Programming Languages
(CS 550)

Scanner and Parser Generators

Jeremy R. Johnson
Theme

- We have now seen how to describe syntax using regular expressions and grammars and how to create scanners and parsers, by hand and using automated tools. In this lecture we provide more details on parsing and scanning and indicate how these tools work.

- Reading: chapter 2 of the text by Scott.
Parser and Scanner Generators

- Tools exist (e.g. yacc/bison¹ for C/C++, PLY for python, CUP for Java) to automatically construct a parser from a restricted set of context free grammars (LALR(1) grammars for yacc/bison and the derivatives CUP and PLY)

- These tools use table driven bottom up parsing techniques (commonly shift/reduce parsing)

- Similar tools (e.g. lex/flex for C/C++, Jflex for Java) exist, based on the theory of finite automata, to automatically construct scanners from regular expressions

¹bison in the GNU version of yacc
Outline

- Scanners and DFA
- Regular Expressions and NDFA
- Equivalence of DFA and NDFA
- Regular Languages and the limitations of regular expressions
- Recursive Descent Parsing
- LL(1) Grammars and Tob-down (Predictive) Parsing
- LR(1) Grammars and Bottom-up Parsing
Regular Expressions

- Alphabet = \( \Sigma \)
- A language over \( \Sigma \) is subset of strings in \( \Sigma \)
- Regular expressions describe certain types of languages
  - \( \emptyset \) is a regular expression
  - \( \varepsilon = \{ \varepsilon \} \) is a regular expression
  - For each \( a \) in \( \Sigma \), \( \{a\} \) is a regular expression
  - If \( r \) and \( s \) are regular expressions denoting languages \( R \) and \( S \) respectively then \( (r | s) \), \( rs \), and \( (r^*) \) are regular expressions
- E.G. 00, (0|1)*, (0|1)*00(0|1)*, 00*11*22*, (1|10)*
List Tokens

- LPAREN = ‘(‘
- RPAREN = ‘)’
- COMMA = ‘,’
- NUMBER = DIGIT DIGIT*
- DIGIT = 0|1|2|3|4|5|6|7|8|9

- Unix shorthand: [0-9], DIGIT+

- Whitespace: (‘ ’ | ‘\n’ | ‘\t’)*
TOKEN GetToken()
{
    int val = 0;
    if (c = getchar() == eof) then return None end if;
    while c ∈ {‘ ’, ‘\n’, ‘\t’} then c = getchar() end do;
    if c ∈ {‘(’, ‘,’ , ‘)’} then return c end if;
    if c ∈ {‘0’,…, ‘9’} then
        while c ∈ {‘0’,…, ‘9’} do
            val = val*10 + (c – ‘0’);  c = getchar();
        end do;
        putchar(c);  return (NUMBER,val);
    else 
        return None;
    end if;
}
Flex List Tokens

{%
#include "list.tab.h"
extern int yylval;
%
%
[ \t\n] ;
"(" return yytext[0];
")" return yytext[0];
"," return yytext[0];
[0-9]+$ { yylval = atoi(yytext); return NUMBER; }
%
%}
Deterministic Finite Automata

● Input comes from alphabet $A$
● Finite set of states, $S$, start state, $s_0$, Accepting States, $F$
● Transition $T$ from state to state depending on next input
  $M = (A, S, s_0, F, T)$
● The language accepted by a finite automata is the set of input strings that end up in accepting states
Example 1

- Create a finite state automata that accepts strings of a’s and b’s with an even number of a’s.

```
Example 1
 Create a finite state automata that accepts strings of a’s and b’s with an even number of a’s.

S0
b

S1
a

abbbabaabbb
011110010000
```
DFA Implementation

- Program to implement DFA

bool EA()
{
    S0: x = getchar();
    if (x == 'b') goto S0;
    if (x == 'a') goto S1;
    if (x == ENDM) return true;

    S1: x = getchar();
    if (x == 'b') goto S1;
    if (x == 'a') goto S0;
    if (x == ENDM) return false;
}

List DFA
Calculator Tokens

- **ASSIGN** = ‘:='
- **PLUS** = ‘+’, **MINUS** = ‘-’, **TIMES** = ‘*’, **DIV** = ‘/’
- **LPAREN** = ‘(’, **RPAREN** = ‘)’
- **NUMBER** = **DIGIT** **DIGIT*** | **DIGIT*** (. **DIGIT**|**DIGIT** .) **DIGIT***
- **ID** = **LETTER** (**LETTER** | **DIGIT**)*
- **DIGIT** = 0|1| … |9, **LETTER** = a|…|z|A|…|Z
- **COMMENT** = /* (non-* | * non-/))* */ | // (non-newline)* newline
- **WHITESPACE** = (‘ ’ | ‘\n’ | ‘\t’)*
Calculator DFA
# Table Driven Scanner

<table>
<thead>
<tr>
<th>State</th>
<th>' ', \t \n \r / * ( ) \n + \n - : = \n . digit letter other token</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17 17 10 6 7 8 9 11 13 14 16</td>
</tr>
<tr>
<td>2</td>
<td>3 4 4 3 4 3 3 3 3 3 3 3 3 3 3</td>
</tr>
<tr>
<td>3</td>
<td>3 18 3 3 3 3 3 3 3 3 3 3 3 3 3</td>
</tr>
<tr>
<td>4</td>
<td>4 4 4 5 4 4 4 4 4 4 4 4 4 4 4</td>
</tr>
<tr>
<td>5</td>
<td>4 4 18 5 4 4 4 4 4 4 4 4 4 4</td>
</tr>
<tr>
<td>6</td>
<td>4 18 15 4 4 4 4 4 4 4 4 4 4 4 4</td>
</tr>
<tr>
<td>7</td>
<td>div</td>
</tr>
<tr>
<td>8</td>
<td>lparen</td>
</tr>
<tr>
<td>9</td>
<td>rparen</td>
</tr>
<tr>
<td>10</td>
<td>plus</td>
</tr>
<tr>
<td>11</td>
<td>minus</td>
</tr>
<tr>
<td>12</td>
<td>times</td>
</tr>
<tr>
<td>13</td>
<td>assign</td>
</tr>
<tr>
<td>14</td>
<td>number</td>
</tr>
<tr>
<td>15</td>
<td>number</td>
</tr>
<tr>
<td>16</td>
<td>identifier</td>
</tr>
<tr>
<td>17</td>
<td>white-space</td>
</tr>
<tr>
<td>18</td>
<td>comment</td>
</tr>
</tbody>
</table>
Non-Deterministic Finite Automata

- Same as DFA $M = (A,S,s_0,F,T)$ except
  - Can have multiple transitions from same state with same input
  - Can have epsilon transitions
  - Except input string if there is a path to an accepting state

- The languages accepted by NDFA are the same as DFA
Example 2a

- DFA accepting \((a|b)^*abb\)
Example 2b

- NDFA accepting \((a|b)^*abb\)
Simulating an NDFA

- Compute $S = \text{set of states NDFA could be in after reading each symbol in the input.}$
  - $S_i = \text{set of possible states after reading } i \text{ input symbols}$

1. Initialize $S_0 = \text{EpsilonClosure} \{0\}$
2. for $i = 1, \ldots, \text{len(str)}$
   1. $T_i = \bigcup \{s \in S_{i-1}\} T[s, \text{str}[i]]$
   2. $S_i = \text{EpsilonClosure}(T_i)$

\[
\begin{array}{cccccc}
  b & a & b & b \\
  \{0,1\} & \{0,1\} & \{0,1,2\} & \{0,1,3\} & \{0,1,4\}
\end{array}
\]
NDFA from Regular Expressions

- Base case – c
- Union – R|S
- Concatenation – RS
- Closure – R*
Example 3

- Construct a NDFA that accepts the language generated by the regular expression $(a|bc)$
Regular Expression Compiler

```c
{%
#include "machine.h"
char input[100];
%
%union{
  MachinePtr ndfa;
  char symbol;
}

%token <symbol> LETTER
%type <ndfa> regexp
%type <ndfa> cat
%type <ndfa> kleene
%%

statement: regexp  {
  do {
    printf("Enter string\n");
    if (scanf("%s",input) != EOF)
      Simulate($1,input);
    else exit(1);
  }  while (1);  }
regexp:     regexp '|' cat  { $$ = MachineOr($1,$3); }
|     cat     { $$ = $1; }  ;
cat:        cat kleene { $$ = MachineConcat($1,$2); }
|       kleene     { $$ = $1; } ;
kleene:     '(' regexp ')' { $$ = $2; }
|       kleene '*'     { $$ = MachineStar($1); }
|       LETTER         { $$ = BaseMachine($1); };
%%
```
NDFA $\rightarrow$ DFA

- Find an equivalent DFA for Example 3
- States in DFA are sets of states from NDFA [keep track of all possible transitions]
Exercise 1

1. Construct NDFA that recognizes (see pages 55-57 of text)
   1. \( d^* (\cdot d \mid d.)d^* \)

2. Convert NDFA from (1) to DFA (see pages 56-58 of text)
Solution 1.1
Solution 1.2

A[1,2,4,5,8] -> B[2,3,4,5,8,9]

C[6] -> D[6,10,11,12,14]

E[7,11,12,14] -> F[7,11,12,13,14]

G[11,12,13,14]
Minimizing States in DFA

- The exists a unique minimal state DFA for any language described by a regular expression

- Combine equivalent states
  - \( p \equiv q \) if for each input string \( x \) \( T(p,x) \) is an accepting state iff \( T(q,x) \) is an accepting state
  - Initialize two sets of states: accepting and non-accepting
  - Partition state sets which transition into multiple sets of states
Exercise 2

1. Find an equivalent DFA to the one in Exercise 1 that minimizes the number of states (see page 59 of text)

Ambiguity: \( T(A,d) = T(B,d) \neq T(C,d) \Rightarrow \text{split} \ ABC \rightarrow AB,C \)
Solution 2
Equivalence of Regular Expressions and DFA

- The languages accepted by finite automata are equivalent to those generated by regular expressions
  - Given any regular expression $R$, there exists a finite state automata $M$ such that $L(M) = L(R)$
    - Proof is given by previous construction
  - Given any finite state automata $M$, there exists a regular expression $R$ such that $L(R) = L(M)$
    - The basic idea is to combine the transitions in each node along all paths that lead to an accepting state. The combination of the characters along the paths are described using regular expressions.
Example 4

- Create a regular expression for the language that consists of strings of a’s and b’s with an even number of a’s.

\[ b^*|(b^*ab^*a)^* \]
Grammars and Regular Expressions

- Given a regular expression \( R \), there exists a grammar with syntactic category \(<S>\) such that \( L(R) = L(<S>) \).

- There are grammars such that there does NOT exist a regular expression \( R \) with \( L(<S>) = L(R) \)

  \[ <S> \rightarrow a<S>b | \varepsilon \]
  \[ L(<S>) = \{a^n b^n, \; n=0,1,2,\ldots\} \]
Example 5

- Create a grammar that generates the language that consists of strings of a’s and b’s with an even number of a’s.

\[
\begin{align*}
<S0> & \rightarrow b<S0> \\
<S0> & \rightarrow a<S1> \\
<S0> & \rightarrow \varepsilon \\
<S1> & \rightarrow b<S1> \\
<S1> & \rightarrow a<S0>
\end{align*}
\]
Proof that $a^n b^n$ is not Recognized by a Finite State Automata

- To show that there is no finite state automata that recognizes the language $L = \{a^n b^n, n = 0,1,2,...\}$, we assume that there is a finite state automata $M$ that recognizes $L$ and show that this leads to a contradiction.

- Since $M$ is a finite state automata it has a finite number of states. Let the number of states = $m$.

- Since $M$ recognizes the language $L$ all strings of the form $a^k b^k$ must end up in accepting states. Choose such a string with $k = n$ which is greater than $m$. 
Proof that $a^n b^n$ is not Recognized by a Finite State Automata

- Since $n > m$ there must be a state $s$ that is visited twice while the string $a^n$ is read [we can only visit $m$ distinct states and since $n > m$ after reading $(m+1)$ a’s, we must go to a state that was already visited].

- Suppose that state $s$ is reached after reading the strings $a^j$ and $a^k$ ($j \neq k$). Since the same state is reached for both strings, the finite state machine can not distinguish strings that begin with $a^j$ from strings that begin with $a^k$.

- Therefore, the finite state automata must either accept or reject both of the strings $a^j b^j$ and $a^k b^j$. However, $a^j b^j$ should be accepted, while $a^k b^j$ should not be accepted.
List Grammar

- $<\text{list}> \rightarrow ( <\text{sequence}> ) | ( )$

- $<\text{sequence}> \rightarrow <\text{listerlement}> , <\text{sequence}> | <\text{listerlement}>$

- $<\text{listerlement}> \rightarrow <\text{list}> | \text{NUMBER}$
Recursive Descent Parser

list()
{
    match(‘(‘);
    if token ≠ ‘)’ then
        seq();
    endif;
    match(‘)’);
}
Recursive Descent Parser

seq()
{
    elt();
    if token = ‘,’ then
        match(‘,’);
        seq();
        seq();
    endif;
}
Recursive Descent Parser

elt()
{
    if token = ‘(‘ then
        list();
    else
        match(NUMBER);
    endif;
}


Yacc (bison) Example

%token NUMBER  /* needed to communicate with scanner */
%%
list:
    '(' sequence ')' { printf("L -> ( seq )\n"); }
|  '(' ')'  { printf("L -> () \n "); }
sequence:
    listelement ',' sequence { printf("seq -> LE,seq\n"); }
|  listelement { printf("seq -> LE\n"); } ;
listelement:
    NUMBER  { printf("LE -> %d\n",$1); }
|    list { printf("LE -> L\n"); } ;
%%
/* since no code here, default main constructed that simply calls parser. */
Top-down vs. Bottom-up Parsing

```
id_list
  id_list
    id(A)  id_list_tail
    , id(B)  id_list_tail
  id_list
    id(A)  id_list_tail
      , id(B)  id_list_tail
        , id(C)  id_list_tail
  id_list
    id(A)  id_list_tail
      , id(B)  id_list_tail
        , id(C)  id_list_tail
  id_list
    id(A)  id_list_tail
      , id(B)  id_list_tail
        , id(C)  id_list_tail

id_list -> id  id_list_tail
id_list_tail -> , id  id_list_tail
id_list_tail -> ;
```
Bottom-up Parsing LR Grammar

\[
\begin{align*}
\text{id} & \rightarrow \text{id}
\end{align*}
\]
LR Calculator Grammar

Figure 2.24

Program → stmt list $$
stmt_list → stmt_list stmt 
    | stmt
stmt → id := expr 
    | read id 
    | write expr
expr → term 
    | expr add op term
term → factor 
    | term mult_op factor
factor → ( expr ) 
    | id 
    | number
add op → + | -
mult op → * | /
Here is an LL(1) grammar (Fig 2.15):

1. program → stmt_list $$
2. stmt_list → stmt stmt_list
   | ε
3. stmt → id := expr
4. stmt → read id
5. stmt → write expr
6. expr → term term_tail
7. term_tail → add op term term_tail
8. term_tail → ε
LL Calculator Grammar

LL(1) grammar (continued)

10. term → factor fact_tail
11. fact_tail → mult_op fact fact_tail
12.                | ε
13. factor → ( expr )
14.                | id
15.                | number
16. add_op → +
17.                | −
18. mult_op → *
19.                | /
Predictive Parser

Predict which rules to match

\[ A \rightarrow \alpha \]
- when next token can start \( \alpha \)
- \( \alpha \rightarrow * \) \( \epsilon \) and the next token can follow \( A \)

\[ \text{PREDICT}(A \rightarrow \alpha) = \text{FIRST}(\alpha) \cup \text{FOLLOW}(A) \text{ if } \text{EPS}(\alpha) \]

- \( \text{PREDICT} \( \text{program} \rightarrow \text{stmt_list} \) \)
- \( \text{FIRST}(\text{stmt_list}) = \{ \text{id, read, write} \} \)
- \( \text{FOLLOW}(\text{stmt_list}) = \{ $$ \} \)

Intersection of PREDICT sets for same lhs must be empty
Recursive Descent Parser

procedure program
  case input_token of
    id, read, write, $$:
      stmt_list; match($$)
    otherwise error
  end;

procedure stmt_list
  case input_token of
    id, read, write:
      stmt; stmt_list
    $$: skip
    otherwise error
  end;

procedure stmt
  case input_token of
    id:
      match(id); match(=); expr
    read:
      match(read); match(id)
    write:
      match(write); expr
    otherwise error
  end;

procedure expr
  case input_token of
    id, number, (:
      term; term_tail
    otherwise error
  end;

procedure term_tail
  case input_token of
    +, -: add_op;
      term;
      term_tail
    ),
    id, read, write, $$:
      skip
    otherwise error
  end;

procedure term
  case input_token of
    id, number, (:
      factor;
      factor_tail
    otherwise error
  end;
Exercise 3

Trace through the recursive descent parser and build parse tree for the following program

1. read A
2. read B
3. sum := A + B
4. write sum
5. write sum / 2
Solution 3
# Table-Driven LL Parser

<table>
<thead>
<tr>
<th>Top-of-stack nonterminal</th>
<th>id</th>
<th>number</th>
<th>read</th>
<th>write</th>
<th>:=</th>
<th>( )</th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>$$</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>program</code></td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td><code>stmt_list</code></td>
<td>2</td>
<td>-</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td><code>stmt</code></td>
<td>4</td>
<td>-</td>
<td>5</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><code>expr</code></td>
<td>7</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><code>term_tail</code></td>
<td>9</td>
<td>-</td>
<td>9</td>
<td>9</td>
<td>-</td>
<td>-</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>-</td>
<td>9</td>
</tr>
<tr>
<td><code>term</code></td>
<td>10</td>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><code>factor_tail</code></td>
<td>12</td>
<td>-</td>
<td>12</td>
<td>12</td>
<td>-</td>
<td>-</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td><code>factor</code></td>
<td>14</td>
<td>15</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>13</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><code>add_op</code></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>16</td>
<td>17</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><code>mul_op</code></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>18</td>
<td>19</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

Copyright © 2009 Elsevier
# Table-Driven LL Parser

## Parse Stack

<table>
<thead>
<tr>
<th>Program</th>
<th>Input Stream</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>program</code></td>
<td><code>read A read B …</code></td>
<td>Initial stack contents</td>
</tr>
<tr>
<td><code>stmt_list $$</code></td>
<td><code>read A read B …</code></td>
<td><code>program → stmt_list $$</code></td>
</tr>
<tr>
<td><code>stmt stmt_list $$</code></td>
<td><code>read A read B …</code></td>
<td><code>stmt_list → stmt stmt_list</code></td>
</tr>
<tr>
<td><code>read id stmt_list $$</code></td>
<td><code>read A read B …</code></td>
<td><code>stmt → read id</code></td>
</tr>
<tr>
<td><code>id stmt_list $$</code></td>
<td><code>A read B …</code></td>
<td><code>match(read)</code></td>
</tr>
<tr>
<td><code>stmt_list $$</code></td>
<td><code>read B …</code></td>
<td><code>match(id)</code></td>
</tr>
<tr>
<td><code>stmt stmt_list $$</code></td>
<td><code>read B …</code></td>
<td><code>stmt_list → stmt stmt_list</code></td>
</tr>
<tr>
<td><code>…</code></td>
<td><code>…</code></td>
<td><code>…</code></td>
</tr>
<tr>
<td><code>stmt_list $$</code></td>
<td><code>read B …</code></td>
<td><code>term_tail → ε</code></td>
</tr>
<tr>
<td>$$</td>
<td><code>…</code></td>
<td><code>stmt_list → ε</code></td>
</tr>
</tbody>
</table>
Computing First, Follow, Predict

Algorithm First/Follow/Predict:

- FIRST(α) = \{c : α →* c β\}
- FOLLOW(A) = \{c : S →+ α A c β\}
- EPS(α) = if α →* ε then true else false
- Predict (A → α) = FIRST(α) ∪ (if EPS(α) then FOLLOW(A) else ∅)
Predict Set for LL Parser

FIRST

program {id, read, write, $$}
stmt_list {id, read, write, }$
stmt {id, read, write}
expr { (, id, number}
term_tail {+, -, }$
term { (, id, number}
factor_tail {*, /, }$
factor { (, id, number}
add_op {+, - }$
mult_op {*, /, }$

Also note that FIRST(a) = \{a\} \forall \text{ tokens a.}

FOLLOW

id {+, -, *, /, }, :=, id, read, write, $$
number {+, -, *, /, }, id, read, write, $$
read {id}$
write { (, id, number}
( { (, id, number}
) {+, -, *, /, }, id, read, write, $$
:= { (, id, number}
+ { (, id, number}
- { (, id, number}
* { (, id, number}
/ { (, id, number}
$$ {e}$
program {e}$
stmt_list {$$
stmt {id, read, write, $$}
expr { (, id, read, write, $$}
term_tail { (, id, read, write, $$
term {+ , - , }, id, read, write, $$
factor_tail {+ , - , }, id, read, write, $$
factor {+ , - , *, / , }, id, read, write, $$
add_op { (, id, number}
mult_op { (, id, number}

PREDICT

1  program \rightarrow stmt\_list $$ \{ id, read, write, $$
2  stmt\_list \rightarrow stmt stmt\_list \{ id, read, write, $$
3  stmt\_list \rightarrow e $$
4  stmt \rightarrow id := expr \{ id}\n5  stmt \rightarrow read id \{read\}
6  stmt \rightarrow write expr \{write\}
7  expr \rightarrow term term\_tail \{ , id, number\}
8  term\_tail \rightarrow add\_op term term\_tail \{+, -\}
9  term\_tail \rightarrow e \{id, read, write, $$
10  term \rightarrow factor factor\_tail \{ , id, number\}
11  factor\_tail \rightarrow mult\_op factor factor\_tail \{*, /\}
12  factor\_tail \rightarrow e \{+ , - , \}, id, read, write, $$
13  factor \rightarrow ( expr ) \{\}
14  factor \rightarrow id \{id\}
15  factor \rightarrow number \{number\}
16  add\_op \rightarrow + \{+\}
17  add\_op \rightarrow - \{-\}
18  mult\_op \rightarrow * \{*\}
19  mult\_op \rightarrow / \{/\}

Figure 2.22: FIRST, FOLLOW, and PREDICT sets for the calculator language.
LR Parsing

- Bottom up (rightmost derivation)
  - Maintain forest of partially completed subtrees of the parse tree
  - Join trees together when recognizing symbols in rhs of production
  - Keep roots of partially completed trees on stack
    - Shift when new token
    - Reduce when top symbols match rhs
- Table driven
Top-down vs. Bottom-up Parsing

\[
\begin{align*}
\text{id\_list} & \rightarrow \text{id} \text{id\_list\_tail} \\
\text{id\_list\_tail} & \rightarrow , \text{id} \text{id\_list\_tail} \\
\text{id\_list\_tail} & \rightarrow ;
\end{align*}
\]
LR Parsing Example

Stack

ε
id(A)
id(A),
id(A), id(B)
id(A), id(B),
id(A), id(B), id(C)
id(A), id(B), id(C);
id(A), id(B), id(C) id_list_tail
id(A), id(B) id_list_tail
id(A) id_list_tail
id_list

Remaining Input

A, B, C;
, B, C;
B, C;
, C;
C;
;
LR Calculator Grammar

(Figure 2.24, Page 73):

1. program → stmt list $$
2. stmt_list → stmt_list stmt
3. | stmt
4. stmt → id := expr
5. | read id
6. | write expr
7. expr → term
8. | expr add op term
LR Calculator Grammar

LR grammar (continued):

9. term → factor
10. | term mult_op factor
11. factor → ( expr )
12. | id
13. | number
14. add op → +
15. | −
16. mult op → *
17. | /
LR Parser State

- Keep track of set of productions we might be in along with where in those productions we might be.

- Initial state for calculator grammar

```
program → . stmt_list $$             // basis
stmt_list → . stmt_list stmt        // yield
stmt_list → . stmt
stmt → . id := expr
stmt → . read id
stmt → . write expr
```
### LR Parser States

<table>
<thead>
<tr>
<th>State</th>
<th>Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0. program  →  stmt_list $1</td>
<td>on stmt_list shift and goto 2</td>
</tr>
<tr>
<td>stmt_list  →  stmt stmt</td>
<td>on stmt shift and reduce (pop 1 state, push stmt on input)</td>
</tr>
<tr>
<td>stmt_list  →  stmt_list stmt</td>
<td>on stmt shift and reduce (pop 2 states, push stmt_list on input)</td>
</tr>
<tr>
<td>stmt  →  id  →  expr</td>
<td>on id shift and goto 3</td>
</tr>
<tr>
<td>stmt  →  read id</td>
<td>on read shift and goto 1</td>
</tr>
<tr>
<td>stmt  →  write expr</td>
<td>on write shift and goto 4</td>
</tr>
<tr>
<td>1. stmt  →  read id</td>
<td>on id shift and reduce (pop 2 states, push stmt on input)</td>
</tr>
<tr>
<td>2. program  →  stmt_list $2</td>
<td>on stmt_list shift and goto 3</td>
</tr>
<tr>
<td>stmt_list  →  stmt stmt</td>
<td>on stmt shift and reduce (pop 2 states, push stmt_list on input)</td>
</tr>
<tr>
<td>stmt  →  id  →  expr</td>
<td>on id shift and goto 3</td>
</tr>
<tr>
<td>stmt  →  read id</td>
<td>on read shift and goto 1</td>
</tr>
<tr>
<td>stmt  →  write expr</td>
<td>on write shift and goto 4</td>
</tr>
<tr>
<td>3. stmt  →  id  →  expr</td>
<td>on expr shift and goto 5</td>
</tr>
<tr>
<td>4. stmt  →  write expr</td>
<td>on expr shift and goto 6</td>
</tr>
<tr>
<td>expr  →  term</td>
<td>on term shift and goto 7</td>
</tr>
<tr>
<td>expr  →  expr addOp term</td>
<td>on term shift and reduce (pop 1 state, push term on input)</td>
</tr>
<tr>
<td>term  →  term addOp factor</td>
<td>on factor shift and reduce (pop 1 state, push factor on input)</td>
</tr>
<tr>
<td>factor  →  ( expr )</td>
<td>on factor shift and reduce (pop 1 state, push factor on input)</td>
</tr>
<tr>
<td>factor  →  id</td>
<td>on id shift and goto 8</td>
</tr>
<tr>
<td>factor  →  number</td>
<td>on number shift and reduce (pop 1 state, push factor on input)</td>
</tr>
<tr>
<td>5. stmt  →  id  →  expr</td>
<td>on expr shift and goto 9</td>
</tr>
<tr>
<td>6. stmt  →  write expr</td>
<td>on expr shift and goto 7</td>
</tr>
<tr>
<td>expr  →  term</td>
<td>on term shift and reduce (pop 1 state, push term on input)</td>
</tr>
<tr>
<td>expr  →  expr addOp term</td>
<td>on term shift and reduce (pop 1 state, push term on input)</td>
</tr>
<tr>
<td>term  →  term addOp factor</td>
<td>on factor shift and reduce (pop 1 state, push factor on input)</td>
</tr>
<tr>
<td>factor  →  ( expr )</td>
<td>on factor shift and reduce (pop 1 state, push factor on input)</td>
</tr>
<tr>
<td>factor  →  id</td>
<td>on id shift and goto 8</td>
</tr>
<tr>
<td>factor  →  number</td>
<td>on number shift and reduce (pop 1 state, push factor on input)</td>
</tr>
<tr>
<td>7. expr  →  term</td>
<td>on expr shift and reduce (pop 1 state, push expr on input)</td>
</tr>
<tr>
<td>term  →  term addOp factor</td>
<td>on addOp shift and goto 11</td>
</tr>
<tr>
<td>addOp  →  +</td>
<td>on + shift and reduce (pop 1 state, push addOp on input)</td>
</tr>
<tr>
<td>addOp  →  -</td>
<td>on - shift and reduce (pop 1 state, push addOp on input)</td>
</tr>
<tr>
<td>8. factor  →  ( expr )</td>
<td>on factor shift and reduce (pop 1 state, push factor on input)</td>
</tr>
<tr>
<td>factor  →  id</td>
<td>on id shift and reduce (pop 1 state, push factor on input)</td>
</tr>
<tr>
<td>factor  →  number</td>
<td>on number shift and reduce (pop 1 state, push factor on input)</td>
</tr>
<tr>
<td>9. stmt  →  id  →  expr</td>
<td>on expr shift and reduce (pop 1 state, push stmt on input)</td>
</tr>
<tr>
<td>expr  →  expr addOp term</td>
<td>on addOp shift and goto 10</td>
</tr>
<tr>
<td>addOp  →  +</td>
<td>on + shift and reduce (pop 1 state, push addOp on input)</td>
</tr>
<tr>
<td>addOp  →  -</td>
<td>on - shift and reduce (pop 1 state, push addOp on input)</td>
</tr>
<tr>
<td>10. expr  →  expr addOp term</td>
<td>on expr shift and reduce (pop 1 state, push expr on input)</td>
</tr>
<tr>
<td>term  →  factor</td>
<td>on term shift and reduce (pop 1 state, push term on input)</td>
</tr>
<tr>
<td>factor  →  ( expr )</td>
<td>on factor shift and reduce (pop 1 state, push factor on input)</td>
</tr>
<tr>
<td>factor  →  id</td>
<td>on id shift and goto 8</td>
</tr>
<tr>
<td>factor  →  number</td>
<td>on number shift and reduce (pop 1 state, push factor on input)</td>
</tr>
<tr>
<td>11. stmt  →  write expr</td>
<td>on stmt shift and goto 11</td>
</tr>
<tr>
<td>expr  →  expr addOp term</td>
<td>on expr shift and reduce (pop 1 state, push expr on input)</td>
</tr>
<tr>
<td>addOp  →  +</td>
<td>on + shift and reduce (pop 1 state, push addOp on input)</td>
</tr>
<tr>
<td>addOp  →  -</td>
<td>on - shift and reduce (pop 1 state, push addOp on input)</td>
</tr>
<tr>
<td>12. factor  →  ( expr )</td>
<td>on factor shift and reduce (pop 1 state, push factor on input)</td>
</tr>
<tr>
<td>expr  →  expr addOp term</td>
<td>on addOp shift and goto 10</td>
</tr>
<tr>
<td>addOp  →  +</td>
<td>on + shift and reduce (pop 1 state, push addOp on input)</td>
</tr>
<tr>
<td>addOp  →  -</td>
<td>on - shift and reduce (pop 1 state, push addOp on input)</td>
</tr>
<tr>
<td>13. expr  →  expr addOp term</td>
<td>on expr shift and reduce (pop 1 state, push expr on input)</td>
</tr>
<tr>
<td>term  →  term addOp factor</td>
<td>on term shift and reduce (pop 1 state, push term on input)</td>
</tr>
<tr>
<td>addOp  →  +</td>
<td>on + shift and reduce (pop 1 state, push addOp on input)</td>
</tr>
<tr>
<td>addOp  →  -</td>
<td>on - shift and reduce (pop 1 state, push addOp on input)</td>
</tr>
</tbody>
</table>

Figure 2.25: CFSA for the calculator grammar (Figure 2.24). Basis and closure items in each state are separated by a horizontal rule. Trivial reduce-only states have been eliminated by use of "shift and reduce" transitions (continued).
Figure 2.26: Pictorial representation of the CFSM of Figure 2.25. Symbol names have been abbreviated for clarity. Reduce actions are not shown.
### LR Parser Table

<table>
<thead>
<tr>
<th>Top-of-stack state</th>
<th>sl</th>
<th>s</th>
<th>e</th>
<th>t</th>
<th>f</th>
<th>ao</th>
<th>mo</th>
<th>Current input symbol</th>
<th>id</th>
<th>lit</th>
<th>r</th>
<th>w</th>
<th>:=</th>
<th>(</th>
<th>)</th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>$$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s2</td>
<td>b3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s3</td>
<td></td>
<td>s1</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>b5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>b2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s3</td>
<td></td>
<td>s1</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>b1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>b1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>s6</td>
<td>s7</td>
<td>b9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>s9</td>
<td>s7</td>
<td>b9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>s12</td>
<td>s7</td>
<td>b9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r7</td>
<td>r7</td>
<td>r7</td>
<td>r7</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2.27:** *SLR(1) parse table for the calculator language.* Table entries indicate whether to shift (s), reduce (r), or shift and then reduce (b). The accompanying number is the new state when shifting, or the production that has been recognized when (shifting and) reducing. Production numbers are given in Figure 2.24. Symbol names have been abbreviated for the sake of formatting. A dash indicates an error. An auxiliary table, not shown here, gives the left-hand side symbol and right-hand side length for each production.
## LR Parsing Example

<table>
<thead>
<tr>
<th>Parse stack</th>
<th>Input stream</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>( o )</td>
<td>read A read B ...</td>
<td>shift read</td>
</tr>
<tr>
<td>( 0 )</td>
<td>A read B ...</td>
<td>shift id(a) &amp; reduce by stmt — read id</td>
</tr>
<tr>
<td>( o )</td>
<td>stmt read B ...</td>
<td>shift stmt &amp; reduce by stmtList — stmt</td>
</tr>
<tr>
<td>( o )</td>
<td>stmtList read B ...</td>
<td>shift stmtList</td>
</tr>
<tr>
<td>( 0 )</td>
<td>B sum := ...</td>
<td>shift read</td>
</tr>
<tr>
<td>( o )</td>
<td>stmt sum := ...</td>
<td>shift id(b) &amp; reduce by stmt — read id</td>
</tr>
<tr>
<td>( 0 )</td>
<td>stmtList sum := ...</td>
<td>shift stmt &amp; reduce by stmtList — stmtList stmt</td>
</tr>
<tr>
<td>( o )</td>
<td>sum := B ...</td>
<td>shift stmtList</td>
</tr>
<tr>
<td>( 0 )</td>
<td>stmtList Sun := ...</td>
<td>shift stmtList</td>
</tr>
<tr>
<td>( o )</td>
<td>id(a) &amp; reduce by factor — id</td>
<td></td>
</tr>
<tr>
<td>( 0 )</td>
<td>factor := B ...</td>
<td>shift factor &amp; reduce by term — factor</td>
</tr>
<tr>
<td>( o )</td>
<td>term := B ...</td>
<td>shift factor &amp; reduce by term — factor</td>
</tr>
<tr>
<td>( 0 )</td>
<td>+ B write ...</td>
<td>shift term</td>
</tr>
<tr>
<td>( o )</td>
<td>expr := B = write ...</td>
<td>reduce by expr — term</td>
</tr>
<tr>
<td>( 0 )</td>
<td>+ B write ...</td>
<td>shift term</td>
</tr>
<tr>
<td>( 0 )</td>
<td>id(b) &amp; reduce by factor — id</td>
<td></td>
</tr>
<tr>
<td>( 0 )</td>
<td>addOp B = write ...</td>
<td>shift + &amp; reduce by addOp — +</td>
</tr>
<tr>
<td>( 0 )</td>
<td>id(b) &amp; reduce by factor — id</td>
<td></td>
</tr>
<tr>
<td>( 0 )</td>
<td>addOp = write ...</td>
<td>shift addOp</td>
</tr>
<tr>
<td>( 0 )</td>
<td>id(b) &amp; reduce by factor — id</td>
<td></td>
</tr>
<tr>
<td>( 0 )</td>
<td>addOp = write sum ...</td>
<td>shift addOp</td>
</tr>
<tr>
<td>( 0 )</td>
<td>id(b) &amp; reduce by factor — id</td>
<td></td>
</tr>
<tr>
<td>( 0 )</td>
<td>write sum ...</td>
<td>shift term</td>
</tr>
<tr>
<td>( 0 )</td>
<td>expr write sum ...</td>
<td>reduce by expr — expr addOp term</td>
</tr>
<tr>
<td>( 0 )</td>
<td>write sum ...</td>
<td>shift expr</td>
</tr>
<tr>
<td>( 0 )</td>
<td>stmt write sum ...</td>
<td>reduce by stmt — id := expr</td>
</tr>
<tr>
<td>( 0 )</td>
<td>stmtList write sum ...</td>
<td>shift stmt &amp; reduce by stmtList — stmtList</td>
</tr>
<tr>
<td>( 0 )</td>
<td>write sum ...</td>
<td>shift stmtList</td>
</tr>
<tr>
<td>( 0 )</td>
<td>stmtList write sum ...</td>
<td>shift stmtList</td>
</tr>
<tr>
<td>( 0 )</td>
<td>write sum ...</td>
<td>shift stmtList</td>
</tr>
<tr>
<td>( 0 )</td>
<td>expr write sum ...</td>
<td>reduce by expr — expr addOp term</td>
</tr>
<tr>
<td>( 0 )</td>
<td>write sum ...</td>
<td>shift expr</td>
</tr>
<tr>
<td>( 0 )</td>
<td>stmt write sum ...</td>
<td>reduce by stmt — id := expr</td>
</tr>
<tr>
<td>( 0 )</td>
<td>stmtList write sum ...</td>
<td>shift stmt &amp; reduce by stmtList — stmtList</td>
</tr>
<tr>
<td>( 0 )</td>
<td>write sum ...</td>
<td>shift stmtList</td>
</tr>
<tr>
<td>( 0 )</td>
<td>expr write sum ...</td>
<td>reduce by expr — expr addOp term</td>
</tr>
<tr>
<td>( 0 )</td>
<td>write sum ...</td>
<td>shift expres</td>
</tr>
<tr>
<td>( 0 )</td>
<td>stmt write sum ...</td>
<td>reduce by stmt — id := expr</td>
</tr>
<tr>
<td>( 0 )</td>
<td>stmtList write sum ...</td>
<td>shift stmt &amp; reduce by stmtList — stmtList</td>
</tr>
<tr>
<td>( 0 )</td>
<td>write sum ...</td>
<td>shift stmtList</td>
</tr>
<tr>
<td>( 0 )</td>
<td>expr write sum ...</td>
<td>reduce by expr — expr addOp term</td>
</tr>
<tr>
<td>( 0 )</td>
<td>write sum ...</td>
<td>shift expres</td>
</tr>
<tr>
<td>( 0 )</td>
<td>stmt write sum ...</td>
<td>reduce by stmt — id := expr</td>
</tr>
<tr>
<td>( 0 )</td>
<td>stmtList write sum ...</td>
<td>shift stmt &amp; reduce by stmtList — stmtList</td>
</tr>
<tr>
<td>( 0 )</td>
<td>write sum ...</td>
<td>shift stmtList</td>
</tr>
<tr>
<td>( 0 )</td>
<td>expr write sum ...</td>
<td>reduce by expr — expr addOp term</td>
</tr>
<tr>
<td>( 0 )</td>
<td>write sum ...</td>
<td>shift expres</td>
</tr>
<tr>
<td>( 0 )</td>
<td>stmt write sum ...</td>
<td>reduce by stmt — id := expr</td>
</tr>
<tr>
<td>( 0 )</td>
<td>stmtList write sum ...</td>
<td>shift stmt &amp; reduce by stmtList — stmtList</td>
</tr>
<tr>
<td>( 0 )</td>
<td>write sum ...</td>
<td>shift stmtList</td>
</tr>
<tr>
<td>( 0 )</td>
<td>expr write sum ...</td>
<td>reduce by expr — expr addOp term</td>
</tr>
<tr>
<td>( 0 )</td>
<td>write sum ...</td>
<td>shift expres</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>program.</td>
</tr>
</tbody>
</table>

Figure 2.29: Trace of a table-driven SLR(1) parse of the sum-and-average program. States in the parse stack are shown in boldface type. Symbols in the parse stack are for clarity only; they are not needed by the parsing algorithm. Parsing begins with the initial state of the CFSM (State 0) in the stack. It ends when we reduce by program — stmtList $\$$, uncovering State 0 again and pushing program onto the input stream.