + E * E ⇒ E + E * **id** ⇒ E + **id** * **id** ⇒ **id** + **id** * **id**

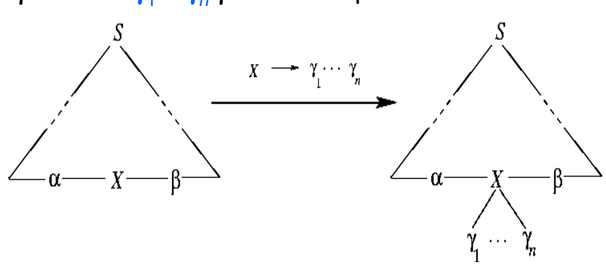
Context-free Grammars: Terminology

- The *language* of a grammar $G = (V,T,P,S)$ is
  $L(G) = \{ w \mid w \in T^* \text{ and } S \Rightarrow^* w \}$.
  The language of a grammar contains only strings of terminal symbols.

- Two grammars G_1 and G_2 are *equivalent* if
  $L(G_1) = L(G_2)$.

Parse Trees

- A *parse tree* is a tree representation of a derivation.
- Constructing a parse tree:
  - The root is the start symbol S of the grammar.
  - Given a parse tree for $\alpha X \beta$, if the next derivation step is
    $\alpha X \beta \Rightarrow \alpha \gamma_1 \ldots \gamma_n \beta$ then the parse tree is obtained as:



Approaches to Parsing

- *Top-down parsing*:
  - attempts to figure out the derivation for the input string, starting from the start symbol.
- *Bottom-up parsing*:
  - starting with the input string, attempts to "derive in reverse" and end up with the start symbol;
  - forms the basis for parsers obtained from parser-generator tools such as yacc, bison.

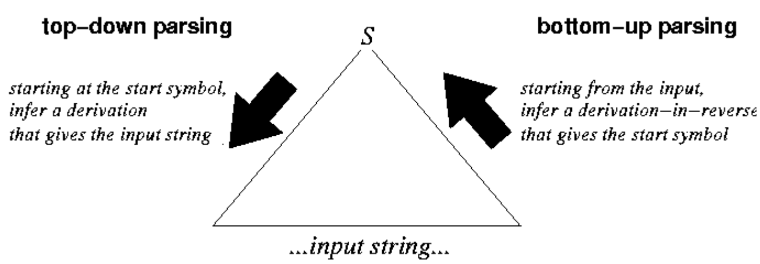
Top-down Parsing

- "top-down:" starting with the start symbol of the grammar, try to derive the input string.
- *Parsing process*: use the current state of the parser, and the next input token, to guide the derivation process.
- *Implementation*: use a finite state automaton augmented with a runtime stack ("*pushdown automaton*").

Bottom-up Parsing

- "bottom-up:" work backwards from the input string to obtain a derivation for it.
- *Parsing process*: use the parser state to keep track of:
  - what has been seen so far, and
  - given this, what the rest of the input might look like.
- *Implementation*: use a finite state automaton augmented with a runtime stack ("*pushdown automaton*").

Parsing: Top-down vs. Bottom-up



Parsing Problems: Ambiguity

- A grammar G is *ambiguous* if some string in L(G) has more than one parse tree.
- Equivalently: if some string in L(G) has more than one leftmost (rightmost) derivation.
- *Example*: The grammar

  $E \rightarrow E + E \mid E * E \mid \mathbf{id}$

  is ambiguous, since "id+id*id" has multiple parses:

Dealing with Ambiguity

1. Transform the grammar to an equivalent unambiguous grammar.
2. Use *disambiguating rules* along with the ambiguous grammar to specify which parse to use.

Comment: It is not possible to determine algorithmically whether:

- Two given CFGs are equivalent;
- A given CFG is ambiguous.

Removing Ambiguity: Operators

- *Basic idea*: use additional nonterminals to enforce associativity and precedence:
  - Use one nonterminal for each precedence level:
    - E → E * E | E + E | id

    needs 2 nonterminals (2 levels of precedence).
  - Modify productions so that "lower precedence" nonterminal is in direction of precedence:

    E → E + E ⇒ E → E + T (+ is left-associative)

Example

- *Original grammar*:

  E → E * E | E / E | E + E | E – E | (E) | **id**

  precedence levels: { *, / } > { +, – }

  associativity: *, /, +, – are all left-associative.

- *Transformed grammar*:

  E → E + T | E – T | T (precedence level for: +, -)

  T → T * F | T / F | F (precedence level for: *, /)

  F → (E) | **id**

Bottom-up parsing: Approach

1. Preprocess the grammar to compute some info about it.
   (FIRST and FOLLOW sets)
2. Use this info to construct a pushdown automaton for the grammar:
   - the automaton uses a table ("parsing table") to guide its actions;
   - constructing a parser amounts to constructing this table.

FIRST Sets

Defn: For any string of grammar symbols α,

- FIRST(α) = { **a** | **a** is a terminal and $\alpha \Rightarrow^* \mathbf{a}\beta$}.
- if $\alpha \Rightarrow^* \varepsilon$ then ε is also in FIRST(α).

- *Example*: E → T E′
  E′ → + T E′ | ε
  T → F T′
  T′ → * F T′ | ε
  F → (E) | **id**

FIRST(E) = FIRST(T) = FIRST(F) = { (, **id** }
FIRST(E′) = { +, ε }
FIRST(T′) = { *, ε }

Computing FIRST Sets

Given a sequence of grammar symbols A:

- **if** A is a terminal or $A = \varepsilon$ **then** FIRST(A) = {A}.
- **if** A is a nonterminal with productions $A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$ **then**:
  - FIRST(A) = FIRST(α_1) $\cup \ldots \cup$ FIRST(α_n).
- **if** A is a sequence of symbols $Y_1 \ldots Y_k$ **then**:
  - **for** i = 1 to k **do**:
    - add each $a \in$ FIRST(Y_i), such that $a \neq \varepsilon$, to FIRST(A).
    - **if** $\varepsilon \notin$ FIRST(Y_i) **then** break;
  - **if** ε is in each of FIRST(Y_1), …, FIRST(Y_k) **then** add ε to FIRST(A).

Computing FIRST sets: cont'd

- For each nonterminal A in the grammar, initialize FIRST(A) = $\emptyset$.
- **repeat** {

  for each nonterminal A in the grammar {

  compute FIRST(A); /* as described previously */

  }

  } **until** there is no change to any FIRST set.

Example (FIRST Sets)

$X \rightarrow YZ \mid$ **a**
$Y \rightarrow$ **b** $\mid \varepsilon$
$Z \rightarrow$ **c** $\mid \varepsilon$

- $X \rightarrow$ **a**, so add **a** to FIRST(X).
- $X \rightarrow YZ$, **b** $\in$ FIRST(Y), so add **b** to FIRST(X).
- $Y \rightarrow \varepsilon$, i.e. $\varepsilon \in$ FIRST(Y), so add non-ε symbols from FIRST(Z) to FIRST(X).
  - ► add **c** to FIRST(X).
- $\varepsilon \in$ FIRST(Y) and $\varepsilon \in$ FIRST(Z), so add ε to FIRST(X).

Final: FIRST(X) = { **a**, **b**, **c**, ε }.

FOLLOW Sets

Definition: Given a grammar G = (V, T, P, S), for any nonterminal $A \in$ V:

- FOLLOW(A) = { **a** $\in$ T | S $\Rightarrow^*$ αA**a**β for some α, β}.

  i.e., FOLLOW(A) contains those terminals that can appear after A in something derivable from the start symbol S.
- if S $\Rightarrow^*$ αA then \$ is also in FOLLOW(A). (\$ $\equiv$ EOF, "end of input.")

Example:

$E \rightarrow E + E \mid$ **id**
FOLLOW(E) = { +, \$ }.

Computing FOLLOW Sets

Given a grammar G = (V, T, P, S):

1. add \$ to FOLLOW(S);
2. **repeat** {
   - for each production $A \rightarrow \alpha B \beta$ in P, add every non-ε symbol in FIRST(β) to FOLLOW(*B*).
   - for each production $A \rightarrow \alpha B \beta$ in P, where $\varepsilon \in$ FIRST(β), add everything in FOLLOW(*A*) to FOLLOW(*B*).
   - for each production $A \rightarrow \alpha B$ in P, add everything in FOLLOW(*A*) to FOLLOW(*B*).

} **until** no change to any FOLLOW set.

Example (FOLLOW Sets)

X → YZ | **a**
Y → **b** | ε
Z → **c** | ε

- X is start symbol: add \$ to FOLLOW(X);
- X → YZ, so add everything in FOLLOW(X) to FOLLOW(Z).
  ► add \$ to FOLLOW(Z).
- X → YZ, so add every non-ε symbol in FIRST(Z) to FOLLOW(Y).
  ► add **c** to FOLLOW(Y).
- X → YZ and ε ∈ FIRST(Z), so add everything in FOLLOW(X) to FOLLOW(Y).
  ► add \$ to FOLLOW(Y).

Shift-reduce Parsing

- An instance of bottom-up parsing
- *Basic idea*: repeat
  1. in the string being processed, find a substring α such that $A \rightarrow \alpha$ is a production;
  2. replace the substring α by A (i.e., reverse a derivation step).

  until we get the start symbol.
- *Technical issues*: Figuring out
  1. which substring to replace; and
  2. which production to reduce with.

Shift-reduce Parsing: Example

Grammar: $S \rightarrow \mathbf{a}AB\mathbf{e}$

$A \rightarrow A\mathbf{bc} \mid \mathbf{b}$

$B \rightarrow \mathbf{d}$

Input: **abbcde** (using $A \rightarrow \mathbf{b}$)

⇒ **a***A***bcde** (using $A \rightarrow A\mathbf{bc}$)

⇒ **a***A***de** (using $B \rightarrow \mathbf{d}$)

⇒ **a***AB***e** (using $S \rightarrow \mathbf{a}AB\mathbf{e}$)

⇒ *S*

Shift-Reduce Parsing: cont'd

- Need to choose reductions carefully:

  abbcde ⇒ **a*A*bcde** ⇒ **a*A*bc*B*e** ⇒ …

  doesn't work.
- A *handle* of a string *s* is a substring α s.t.:
  - α matches the RHS of a rule $A \to \alpha$; and
  - replacing α by *A* (the LHS of the rule) represents a step in the reverse of a rightmost derivation of *s*.
- For shift-reduce parsing, reduce only handles.

Shift-reduce Parsing: Implementation

- *Data Structures*:
  - a stack, its bottom marked by '**$**'. Initially empty.
  - the input string, its right end marked by '**$**'. Initially w.
- *Actions*:

  repeat
  1. *Shift* some (≥ 0) symbols from the input string onto the stack, until a handle β appears on top of the stack.
  2. *Reduce* β to the LHS of the appropriate production.

  until ready to accept.
  - *Acceptance*: when input is empty and stack contains only the start symbol.

Example

Stack (→)	*Input*	*Action*
$	**abbcde$**	shift
$a	**bbcde$**	shift
$ab	**bcde$**	reduce: *A* → **b**
$a*A*	**bcde$**	shift
$a*A***b**	**cde$**	shift
$a*A***bc**	**de$**	reduce: *A* → *A***bc**
$a*A*	**de$**	shift
$a*A***d**	**e$**	reduce: *B* → **d**
$a*AB*	**e$**	shift
$a*AB***e**	**$**	reduce: *S* → **a***AB***e**
$*S*	**$**	accept

Grammar :

S → **a***AB***e**

A → *A***bc** | **b**

B → **d**

Conflicts

- Can't decide whether to shift or to reduce — both seem OK ("*shift-reduce conflict*").

  Example: *S* → **if** *E* **then** *S* | **if** *E* **then** *S* **else** *S* | …

- Can't decide which production to reduce with — several may fit ("*reduce-reduce conflict*").

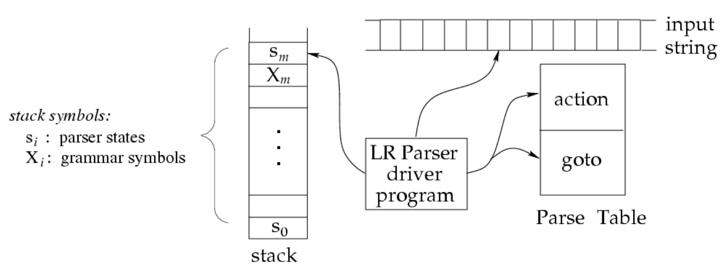
  Example: *Stmt* → **id (** *args* **)** | *Expr*

  Expr → **id (** *args* **)**

LR Parsing

- A kind of shift-reduce parsing. An LR(k) parser:
  - scans the input L-to-R;
  - produces a Rightmost derivation (in reverse); and
  - uses k tokens of lookahead.
- Advantages:
  - very general and flexible, and handles a wide class of grammars;
  - efficiently implementable.
- Disadvantages:
  - difficult to implement by hand (use tools such as ***yacc*** or ***bison***).

LR Parsing: Schematic



- The driver program is the same for all LR parsers (SLR(1), LALR(1), LR(1), …). Only the parse table changes.
- Different LR parsing algorithms involve different tradeoffs between parsing power, parse table size.

LR Parsing: the parser stack

- The parser stack holds strings of the form

  $s_0\, X_1 s_1\, X_2 s_2\, \dots\, X_m s_m$ (s_m is on top)

  where s_i are parser states and X_i are grammar symbols.

  (**Note**: the X_i and s_i always come in pairs, with the state component s_i on top.)

- A parser configuration is a pair

  ⟨stack contents, unexpended input⟩

LR Parsing: Roadmap

- LR parsing algorithm:
  - parse table structure
  - parsing actions
- Parse table construction:
  - viable prefix automaton
  - parse table construction from this automaton
  - improving parsing power: different LR parsing algorithms

LR Parse Tables

- The parse table has two parts: the **action** function and the **goto** function.
- At each point, the parser's next move is given by **action**[s_m, $\mathbf{a}_i$], where:
  - s_m is the state on top of the parser stack, and
  - $\mathbf{a}_i$ the next input token.
- The **goto** function is used only during *reduce* moves.

LR Parser Actions: shift

- Suppose:
  - the parser configuration is $\langle s_0\ X_1 s_1 \ldots X_m s_m,\ \ \mathbf{a}_i \ldots \mathbf{a}_n \rangle$, and
  - **action**[s_m, $\mathbf{a}_i$] = '*shift* s_n'.
- Effects of shift move:
  1. push the next input symbol $\mathbf{a}_i$; and
  2. push the state s_n
- New configuration: $\langle s_0\ X_1 s_1 \ldots X_m s_m\ \mathbf{a}_i\ s_n,\ \ \mathbf{a}_{i+1} \ldots \mathbf{a}_n \rangle$

LR Parser Actions: reduce

- Suppose:
  - the parser configuration is $\langle s_0\, X_1 s_1 \ldots X_m s_m,\quad \mathbf{a}_i \ldots \mathbf{a}_n \rangle$, and
  - **action**[s_m, $\mathbf{a}_i$] = '*reduce* $A \rightarrow \beta$'.
- Effects of reduce move:
  1. pop n states and n grammar symbols off the stack ($2n$ symbols total), where $n = |\beta|$.
  2. suppose the (newly uncovered) state on top of the stack is t, and **goto**[t, A] = u.
  3. push A, then u.
- New configuration: $\langle s_0\, X_1 s_1 \ldots X_{m-n} s_{m-n}\, \boldsymbol{A}\, \boldsymbol{u},\quad \mathbf{a}_i \ldots \mathbf{a}_n \rangle$

LR Parsing Algorithm

1. set `ip` to the start of the input string w**$**.
2. **while** TRUE **do**:
   1. let s = state on top of parser stack, **a** = input symbol pointed at by `ip`.
   2. if **action**[s,**a**] == '*shift* t' then: (i) push the input symbol **a** on the stack, then the state t; (ii) advance `ip`.
   3. if **action**[s,**a**] == '*reduce* $A \rightarrow \beta$' then: (i) pop 2*|β| symbols off the stack; (ii) suppose t is the state that now gets uncovered on the stack; (iii) push the LHS grammar symbol A and the state u = **goto**[A, t].
   4. if **action**[s,**a**] == '*accept*' then accept;
   5. else signal a syntax error.

LR parsing: Viable Prefixes

- ***Goal***: to be able to identify handles, and so produce a rightmost derivation in reverse.
- Given a configuration $\langle s_0\ X_1 s_1 \ldots X_m s_m,\ \mathbf{a}_i \ldots \mathbf{a}_n \rangle$:
  - $X_1\ X_2 \ldots X_m\ \mathbf{a}_i \ldots \mathbf{a}_n$ is obtainable on a rightmost derivation.
  - $X_1\ X_2 \ldots X_m$ is called a *viable prefix*.
- The set of viable prefixes of a grammar are recognizable using a finite automaton.

  This automaton is used to recognize handles.

Viable Prefix Automata

- An LR(0) item of a grammar G is a production of G with a dot "•" somewhere in the RHS.
  - *Example*: The rule $A \rightarrow \mathbf{a}\ A\ \mathbf{b}$ gives these LR(0) items:
    - $A \rightarrow \bullet\ \mathbf{a}\ A\ \mathbf{b}$
    - $A \rightarrow \mathbf{a} \bullet A\ \mathbf{b}$
    - $A \rightarrow \mathbf{a}\ A \bullet \mathbf{b}$
    - $A \rightarrow \mathbf{a}\ A\ \mathbf{b} \bullet$
- Intuition: '$A \rightarrow \alpha \bullet \beta$' denotes that:
  - we've seen something derivable from α; and
  - it would be legal to see something derivable from β at this point.

Overall Approach

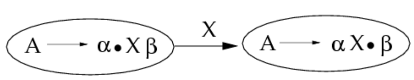
Given a grammar G with start symbol S:

- Construct the augmented grammar by adding a new start symbol S' and a new production $S' \rightarrow S$.
- Construct a finite state automaton whose start state is labeled by the LR(0) item $S' \rightarrow \bullet\, S$.
- Use this automaton to construct the parsing table.

Viable Prefix NFA for LR(0) items

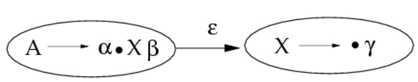
- Each state is labeled by an LR(0) item. The initial state is labeled $S' \rightarrow \bullet\, S$.
- Transitions:

1.



where X is a terminal or nonterminal.

2.



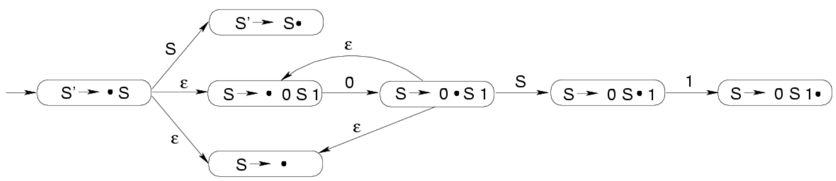
where X is a nonterminal, and $X \rightarrow \gamma$ is a production.

Viable Prefix NFA: Example

Grammar :

$S \rightarrow 0\ S\ 1$

$S \rightarrow \varepsilon$



Viable Prefix NFA ⇒ DFA

- Given a set of LR(0) items *I*, the set *closure*(*I*) is constructed as follows:

  repeat

  1. add every item in *I* to *closure*(*I*);
  2. if $A \rightarrow \alpha \bullet B\beta \in$ *closure*(I) and B is a nonterminal, then for each production $B \rightarrow \gamma$, add the item $B \rightarrow \bullet\ \gamma$ to *closure*(*I*).

  until no new items can be added to *closure*(*I*).

- ***Intuition***:

  $A \rightarrow \alpha \bullet B\beta \in$ *closure*(I) means something derivable from Bβ is legal at this point. This means that something derivable from B (and thus γ) is also legal.

Viable Prefix NFA ⇒ DFA (cont'd)

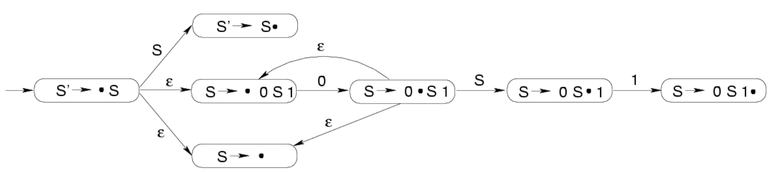
- Given a set of LR(0) items I, the set *goto*(I,X) is defined as

  goto(I, X) = *closure*($\{ A \rightarrow \alpha X \bullet \beta \mid A \rightarrow \alpha \bullet X \beta \in I \}$)

- ***Intuition***:
  - if $A \rightarrow \alpha \bullet X \beta \in I$ then (a) we've seen something derivable from α; and (b) something derivable from $X\beta$ would be legal at this point.
  - Suppose we now see something derivable from X.

    The parser should "go to" a state where (a) we've seen something derivable from αX; and (b) something derivable from β would be legal.

Example



- Let $I_0 = \{S' \rightarrow \bullet S\}$.
- I_1 = closure(I_0) = { S′ → •S, /* from I_0 */

  S → • **0** S **1**, S → • }
- goto (I_1, **0**) = closure({ S → **0** • S **1** })

  = {S → **0** • S **1**, S → • **0** S **1**, S → • }

Viable Prefix DFA for LR(0) Items

1. Given a grammar G with start symbol S, construct the augmented grammar with new start symbol S' and new production $S' \rightarrow S$.
2. *C* = { closure({ $S' \rightarrow \bullet S$ }) }; *// C = a set of sets of items = set of parser states*
3. **repeat** {

```
  for each set of items I ∈ C {
     for each grammar symbol X {
       if ( goto(I,X) ≠ ∅  &&  goto(I,X) ∉ C ) {      // new state
           add goto(I,X) to C;
       }
     }
  }
} until no change to C;
```

4. **return** *C*.

SLR(1) Parse Table Construction I

Given a grammar *G* with start symbol S:

- Construct the augmented grammar *G′* with start symbol S'.
- Construct the set of states $\{I_0, I_1, \ldots, I_n\}$ for the Viable Prefix DFA for the augmented grammar *G′*.
- Each DFA state I_i corresponds to a parser state s_i.
- The initial parser state s_0 coresponds to the DFA state I_0 obtained from the item $S' \rightarrow \bullet\, S$.
- The parser actions in state s_i are defined by the items in the DFA state I_i.

SLR(1) Parse Table Construction II

Parsing action for parser state s_i:

- action table entries:
  - if DFA state I_i contains an item $A \rightarrow \alpha \bullet \mathbf{a}\, \beta$ where **a** is a terminal, and *goto*(I_i, **a**) = I_j : set **action**[i, **a**] = *shift* j.
  - if DFA state I_i contains an item $A \rightarrow \alpha \bullet$, where $A \neq S'$: for each $\mathbf{b} \in$ FOLLOW(A), set **action**[i, **b**] = *reduce* $A \rightarrow \alpha$.
  - if state I_i contains the item $S' \rightarrow S \bullet$: set **action**[i, \$] = *accept*.
- goto table entries:
  - for each nonterminal A, if *goto*(I_i, A) = I_j, then **goto**[i, A] = j.
- any entry not defined by these steps is an *error state*.
- *if any state has multiple entries, the grammar is not SLR(1).*

SLR(1) Shortcomings

- SLR(1) parsing uses *reduce* actions too liberally. Because of this it fails on many reasonable grammars.
- Example *(simple pointer assignments):*

  S → R | L = R

  L → *R | **id**

  R → L

  The SLR parse table has a state { S → L • = R, R → L • }, and FOLLOW(L) = { =, \$ }.

  ⇒ *shift-reduce conflict.*

Improving LR Parsing

- SLR(1) parsing weaknesses can be addressed by incorporating *lookahead* into the LR items in parser states.
  - The lookahead makes it possible to remove some "spurious" reduce actions in the parse table.
  - *The LALR(1) parsers produced by* ***bison*** *and* ***yacc*** *incorporate such lookahead items.*
- This improves parsing power, but at the cost of larger parse tables.

Error Handling

Possible reactions to lexical and syntax errors:

- *ignore the error*. Unacceptable!
- *crash, or quit, on first error*. Unacceptable!
- *continue to process the input*. No code generation.
- *attempt to repair the error*: transform an erroneous program into a similar but legal input.
- *attempt to correct the error*: try to guess what the programmer meant. Not worthwhile.

Error Reporting

- Error messages should refer to the source program.
  - prefer "line 11: X redefined" to "conflict in hash bucket 53"
- Error messages should, as far as possible, indicate the location and nature of the error.
  - avoid "syntax error" or "illegal character"
- Error messages should be specific.
  - prefer "x not declared in function foo" to "missing declaration"
- They should not be redundant.

Error Recovery

- *Lexical errors*: pass the illegal character to the parser and let it deal with the error.
- *Syntax errors*: "*panic mode* error recovery"
  - ***Essential idea***: *skip part of the input and pretend as though we saw something legal, then hope to be able to continue.*
  - Pop the stack until we find a state s such that **goto**[s,A] is defined for some nonterminal A.
  - discard input tokens until we find some token **a** that can legitimately follow A (i.e., $\mathbf{a} \in$ FOLLOW(A)).
  - push the state **goto**[s,A] and continue parsing.