Syntax Analysis

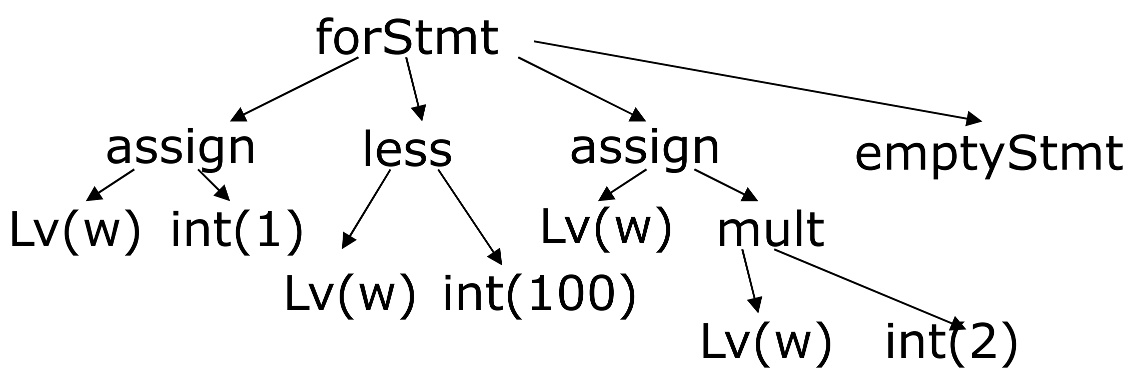
Context-free grammar

Top-down and bottom-up parsing

Front end

- Source program

  for (w = 1; w < 100; w = w * 2);
- Input: a stream of characters
  - ‘f’ ‘o’ ‘r’ ‘(’ `w’ ‘=’ ‘1’ ‘;’ ‘w’ ‘<’ ‘1’ ‘0’ ‘0’ ‘;’ ‘w’…
- Scanning--- convert input to a stream of words (tokens)
  - “for” “(“ “w” “=“ “1” “;” “w” “<“ “100” “;” “w”…
- Parsing---discover the syntax/structure of sentences



Context-free Syntax Analysis

- Goal: recognize the structure of programs
- Description of the language
  - Context-free grammar
- Parsing: discover the structure of an input string
  - Reject the input if it cannot be derived from the grammar

Describing context-free syntax

- Describe how to recursively compose programs/sentences from tokens

```
forStmt: "for" "(" expr ";" expr ";"  expr ")" stmt
expr:  expr + expr
       |  expr – expr
       |  expr * expr
       |  expr / expr
       | ! expr
        ……
stmt: assignment
        | forStmt
        | whileStmt
        | ……
```

Context-free Grammar

- A context-free grammar includes (T,NT,S,P)
  - A set of tokens or terminals --- T
    - Atomic symbols in the language
  - A set of non-terminals --- NT
    - Variables representing constructs in the language
  - A set of productions --- P
    - Rules identifying components of a construct
    - BNF: each production has format A ::= B (or A→B) where
      - A is a single non-terminal
      - B is a sequence of terminals and non-terminals
  - A start non-terminal --- S
    - The main construct of the language
- Backus-Naur Form: textual formula for expressing context-free grammars

Example: simple expressions

- BNF: a collection of production rules

  e ::= n | e+e | ee | e * e | e / e

  - Non-terminals: e
  - Terminal (token): n, +, -, *, /
  - Start symbol: e
- Using CFG to describe regular expressions
  - n ::= d n | d
  - d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- Derivation: top-down replacement of non-terminals
  - Each replacement follows a production rule
  - One or more derivations exist for each program
  - Example: derivations for 5 + 15 * 20

e=>**e*e**=>**e+e*e**=>**5+e*e**=>**5+15*e=>5+15*20**

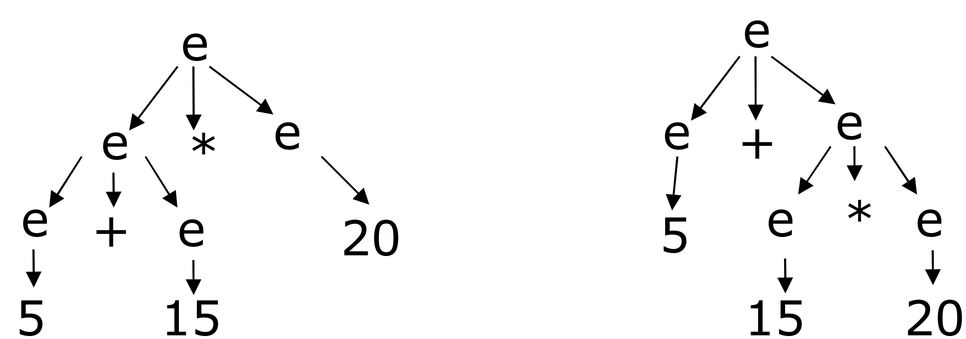
e=>**e+e**=>**5+e**=>**5+e*e**=>5**+15*e**=>**5+15*20**

Parse trees and derivations

- Given a CFG G=(T,NT,P,S), a sentence si belongs to L(G) if there is a derivation from S to si
  - Left-most derivation
    - replace the left-most non-terminal at each step
  - Right-most derivation
    - replace the right-most non-terminal at each step
  - Parse tree: graphical representation of derivations

Grammar: e ::= n | e+e | ee | e * e | e / e
Sentence: 5 + 15 * 20
Derivations:
  e=>e*e=>e+e*e=>5+e*e=>5+15*e=>5+15*20
  e=>e+e=>5+e=>5+e*e=>5+15*e=>5+15*20

Parse trees:



Languages defined by CFG

e ::= num | string | id | e+e

- Support both alternative (|) and recursion
- Cannot incorporate context information
  - Cannot determine the type of variable names
    - Declaration of variables is in the context (symbol table)
  - Cannot ensure variables are always defined before used

```
int w;
0 = w;
for (w = 1; w < 100; w = 2w)
        a = "c" + 3;
a = "c" + w
```

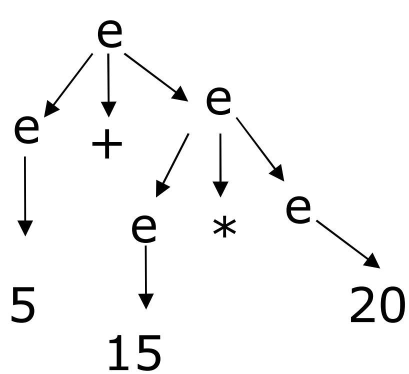
Writing CFGs

- Give BNFs to describe the following languages
  - All strings generated by RE (0|1)*11
  - Symmetric strings of {a,b}. For example
    - "aba" and "babab" are in the language
    - "abab" and "babbb" are not in the language
  - All regular expressions over {0,1}. For example
    - "0|1", "0*", (01|10)* are in the language
    - "0|" and "*0" are not in the language
- For each solution, give an example input of the language. Then draw a parse tree for the input based on your BNF

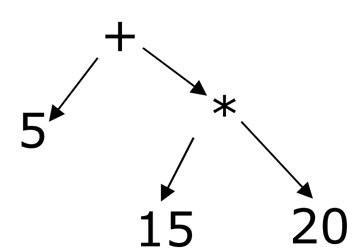
Abstract vs. Concrete Syntax

- Concrete syntax: the syntax programmers write
  - Example: different notations of expressions
    - Prefix + 5 * 15 20
    - Infix 5 + 15 * 20
    - Postfix 5 15 20 * +
- Abstract syntax: the structure recognized by compilers
  - Identifies only the meaningful components
    - The operation
    - The components of the operation

Parse Tree for 5+15*20

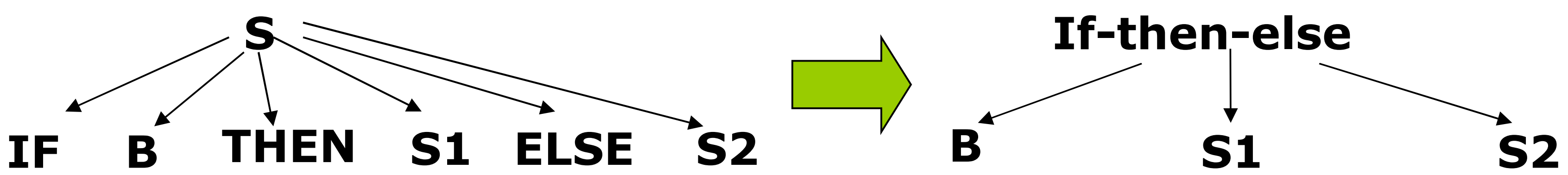


Abstract Syntax Tree for 5 + 15 * 20



Abstract syntax trees

- Condensed form of parse tree
  - Operators and keywords do not appear as leaves
    - They define the meaning of the interior (parent) node



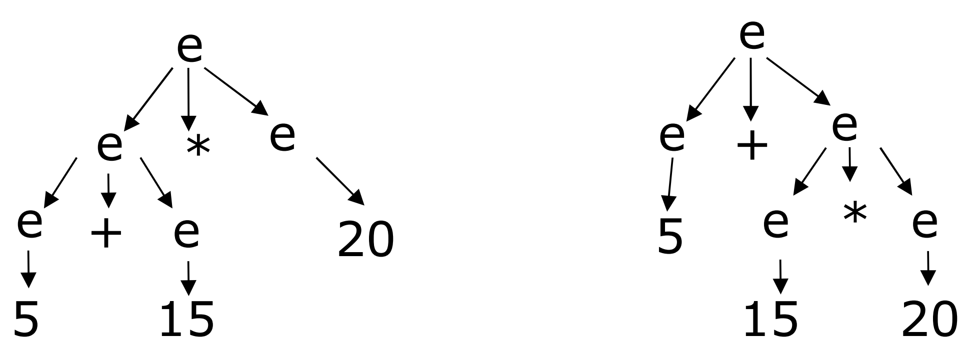
  - Chains of single productions may be collapsed

Ambiguous Grammars

- A grammar is syntactically ambiguous if
  - Some program has multiple parse trees
- Consequence of multiple parse trees
  - Multiple ways to interpret a program

```
Grammar: e  ::=  n  |  e+e  |  ee  | e * e | e / e
Sentence: 5 + 15 * 20
```

Parse trees:



Rewrite ambiguous Expressions

- Solution1: introduce precedence and associativity rules to dictate the choices of applying production rules

  e ::= n | e+e | e-e | e * e | e / e

  - Precedence and associativity
    - * / >> + -
    - All operators are left associative
  - Derivation for n+n*n
    - e=>e+e=>n+e=>n+e*e=>n+n*e=>n+n*n
- Solution2: rewrite productions with additional non-terminals

  E ::= E + T | E – T | T
  T ::= T * F | T / F | F
  F ::= n

  - Derivation for n + n * n
    - E=>E+T=>T+T=>F+T=>n+T=>n+T*F=>n+F*F=>n+n*F=>n+n*n
  - How to modify the grammar if
    - + and - has high precedence than * and /
    - All operators are right associative

Rewrite Ambiguous Grammars

- Disambiguate composition of non-terminals

- Original grammar

  S = IF <expr> THEN S |
  IF <expr> THEN S ELSE S |
  <other>

- Alternative grammar

  S ::= MS | US
  US ::= IF <expr> THEN MS ELSE US |
  IF <expr> THEN S
  MS ::= IF <expr> THEN MS ELSE MS |
  <other>

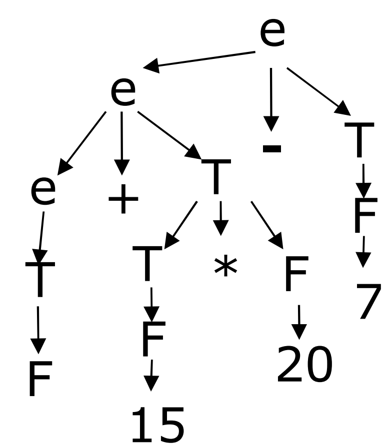
Parsing

- Recognize the structure of programs
  - Given an input string, discover its structure by constructing a parse tree
    - Reject the input if it cannot be derived from the grammar
- Top-down parsing
  - Construct the parse tree in a top-down recursive descent fashion
  - Start from the root of the parse tree, build down towards leaves
- Bottom-up parsing
  - Construct the parse tree in a bottom-up fashion
  - Start from the leaves of the parse tree, build up towards the root

Top-down Parsing

- Start from the starting non-terminal, try to find a left-most derivation

E ::= E + T | E – T | T
T ::= T * F | T / F | F
F ::= n




```
void ParseE() {
  if (use the first rule) {
     ParseE();
     if (getNextToken() != PLUS)
        ErrorRecovery()
     ParseT();
   }
   else if (use the second rule) {
    ...
   }
   else ...
}
void ParseT() { ...... }
void ParseF() { ...... }
```

- Create a procedure for each non-terminal S
  - Recognize the language described by S
  - Parse the whole language in a recursive descent fashion

How to decide which production rule to use?

LL(k) Parsers

- Left-to-right, leftmost-derivation, k-symbol lookahead parsers
  - The production for each non-terminal can be determined by checking at most k input tokens
  - LL(k) grammar: grammars that can be parsed by LL(k) parsers
- LL(1) parser: the selection of every production can be determined by the next input token

Grammar:

E ::= E + T | E – T | T
T ::= T * F | T / F | F
F ::= n | (E)

Every production starts with a number. Not LL(1)
Left recursive ==> not LL(K)

Equivalent LL(1) grammar :

Grammar:

E ::= TE′
E′ ::= + TE′ | - TE′ | ε
T ::= FT′
T′::= *FT′ | / FT′ | ε
F ::= n | (E)

Eliminating left recursion

- A grammar is left-recursive if it has a derivation $A \rightarrow A\diamond$ for some string $\diamond$
  - Left recursive grammar cannot be parsed by recursive descent parsers even with backtracking

$A ::= A\diamond \mid \beta$

$A ::= \beta \quad A'$
$A' ::= \diamond A' \mid \varepsilon$

Grammar:

E ::= E + T | E – T | T
T ::= T * F | T / F | F
F ::= n

Grammar:

E ::= TE′
E′ ::= + TE′ | - TE′ | ε
T ::= FT′
T′::= *FT′ | / FT′ | ε
F ::= n

Problem: Left-recursion could involve multiple derivations

Algorithm: Eliminating left-recursion

1. Arrange the non-terminals in some order $A_1, A_2, \ldots, A_n$
2. for i = 1 to n do
   for j = 1 to i-1 do
   Replace each production

   $A_i ::= A_j$◊ where
   $A_j ::= \beta\ 1 \mid \beta\ 2 \mid \ldots \mid \beta\ k$

   with $A_i ::= \beta\ 1$◊$\mid \beta\ 2$ ◊$\mid \ldots \mid \beta\ k$◊

   end
   Eliminate left-recursion for all A_i productions
   end

Example: S ::= Aa | b
A ::= Ac | Sd

Example: S ::= Aa | b
A ::= Ac | Aad | bd

Example: S ::= Aa | b
A ::= bdA' | A'
A'::= cA' | adA' | ε

Left factoring

- When two alternative productions start with the same symbols, delay the decision until we can make the right choice
  - Can change LL(k) into LL(1)

A::=⋄β 1 | ⋄β 2

A::=⋄A'
A'::=β 1 | β 2

S ::= **IF <expr> THEN S ELSE S**
| IF <expr> THEN S
| <other>

S ::= IF <expr> THEN S S' | <other>
S' ::= ELSE S | ε

S ::= MS | US
US ::= IF <expr> THEN MS ELSE US
| IF <expr> THEN S
MS ::= IF <expr> THEN MS ELSE MS
| <other>

S ::= MS | US
US ::= IF <expr> THEN US'
US'::= MS ELSE S | S
MS ::= IF <expr> THEN MS ELSE MS
| <other>

Predictive parsing table

Grammar:

E ::= TE′
E′ ::= + TE′ | - TE′ | ε
T ::= FT′
T′::= *FT′ | / FT′ | ε
F ::= n

	n	+	-	*	/	$
E	E::=TE′					
E′		E′::=+TE′	E′::=-TE′			E′::= ε
T	T::=FT′					
T′		T′::= ε	T′::= ε	T′::=*FT′	T′::=/FT′	T′::= ε
F	F::=n					

Constructing Predictive Parsing Table

- For each string ⋄, compute
  - First(⋄): terminals that can start all strings derived from ⋄
- For each non-terminal A, compute
  - Follow(A): terminals then can immediately follow A in some derivation
- Algorithm

For each production A::= ⋄, do
For each terminal a in First(⋄), add A::= ⋄ to M[A,a]
If ε ∑ First(⋄),
add A::= ⋄ to M[A,b] for each b ∑ Follow(A).
Each undefined entry of M is error

Compute First

E ::= TE′
E′ ::= + TE′ | - TE′ | ε
T ::= FT′
T′::= *FT′ | / FT′ | ε
F ::= n

Non-terminals:

First(E′)={+,-, ε}
First(T′)={*,/, ε}
First(F) = {n}

First(T)=First(F)={n}
First(E)=First(T)={n}

Strings:

First(TE′)={n}
First(+TE′)={+}
First(-TE′)={-}
First(FT′)={n}
First(*FT′)={*}
First(/FT′)={/}

If X is terminal, then First(X)= {X}

If X::= ε is a production, then ε ∑ First(X)

If x::=y1y2…yk is a production, then First(x)=First(y1y2…yk)

If X=Y1Y2…Yk is a string, then First(Y1)∑ First(X)
If ε ∑ First(Y1), ε ∑ First(Y2)… ε ∑ First(Yi),
then First(Yi+1) ∑ First(X)

Compute Follow

Grammar: E ::= TE′
E′ ::= + TE′ | - TE′ | ε
T ::= FT′
T′::= *FT′ | / FT′ | ε
F ::= n

Non-terminals:

Follow(E)={$}
Follow(E′)={$}
Follow(T) = {$,+,-}
Follow(T′)={$,+,-}
Follow(F)={*,/,+,-,$}

If S is the start non-terminal, then $ ∑ Follow(S)

If A::=⋄Bβ is a production, then First(β)-{ε} ∑ Follow(B)
If ε ∑ First(β), then Follow(A) ∑ Follow(B)

If A::= ⋄B is a production, then Follow(A) ∑ Follow(B)

Build predictive parsing tables

First(TE′)={n}
First(+TE′)={+}
First(-TE′)={-}
First(FT′)={n}
First(*FT′)={*}
First(/FT′)={/}

Follow(E)={$}
Follow(E′)={$}
Follow(T) = {$,+,-}
Follow(T′)={$,+,-}
Follow(F)={*,/,+,-,$}

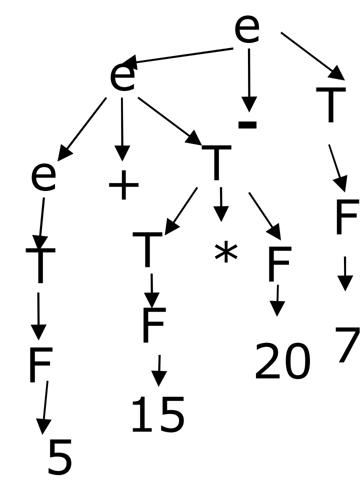
	n	+	-	*	/	$
E	E::=TE′					
E′		E′::=+TE′	E′::=-TE′			E′::= ε
T	T::=FT′					
T′		T′::= ε	T′::= ε	T′::=*FT′	T′::=/FT′	T′::= ε
F	F::=n					

Bottom-up Parsing

- Start from the input string, try reduce it to the starting non-terminal. Equivalent to the reverse of a right-most derivation

Grammar:

E ::= E + T | E – T | T
T ::= T * F | T / F | F
F ::= n



Right-most derivation for 5+15*20-7:

E➔E-T➔E-F➔E-7➔E+T-7➔E+T*F-7
➔E+T*20-7➔E+F*20-7➔E+15*20-7
➔T+15*20-7➔F+15*20-7➔5+15*20-7

Bottom-up parsing: 5+15*20-7➔F+15*20-7➔T+15*20-7
➔E+15*20-7➔E+F*20-7➔E+T*20-7
➔E+T*F-7➔E+T-7➔E-7➔E-F➔E-T➔E

Right-sentential form: any sentence that can appear as an intermediate form of a right-most derivation.

The handle of a right-sentential form ⋄: the substring to reduce to a non-terminal at each step

Handle pruning

Grammar:

E ::= E + T | E – T | T
T ::= T * F | T / F | F
F ::= n

Right-sentential form	Handle	Reducing production
5+15*20-7	**5**	**F::=n**
F+15*20-7	**F**	**T::=F**
T+15*20-7	**T**	**E::=T**
E+15*20-7	**15**	**F::=n**
E+F*20-7	**F**	**T::=F**
E+T*20-7	**20**	**F::=n**
E+T*F-7	**T*F**	**T::=T*F**
E+T-7	**E+T**	**E::=E+T**
E-7	**7**	**F::=n**
E-F	**F**	**T::=F**
E-T	**E-T**	**E::=E-T**
E		

LR(k) parsers

- Left-to-right, rightmost-derivation, k-symbol lookahead
  - Decisions are made by checking the next k input tokens
  - Use a finite automata to configure actions
    - Automata states remember symbols to expect for each production
    - Each (state, input token) pair determines a unique action
- Why use LR parsers?
  - Can recognize more CFGs than can predictive LL(k) parsers
  - Can recognize virtually all programming languages
  - General non-backtracking method, efficient implementation
  - Can detect error at the leftmost position of input string
- Tradeoff: LR(k) vs LL(k) parsers
  - LR parsers are hard to build by hand --- use automatic parser generators (eg., yacc)

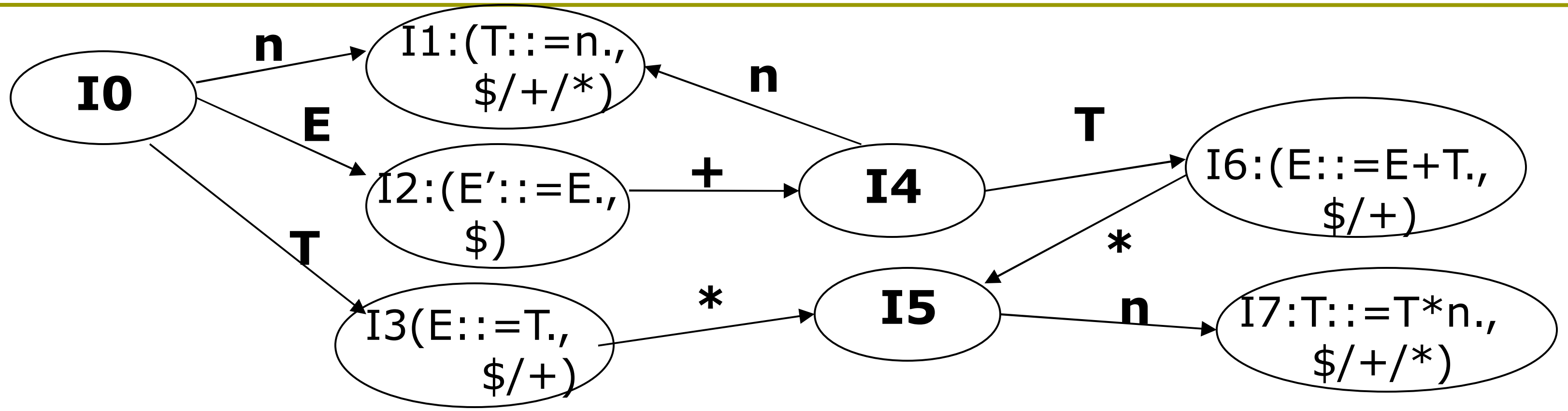
Shift-reduce parsing

- Use a stack to save symbols already processed
  - Prefix of handles processed so far
- Use a finite automata to make decisions
  - State + lookahead => action + goto state
- Implement handle pruning through four actions
  - Shift the current token from input string onto stack
  - Reduce symbols on the top of stack to a non-terminal
  - Accept: success
  - Error

How to locate the handle to be reduced?
Which production to use in reducing a handle?

Shift/reduce conflict: to shift or to reduce?
Reduce/reduce conflict: choose a production to reduce

Example: LR(1) parsing table

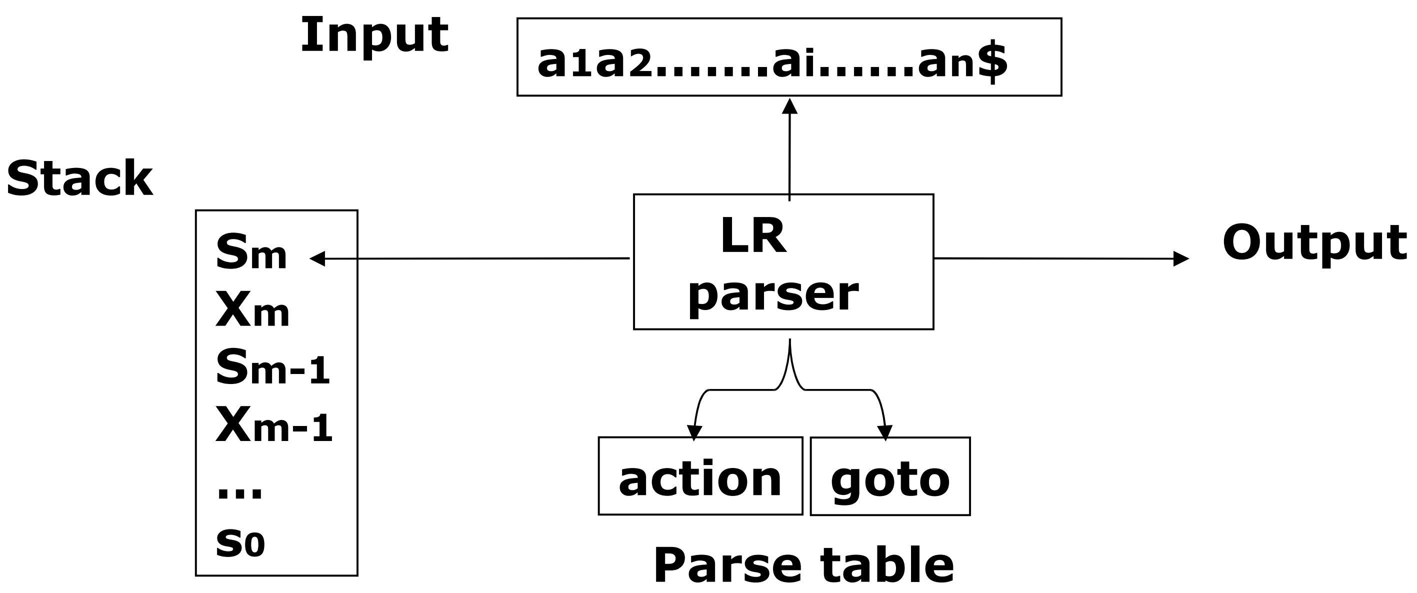


	n	+	*	$		E	T
0	s1					Goto2	Goto3
1		R(T::=n)	R(T::=n)	R(T::=n)			
2		s4		Acc			
3		R(E::=T)	s5	R(E::=T)			
4	s1						Goto6
5	s7						
6		R(E::=E+T)	s5	R(E::=E+T)			
7		R(T::=T*n)	R(T::=T*n)	R(T::=T*n)			

LR shift-reduce parsing

Stack	Input	Action
(0)	**5+15*20$**	**Shift 1**
(0)5(1)	**+15*20$**	**Reduce by T::=n**
(0)T	**+15*20$**	**Goto3**
(0)T(3)	**+15*20$**	**Reduce by E::=T**
(0)E	**+15*20$**	**Goto2**
(0)E(2)	**+15*20$**	**Shift 4**
(0)E(2)+(4)	**15*20$**	**Shift 1**
(0)E(2)+(4)15(1)	***20$**	**Reduce by T::=n**
(0)E(2)+(4)T	***20$**	**Goto6**
(0)E(2)+(4)T(6)	***20$**	**Shift5**
(0)E(2)+(4)T(6)*(5)	**20$**	**Shift7**
(0)E(2)+(4)T(6)*(5)20(7)	**$**	**Reduce by T::=T*n**
(0)E(2)+(4)T	**$**	**Goto6**
(0)E(2)+(4)T(6)	**$**	**Reduce by E::=E+T**
(0)E	**$**	**Goto2**
(0)E(2)	**$**	**Accept**

Model of an LR parser



Configuration of LR parser:

$(s_0X_1s_1X_2s_2...X_ms_m,\ a_ia_{i+1}...a_n\$)$

Right-sentential form: $X_1X_2...X_ma_ia_{i+1}...a_n\$$

Automata states: $s_0s_1s_2...s_m$

Constructing LR parsing tables

- Augmented grammar: add a new starting non-terminal E'
- Build a finite automata to model prefix of handles
  - NFA states: production + position of processed symbols + lookahead
  - Build a DFA by grouping NFA states
- **NFA states: ($S \rightarrow \alpha . \beta$,γ) where $S \rightarrow \alpha\beta$ is a production, γ Σ FOLLOW(S)**
  - **Remembers the handle($\alpha . \beta$) and lookahead(γ) for each state**
- Use lookahead information in automata states
  - LR(0): no lookahead; LR(1): look-ahead one token

Grammar:

E' ::= E E ::= E + T | T T ::= T * n | n

LR(0) items: (E'::=.E) → E → (E'::=E.) (E::=.T) → T → (E::=T.)
(NFA states) (E::=.E+T) → E → (E::=E.+T) → + → (E::=E+.T) → T → (E::=E+T.)

LR(1) items: (E'::=.E,$) → E → (E'::=E.,$) (E::=.T,$) → T → (E::=T.,$)

……

Closure of LR(1) items

- If I is a set of LR(1) items, closure(I)
  - Includes every item in I
  - If (A::= α.Bβ ,a) is in closure(I), and B::=γ is a production, then for every b $\sum$ FIRST(β a), add (B::=.γ,b) to closure(I)

  Repeat until no more new items to add

Grammar:

E' ::= E E ::= E + T | T
T ::= T * n | n

Closure({E'::=.E,$}) = {(E'::=.E,$), (E::=.E+T,$/+), (E::=.T,$/+)
(T::=.T*F,$/+/*), (T::=.n,$/+/*) }

Goto (DFA) transitions

- If I is a set of LR(1) items, X is a grammar symbol, then Goto(I,X) contains
  - For each (A::= $\alpha.X\beta$,a) in I, Closure({(A::= $\alpha X.\beta$,a)})
  - Note: there is no transition from (A::= ε, a)

```
Cononical collection of LR(1) sets
Begin
   C ::= {closure({(S'::=.S,$)})}
   repeat
      for each item set I in C
            for each grammar symbol X
               add Goto(I,X) to C
   until no more item sets can be added to C
```

Example: Building DFA

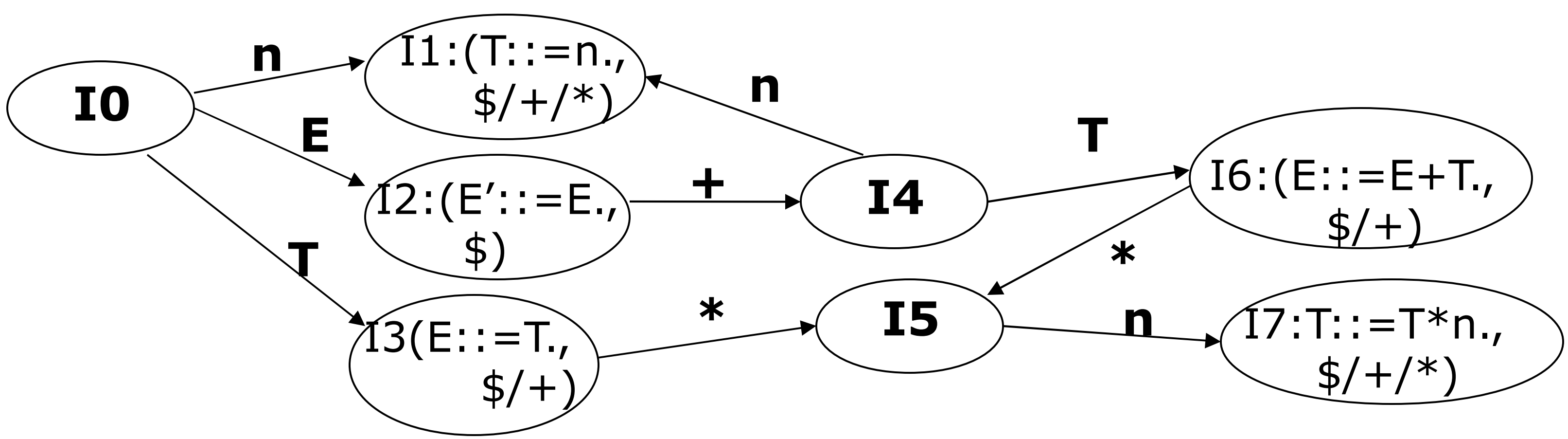
Grammar:

E’ ::= E
E ::= E + T | T
T ::= T * n | n

I0: {(E’::=.E,$), (E::=.E+T,$/+), (E::=.T,$/+), (T::=.T*F,$/+/*), (T::=.n,$/+/*)}
Goto(I0,n): {(T::=n.,$/+/*)} ➔ I1
Goto(I0,E): {(E’::=E.,$), (E::=E.+T,$/+)} ➔ I2
Goto(I0,T): {(E::=T.,$/+), (T::=T.*n,$/+/*)} ➔ I3
Goto(I2,+): {(E::=E+.T,$/+), (T::=.T*n,$/+/*), (T::=.n,$/+/*)} ➔ I4
Goto(I3,*): {(T::=T*.n,$/+/*)} ➔I5
Goto(I4,T): {(E::=E+T.,$/+), (T::=T.*n,$/+/*)} ➔ I6
Goto(I4,n): {(T::=n.,$/+/*)} ➔ I1
Goto(I5,n): {(T::=T*n.,$/+/*)} ➔I7
Goto(I6,*): {(T::=T*.n,$/+/*)} ➔ I5

LR(1) DFA Transitions

I0: {(E'::=.E,$), (E::=.E+T,$/+), (E::=.T,$/+), (T::=.T*n,$/+/*),
(T::=.n,$/+/*)}
Goto(I0,n): {(T::=n.,$/+/*)} ➔ I1
Goto(I0,E): {(E'::=E.,$), (E::=E.+T,$/+)} ➔ I2
Goto(I0,T): {(E::=T.,$/+), (T::=T.*n,$/+/*)} ➔ I3
Goto(I2,+): {(E::=E+.T,$/+), (T::=.T*n,$/+/*), (T::=.n,$/+/*)} ➔ I4
Goto(I3,*): {(T::=T*.n,$/+/*)} ➔I5
Goto(I4,T): {(E::=E+T.,$/+), (T::=T.*n,$/+/*)} ➔ I6
Goto(I4,n): {(T::=n.,$/+/*)} ➔ I1
Goto(I5,n): {(T::=T*n.,$/+/*)} ➔I7
Goto(I6,*): {(T::=T*.n,$/+/*)} ➔ I5



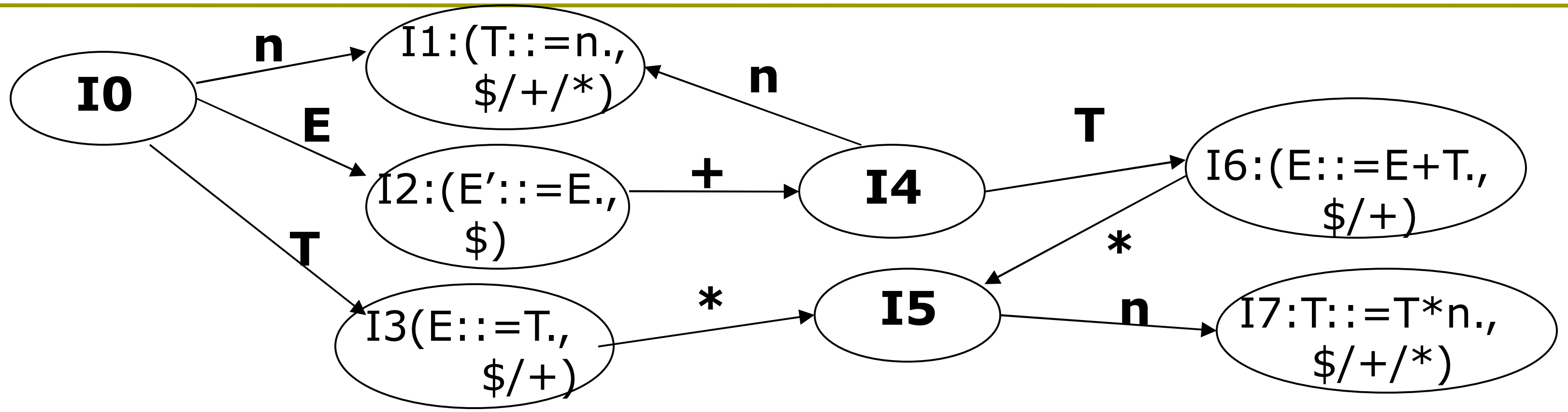
Constructing LR(1) Parsing Table

- Input: augmented grammar G'
- Output: parsing table functions (action and goto)
- Method:

1. Construct C={I0,I1,…,In}, the canonical LR(1) collection
2. Create a state i for each I_i ∑ C

a) if Goto(I_i, a) = I_j and "a" is a terminal
set action[i,a] to "shift j".
b) if Goto(I_i,A) = Ij and "A" is a non-terminal,
set GOTO[i,A] to j.
b) if (A::=⋄.,a) is in I_i (note: ⋄ could be ε)
set action[i,a] to "reduce A::= ⋄"
c) if (S'::=S.,$) is in I_i, set action[i,$] to "accept".

Example: LR(1) parsing table



	n	+	*	$		E	T
0	s1					Goto2	Goto3
1		R(T::=n)	R(T::=n)	R(T::=n)			
2		s4		Acc			
3		R(E::=T)	s5	R(E::=T)			
4	s1						Goto6
5	s7						
6		R(E::=E+T)	s5	R(E::=E+T)			
7		R(T::=T*n)	R(T::=T*n)	R(T::=T*n)			

Precedence and Associativity

E ::= E + E | E * E | (E) | id

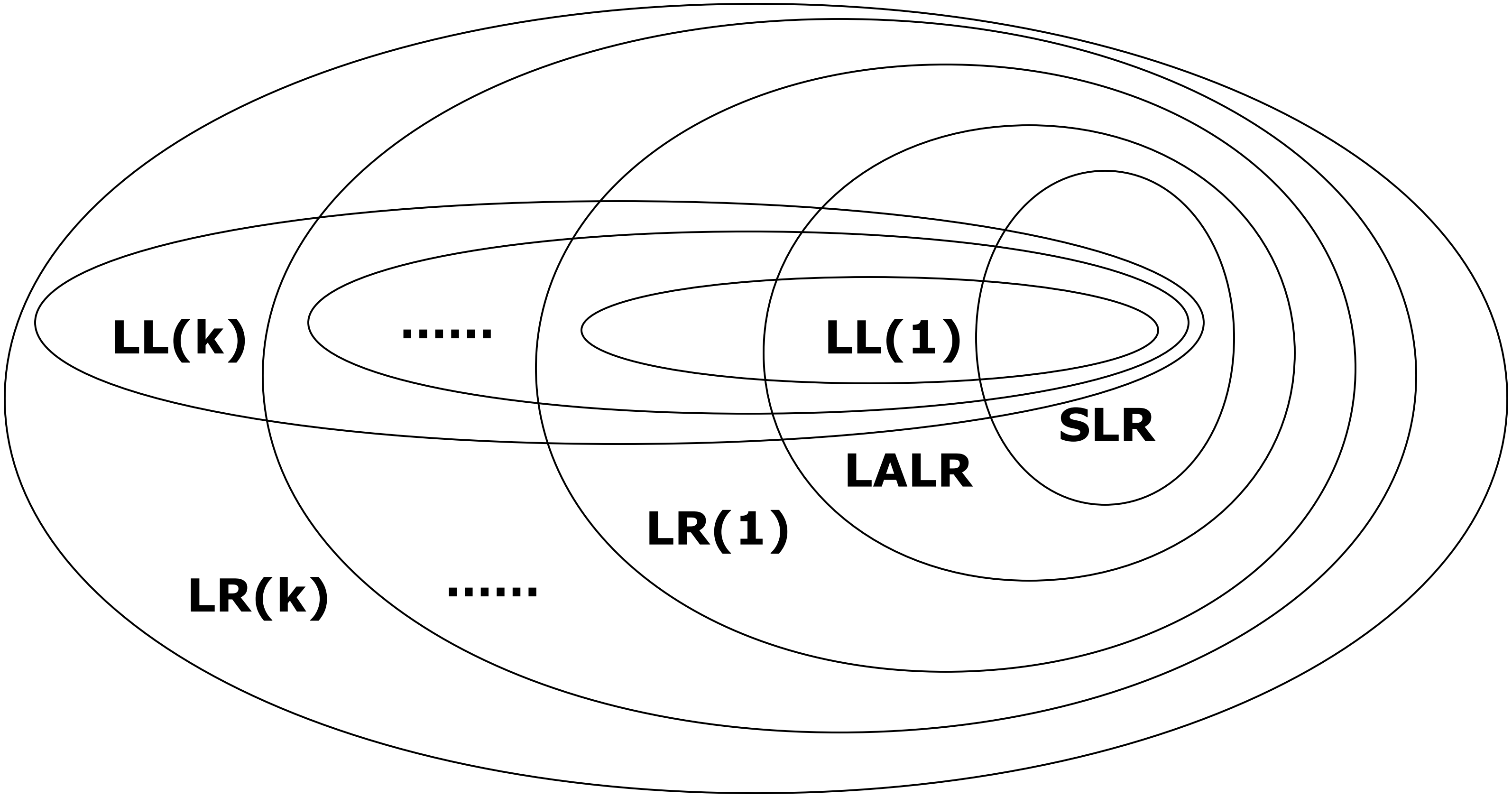
I7: {E::=E+E., E::=E.+E, E::=E.*E}

Operator + is left-associative
on input token +, reduce with E::=E+E
Operator * has higher precedence than +
on input token *, shift * onto stack

I8: {E::=E*E., E::=E.+E, E::=E.*E}

Operator * is left-associative
on input token *, reduce with E::=E*E
Operator * has higher precedence than +
on input token +, reduce with E::=E*E

Parser hierarchy



Summary: grammars and Parsers

- Specification and implementation of languages
  - Grammars specify the syntax of languages
  - Parsers implement the specification
- Context-free grammars
  - Ambiguous vs non-ambiguous grammars
  - Left-recursive grammars vs. LL parsers
  - Left-factoring of grammars
- Parsers
  - Backtracking vs predictive parsers
  - LL parsers vs LR parsers
  - Lookahead information