Chapter 2 - Programming Language Syntax
Scanning

The main task of scanning is to **identify tokens**.
Pseudo-Code Scanner

We **skip** any initial white spaces
We **read** the next character
If it is a `/` we look at the next character
   if that is a `*` we have a comment;
      we skip forward through the terminating `*/`
   otherwise we return a `/` and reuse the look-ahead
If it is one of the one-character tokens `([],;=+- etc.)`
   we return that **token**

...
Pseudo-Code Scanner

- We could just turn this into real code and use that as the scanner, and that would be fine for small programs.
- However, for larger programs that must be correct a more formal approach is more appropriate.
Lexical Analysis

- Lexical analyzer: reads input characters and produces a sequence of tokens as output (nexttoken()).
  - Trying to understand each element in a program.
  - *Token*: a group of characters having a collective meaning.

```plaintext
const pi = 3.14159;
```

Token 1: (const, -)
Token 2: (identifier, ‘pi’)
Token 3: (=, -)
Token 4: (realnumber, 3.14159)
Token 5: (;, -)
Interaction of Lexical Analyzer with Parser

Source program

Lexical analyzer

parser

symbol table

Nexttoken()
Terminology

• Some terminology:
  ▫ **Token**: a group of characters having a collective meaning. A **lexeme** is a particular instant of a token.
    • E.g. token: identifier, lexeme: pi, etc.
  ▫ **pattern**: the rule describing how a token can be formed.
    • E.g: identifier: ([a-z][A-Z]) ([a-z][A-Z][0-9])*  

• Lexical analyzer does not have to be an individual phase.
  ▫ Having a separate phase simplifies the design and improves the efficiency and portability.
Lexical Analysis Issues

- Two issues in lexical analysis.
  - How to specify tokens (patterns)?
  - How to recognize the tokens given a token specification (how to implement the nexttoken() routine)?

- How to specify tokens:
  - All the basic elements in a language must be tokens so that they can be recognized.

```c
main() {
    int i, j;
    for (i=0; i<50; I++) {
        printf("I = %d", i);
    }
}
```

- There are not many types of tokens in a typical programming language: constant, identifier, reserved word, operator and misc. symbol.
Tokens

- Type of tokens in C++:
  - Constants:
    - char constants: ‘a’
    - string constants: “I=%d”
    - int constants: 50
    - float point constants: 3.14159
  - Identifiers: i, j, counter, ...
  - Reserved words: main, int, for, ...
  - Operators: +, =, ++, /, ...
  - Misc symbols: (, ), {, }, ...

- Tokens are specified by regular expressions.
Regular Expressions (RE)

- A RE consist of:
  - A character (e.g., “0”, “1”, ...)
  - The empty string (i.e., “ε”)
  - Two REs next to each other (e.g., “unsigned_integer digit”) to denote concatenation.
  - Two REs separated by “|” next to each other (e.g., “unsigned_integer | digit”) to denote one RE or the other.
  - An RE followed by “*” (called the **Kleene star**) to denote zero or more iterations of the RE.
  - Parentheses (in order to avoid ambiguity).
Regular Expressions (RE)

- **Definitions**
  - *alphabet*: a finite set of symbols. E.g. \{a, b, c\}
  - A *string* over an alphabet is a finite sequence of symbols drawn from that alphabet (sometimes a string is also called a sentence or a word).
  - A *language* is a set of strings over an alphabet.
  - Operation on languages (a set):
    - union of \(L\) and \(M\), \(L \cup M = \{s \mid s \text{ in } L \text{ or } s \text{ in } M\}\)
    - concatenation of \(L\) and \(M\), \(LM = \{st \mid s \text{ in } L \text{ and } t \text{ in } M\}\)
    - Kleene closure of \(L\), \(L^* = \bigcup_{i=0}^{\infty} L^i\)
    - Positive closure of \(L\), \(L^+ = \bigcup_{i=1}^{\infty} L^i\)
  - **Example:**
    - \(L = \{aa, bb, cc\}\), \(M = \{abc\}\)
Formal Definition of Regular Expressions

- Given an alphabet $\Sigma$
  - $\epsilon$ is a regular expression that denote $\{\epsilon\}$, the set that contains the empty string.
  - For each $a \in \Sigma$, $a$ is a regular expression denote $\{a\}$, the set containing the string $a$.
  - $r$ and $s$ are regular expressions denoting the language (set) $L(r)$ and $L(s)$. Then
    - $(r)\vert(s)$ is a regular expression denoting $L(r) \cup L(s)$
    - $(r)(s)$ is a regular expression denoting $L(r)L(s)$
    - $(r)^*$ is a regular expression denoting $(L(r))^*$

- Regular expression is defined together with the language it denotes.
Regular Expressions

- Numerical literals in Pascal may be generated by the following:

\[
\begin{align*}
\text{digit} & \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \\
\text{unsigned_integer} & \rightarrow \text{digit} \ \text{digit}^* \\
\text{unsigned_number} & \rightarrow \text{unsigned_integer} \ ( ( . \ \text{unsigned_integer} ) | \epsilon ) \\
& \quad ( ( ( e | E ) ( + | - | \epsilon ) \ \text{unsigned_integer} ) | \epsilon )
\end{align*}
\]
Regular Expressions

- A RE is **NEVER** defined in terms of itself!
  - Thus, REs cannot define recursive statements.
- The set of tokens that can be recognized by regular expressions is called a **regular set**.
Example

- Let $\Sigma = \{a, b\}$
  
  $a|b$
  
  $(a|b)(a|b)$
  
  $a^*$
  
  $(a|b)^*$
  
  $a|a^*b$

- We assume that ‘*’ has the highest precedence and is left associative. Concatenation has second highest precedence and is left associative and ‘|’ has the lowest precedence and is left associative
  
  $(a)|(b)^*(c)) = a|b^*c$
Regular Definition

- Gives names to regular expressions to construct more complicate regular expressions.
  
  \[
  \begin{align*}
  d_1 & \rightarrow r_1 \\
  d_2 & \rightarrow r_2 \\
  \vdots \\
  d_n & \rightarrow r_n
  \end{align*}
  \]

- Example:
  
  \[
  l\mid s\mid l\rightarrow A \mid B \mid C \mid \ldots \mid Z \mid a \mid b \mid \ldots \mid z
  \]
  
  \[
  d\mid s\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
  \]
  
  \[
  i\mid d\rightarrow letter(letter\mid digit)^*
  \]

- Cannot use recursive definitions
  
  \[
  d\mid g\rightarrow digit\mid digits
  \]
Regular Expression for Tokens in C++

- Constants:
  - char constants: ‘” any_char ‘”
  - string constants: “I=%d”
    - ‘” [^”]* ‘” – not quite right.
  - int constants: 50 -- digit (digit)*
  - float point constants 50.0 -- digit (digit)* ‘.’ digit (digit) *
- Identifiers: letter (letter | digit | ‘_’) *
- Reserved words: ‘m”a”i”n’ for main
- Operators: ‘+’ for +, and ‘+’ ‘+’ for ++
- Misc symbols: ‘(‘ for (
Regular Expression for Tokens in C++

- A signed number in C++:
  3.1416  +2006  1e-010  -3.14159  2006.00000  0.00000
  3.14159e+000  2.00600e+003  -1.00000E-010

Num → digit(digit)*
NUMBER → (± | − | ε) (Num | Num . Num) (ε|(ε|E)(± | − | ε)Num )
Two Issues in Lexical Analysis

• Specifying tokens (regular expression).
• Identifying tokens specified by regular expression.
  ▫ How do we do this?
How to Recognize Tokens Specified by Regular Expressions?

- A recognizer for a language is a program that takes a string $x$ as input and answers “yes” if $x$ is a sentence of the language and “no” otherwise.
  - In the context of lexical analysis, given a string and a regular expression, a recognizer of the language specified by the regular expression answer “yes” if the string is in the language.
  - How to recognize regular expression `int`?

```
0 -----> 1 -----> 2 -----> 3
  i      n      t
error
All other characters
```
How to Recognize Tokens Specified by Regular Expressions?

- How to recognize regular expression `int | for`?

- A regular expression can be compiled into a recognizer (automatically) by constructing a finite automata which can be deterministic or non-deterministic.
Deterministic finite automaton (DFA)

- Every regular set can be defined by using **deterministic finite automaton (DFA)**.
- DFAs are turning machines that have a finite number of states and deterministically move between states.
- For example, the DFA for $0^*10^*1(0|1)^*$ is

![DFA Diagram]

- **S**: Start state
- **Intermediate state**
- **End state (Double Circle)**
Constructing DFAs

A DFA can be constructed from a RE via two steps.
1. Construct a nondeterministic finite automaton (NFA) from the RE.
2. Construct a DFA from the NFA.
3. Minimize the DFA
What is an NFA?

- An NFA is similar to a DFA, except that state transitions are nondeterministic.
- This nondeterminism is encapsulated via the \textit{epsilon transition} (written as $\epsilon$).

The $\epsilon$ transitions imply that either transition can be taken with any (or no) input.
Non-deterministic Finite Automata (NFA)

- A non-deterministic finite automata (NFA) is a mathematical model that consists of: a 5-tuple \((Q, \Sigma, \delta, q_0, F)\)
  - a set of states \(Q\)
  - a set of input symbols \(\Sigma\)
  - a transition \(\delta\) function that maps state-symbol pairs to sets of states.
  - A state \(q_0\) that is distinguished as the start (initial) state
  - A set of states \(F\) distinguished as accepting (final) states.
- An NFA accepts an input string \(x\) if and only if there is some path in the transition graph from the start state to some accepting state (after consuming \(x\)).
Finite State Machines = Regular Expression Recognizers

relop → < | <= | <> | > | >= | =

id → letter ( letter | digit )*

letter or digit

return(gettoken(), install_id())
Non-deterministic Finite Automata (NFA)

• An NFA is non-deterministic in that (1) same character can label two or more transitions out of one state (2) empty string can label transitions.
• For example, here is an NFA that recognizes the language $\{ \}^*$

![NFA Diagram]

• An NFA can easily implemented using a transition table, which can be used in the language recognizer.

<table>
<thead>
<tr>
<th>State</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0, 1}</td>
<td>{0}</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>{2}</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>{3}</td>
</tr>
</tbody>
</table>
The Four RE Rules and NFA

- Rule 1--**Base case**: “a”
The Four RE Rules and NFA

- Rule 2—Concatenation: “AB”
The Four RE Rules and NFA

- Rule 3--**Alternation**: “a|b”
The Four RE Rules and NFA

- Rule 4--**Kleene Closure**: “a*”
Constructing a DFA From an NFA

- Construct the DFA by “collapsing” the states of an NFA.
- Three steps
  1. Identify set of states that can be reached from the start state via epsilon-transitions and make this one state.
  2. For a given DFA state (which is a set of NFA states) consider each possible input and combine the resulting NFA states into one DFA state.
  3. Repeat Step 2 until all states have been added.
An Example: $0|(1^*) 00*$

NFA

DFA
Minimize via Partitioning

- First, partition states into final and non-final.
- Second, determine the effect of the state transition based on what partition the transition goes to.
- Third, create new partition for those states that have different transitions.
- Fourth, repeat.
Minimize via Partitioning

DFA

S, A, B, F

C, G, E, F

B, D, F

G, E, F

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SABF</td>
<td>X-2</td>
<td>X-1</td>
</tr>
<tr>
<td>BDF</td>
<td>X-2</td>
<td>X-1</td>
</tr>
<tr>
<td>CGEF</td>
<td>X-2</td>
<td>N/A</td>
</tr>
<tr>
<td>GEF</td>
<td>X-2</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Minimize via Partitioning

![Diagram showing states and transitions]

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SABF</td>
<td>X-2</td>
<td>X-1</td>
</tr>
<tr>
<td>BDF</td>
<td>X-2</td>
<td>X-1</td>
</tr>
<tr>
<td>CGEF</td>
<td>X-2</td>
<td>N/A</td>
</tr>
<tr>
<td>GEF</td>
<td>X-2</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Another Example of Minimization

- Consider the following DFA.
- **Accepting** states are red.
- **Non-accepting** states are blue.
- Are any states really the same?
Another Example

- $S_2$ and $S_7$ are really the same:
  - Both Final states
  - Both go to $S_6$ under input $b$.
  - Both go to $S_3$ under an $a$.
- $S_0$ and $S_5$ really the same. Why?
- We say each pair is equivalent.

- Are there any other equivalent states?
- We can merge equivalent states into 1 state.
Partitioning Algorithm

• First step:
  - Divide the set of states into final and non-final states.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₀</td>
<td>S₁</td>
<td>S₄</td>
</tr>
<tr>
<td>S₁</td>
<td>S₅</td>
<td>S₂</td>
</tr>
<tr>
<td>S₃</td>
<td>S₃</td>
<td>S₃</td>
</tr>
<tr>
<td>S₄</td>
<td>S₁</td>
<td>S₄</td>
</tr>
<tr>
<td>S₅</td>
<td>S₁</td>
<td>S₄</td>
</tr>
<tr>
<td>S₆</td>
<td>S₃</td>
<td>S₇</td>
</tr>
<tr>
<td>S₂</td>
<td>S₃</td>
<td>S₆</td>
</tr>
<tr>
<td>S₇</td>
<td>S₃</td>
<td>S₆</td>
</tr>
</tbody>
</table>
Partitioning Algorithm

- Next step:
  - See if states in each partition each go to the same partition.
  - $S_1$ & $S_6$ are different from the rest of the states in Partition I (but like each other).
  - We will move them to a new partition.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0$</td>
<td>$S_1$ (I)</td>
<td>$S_4$ (I)</td>
</tr>
<tr>
<td>$S_1$</td>
<td>$S_5$ (I)</td>
<td>$S_2$ (II)</td>
</tr>
<tr>
<td>$S_3$</td>
<td>$S_3$ (I)</td>
<td>$S_3$ (I)</td>
</tr>
<tr>
<td>$S_4$</td>
<td>$S_1$ (I)</td>
<td>$S_4$ (I)</td>
</tr>
<tr>
<td>$S_5$</td>
<td>$S_1$ (I)</td>
<td>$S_4$ (I)</td>
</tr>
<tr>
<td>$S_6$</td>
<td>$S_3$ (I)</td>
<td>$S_7$ (II)</td>
</tr>
<tr>
<td>$S_2$</td>
<td>$S_3$ (I)</td>
<td>$S_6$ (I)</td>
</tr>
<tr>
<td>$S_7$</td>
<td>$S_3$ (I)</td>
<td>$S_6$ (I)</td>
</tr>
</tbody>
</table>
Partitioning Algorithm

- New partitions are created.
- States are updated to match their new partition.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0$</td>
<td>$S_1$ (III)</td>
<td>$S_4$ (I)</td>
</tr>
<tr>
<td>$S_3$</td>
<td>$S_3$ (I)</td>
<td>$S_3$ (I)</td>
</tr>
<tr>
<td>$S_4$</td>
<td>$S_1$ (III)</td>
<td>$S_4$ (I)</td>
</tr>
<tr>
<td>$S_5$</td>
<td>$S_1$ (III)</td>
<td>$S_4$ (I)</td>
</tr>
<tr>
<td>$S_1$</td>
<td>$S_5$ (I)</td>
<td>$S_2$ (II)</td>
</tr>
<tr>
<td>$S_6$</td>
<td>$S_3$ (I)</td>
<td>$S_7$ (II)</td>
</tr>
<tr>
<td>$S_2$</td>
<td>$S_3$ (I)</td>
<td>$S_6$ (III)</td>
</tr>
<tr>
<td>$S_7$</td>
<td>$S_3$ (I)</td>
<td>$S_6$ (III)</td>
</tr>
</tbody>
</table>
Partitioning Algorithm

- Repeat the process.
- See if states in each partition each go to the same partition.
- In Partition I, $S_3$ goes to a different partition from $S_0$, $S_5$ and $S_4$.

- Move $S_3$ to its own partition.
- States $S_2$, $S_6$, and $S_7$ changes respectively.
Partitioning Algorithm

- Now $S_6$ goes to a different partition on an $a$ from $S_1$.
- $S_6$ gets its own partition.
- We now have 5 partitions.
- Note changes in $S_2$ and $S_7$.

- All states within each of the 5 partitions are identical.
- We can call the states I, II III, IV and V.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0$</td>
<td>$S_1$ (III)</td>
<td>$S_4$ (I)</td>
</tr>
<tr>
<td>$S_4$</td>
<td>$S_1$ (III)</td>
<td>$S_4$ (I)</td>
</tr>
<tr>
<td>$S_5$</td>
<td>$S_1$ (III)</td>
<td>$S_4$ (I)</td>
</tr>
<tr>
<td>$S_3$</td>
<td>$S_3$ (IV)</td>
<td>$S_3$ (IV)</td>
</tr>
<tr>
<td>$S_1$</td>
<td>$S_5$ (I)</td>
<td>$S_2$ (II)</td>
</tr>
<tr>
<td>$S_6$</td>
<td>$S_3$ (IV)</td>
<td>$S_7$ (II)</td>
</tr>
<tr>
<td>$S_2$</td>
<td>$S_3$ (IV)</td>
<td>$S_6$ (V)</td>
</tr>
<tr>
<td>$S_7$</td>
<td>$S_3$ (IV)</td>
<td>$S_6$ (V)</td>
</tr>
</tbody>
</table>
# Partitioning Algorithm

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>III</td>
<td>I</td>
</tr>
<tr>
<td>II</td>
<td>IV</td>
<td>V</td>
</tr>
<tr>
<td>III</td>
<td>I</td>
<td>II</td>
</tr>
<tr>
<td>IV</td>
<td>IV</td>
<td>IV</td>
</tr>
<tr>
<td>V</td>
<td>IV</td>
<td>II</td>
</tr>
</tbody>
</table>

![Diagram showing the partitioning algorithm](image)
Scanning

- Recall scanner is responsible for
  - Tokenizing source.
  - Removing comments.
  - Dealing with *pragmas* (i.e., significant comments).
  - Saving text of identifiers, numbers, strings.
  - Saving source locations (file, line, column) for error messages.
Pragmas

- **Pragmas** are “comments” that provide direction for the compiler.
  - `#pragma omp parallel for`
  - `#pragma GCC ivdep`
- For example, “Variable $x$ is used a lot, keep it in memory if possible.”
- These are handled by the parser since this makes the grammar much simpler.
Scanning

- Suppose we are building an ad-hoc (hand-written) scanner for Pascal:
  - We read the characters one at a time with look-ahead.
- If it is one of the one-character tokens \{ ( ) [ ] < > , ; = + - etc \} we announce that token.
- If it is a ., we look at the next character.
  - If that is a dot, we announce “.”
  - Otherwise, we announce “.” and reuse the look-ahead.
Scanning

• If it is a <, we look at the next character..
  ▫ If that is a = we announce <=.
  ▫ Otherwise, we announce < and reuse the look-ahead, etc.

• If it is a letter, we keep reading letters and digits and maybe underscores until we can't anymore.
  ▫ Then we check to see if it is a reserve word.

• If it is a digit, we keep reading until we find a non-digit.
  ▫ If that is not a “.” we announce an integer.
  ▫ Otherwise, we keep looking for a real number.
  ▫ If the character after the “.” is not a digit we announce an integer and reuse the “.” and the look-ahead.
Scanning

- Pictorial representation of a scanner for calculator tokens, in the form of a finite automaton
Scanning

- This is a deterministic finite automaton (DFA)
  - **Lex, scangen**, etc. build these things automatically from a set of regular expressions.
  - Specifically, they construct a machine that accepts the language:
    - identifier | int const
    - | real const | comment | symbol | ...
Scanning

- We run the machine over and over to get one token after another.
  - Nearly universal rule:
    - Always take the longest possible token from the input thus `foobar` is `foobar` and never `f` or `foo` or `foob`
    - more to the point, `3.14159` is a real const and never `3.`, `.`, and `14159`.

- Regular expressions "generate" a regular language; DFAs "recognize" it.
Scanning

- Scanners tend to be built three ways:
  - Ad-hoc.
  - Semi-mechanical pure DFA.
    (usually realized as nested case statements)
  - Table-driven DFA.

- Ad-hoc generally yields the fastest, most compact code by doing lots of special-purpose things, though good automatically-generated scanners come very close.
Scanning

- Writing a pure DFA as a set of nested case statements is a surprisingly useful programming technique.
  - Though it's often easier to use perl, awk, sed.
  - For details see Figure 2.11.
- Table-driven DFA is what lex and scangen produce
  - lex (flex) in the form of C code.
  - Scangen in the form of numeric tables and a separate driver. (for details see Figure 2.12)
Scanning

- Note that the rule about longest-possible tokens means you return only when the next character can't be used to continue the current token.
  - The next character will generally need to be saved for the next token.
- In some cases, you may need to peek at more than one character of look-ahead in order to know whether to proceed.
  - In Pascal, for example, when you have a 3 and you see a dot:
    - Do you proceed (in hopes of getting 3.14, a real number)?
    - or
    - Do you stop (in fear of getting 3..5, an array range)?
- It is possible to maintain a DFA for keywords, but the number of states would be even larger! So, they are handled as exceptions to the rule.
Scanning

• In messier cases, you may not be able to get by with any fixed amount of look-ahead. In Fortran, for example, we have
  \[
  \text{DO 5 I = 1,25 loop}
  \]
  \[
  \text{DO 5 I = 1.25 assignment}
  \]
• Here, we need to remember we were in a potentially final state, and save enough information that we can back up to it, if we get stuck later.
Parsing

The main task of parsing is to **identify the syntax**.
Context-Free Grammars

- **Context-Free Grammars (CFGs)** are similar to REs except that they can handle **recursion**.

  \[
  expr \rightarrow \text{id} \mid \text{number} \mid - \ expr \mid ( \ expr \ ) \mid expr \ op \ expr
  \]

  \[
  op \rightarrow + \mid - \mid * \mid /
  \]

  ▫ Each rule is called a **production**.
  ▫ One of the **nonterminals**, usually the first one, is called the **start symbol**, and it defines the construct defined by the grammar.
  ▫ **Non-terminals** are symbols that are defined by the CFG and can appear on both the **left** and **right** side of “→”.

\[
expr \rightarrow \text{id} \mid \text{number} \mid - \ expr \mid ( \ expr \ ) \mid expr \ op \ expr
\]

\[
op \rightarrow + \mid - \mid * \mid /
\]
Backus-Naur Form

- The notation for context-free grammars (CFG) is sometimes called **Backus-Naur Form** (BNF)
- A CFG consists of
  - A set of *terminals* $T$.
  - A set of *non-terminals* $N$.
  - A start *symbol* $S$ (a non-terminal).
  - A set of *productions*.
Extended Backus-Naur Form

- Extended BNF includes the Kleene star, parentheses, and the Kleene Plus (+), which stands for “one or more iterations.”

\[
\begin{align*}
  id\_list & \rightarrow \text{id} (, \text{id} )^* \\
  id\_list & \rightarrow \text{id} \\
  id\_list & \rightarrow id\_list, \text{id}
\end{align*}
\]
Derivation

• A derivation is “a series of replacement operations that derive a string of terminals from the start symbol.”

• Derivation of \( \text{slope} \times x + \text{intercept} \)

\[
\begin{align*}
expr & \Rightarrow expr \ op \ expr \\
& \Rightarrow expr \ op \ id \\
& \Rightarrow expr + id \\
& \Rightarrow expr \ op \ expr + id \\
& \Rightarrow expr \ op \ id + id \\
& \Rightarrow expr \ op \ id + id \\
& \Rightarrow id \* id + id
\end{align*}
\]

\( \Rightarrow \) denotes “derived from.”

• This derivation replaces the rightmost non-terminal. Derivations with this behavior are called (rightmost or canonical derivation).
A parse tree is the graphical representation of the derivation.

This parse tree constructs the formula: $(\text{slope} \times x) + \text{intercept}$

- This is a rightmost derivation.
Parse Tree (Ambiguous)

- This grammar, is ambiguous and can construct the following parse tree.
- This parse tree constructs the formula \( \text{slope} \times (x + \text{intercept}) \) which is \textit{not equal} to \( \text{slope} \times x + \text{intercept} \)
  - This is a \textit{leftmost} derivation.
Disambiguating Grammar

- The problem with our original grammar was that we did not fully express the grammatical structure (i.e., associativity and precedence).
- To create an unambiguous grammar, we need to fully specify the grammar.

\[
\begin{align*}
expr & \rightarrow term \mid expr \ add\_op \ term \\
term & \rightarrow factor \mid term \ mult\_op \ factor \\
factor & \rightarrow id \mid number \mid -factor \mid (expr) \\
add\_op & \rightarrow + \mid - \\
mult\_op & \rightarrow * \mid / 
\end{align*}
\]
3 + 4 * 5

```
expr
  └── add_op
     ├── term
     │   └── factor
     │       └── number(3)
     └── term
          └── factor
               └── number(4)
  └── term
       └── mul_op
            └── factor
                 └── number(5)
```
Compare 3+4 and 3*4
Compare 3+4 and 3*4

• How can you derive these trees by examining one character at a time?
• In order to derive these trees, the first character that we need to examine is the math symbol.
• To parse these “sentences,” we need to search through them to find the math symbols.
• Then we need to sort out the multiplication from the addition.
Parsing

- By analogy to RE and DFAs, a context-free grammar (CFG) is a *generator* for a context-free language (CFL).
  - A parser is a language recognizer.
- There is an infinite number of grammars for every context-free language.
  - Not all grammars are created equal.
Parsing

- It turns out that for any CFG we can create a parser that runs in $O(n^3)$ time.
- There are two well-known parsing algorithms that permit this:
  - Early's algorithm
  - Cooke-Younger-Kasami (CYK) algorithm
- $O(n^3)$ time is clearly unacceptable for a parser in a compiler - too slow.
Parsing

- There are two types of grammars that can be parsed in linear time, i.e., $O(n)$.
- **LL**: Left-to-right, Left-most derivation
- **LR**: Left-to-right, Right-most derivation
- LL parsers are also called 'top-down', or 'predictive' parsers & LR parsers are also called 'bottom-up', or 'shift-reduce' parsers.
- There are several important sub-classes of LR parsers
  - SLR
  - LALR
- We won't be going into detail on the differences between them.
Parsing

- Every LL(1) grammar is also LR(1), though right recursion in production tends to require very deep stacks and complicates semantic analysis.
- Every CFL that can be parsed deterministically has an SLR(1) grammar (which is LR(1)).
- Every deterministic CFL with the prefix property (no valid string is a prefix of another valid string) has an LR(0) grammar.
Top-down

- **LL parsers** are top-down, i.e., they identify the nonterminals first and terminals second.
- **LL grammars** are grammars that can be parsed by an LL parser.

- Suppose for the following grammar:
  
  \[
  \begin{align*}
  id\_list & \rightarrow id\ id\_list\_tail \\
  id\_list\_tail & \rightarrow \,,id\ id\_list\_tail \\
  id\_list\_tail & \rightarrow ;
  \end{align*}
  \]

- We want to parse: A, B, C;
Top-down

- LL parsers are **predictive** because they “predict” the next state.
- After \( id(A) \) is “discovered”, the next state is “predicted as \( id\_list\_tail \).
- Notice that token are placed in the tree from the **leftmost** to **rightmost**.
Bottom-up

- **LR parsers** are bottom-up, i.e., they discover the terminals first and non-terminals second.
- **LR grammars** are grammars that can be parsed by an LR parser.
- All **LL grammars** are **LR grammars** but not vice versa.

Suppose for the following grammar:

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

- We want to parse: A, B, C;
Bottom-up

1. id (A)
2. id (A),
6. id (A), id (B), id (C);
7. id (A), id (B), id (C) id_list_tail;

2016/9/25
LR parsers are called **shift parsers** because they “shift” the states.

Notice that tokens are added to the tree from the **rightmost** to the **left-most**.
Bottom-up

- Some times you LL and LR parsers written as $LL(n)$ and $LR(n)$ to denote the parser needs to look ahead $n$ tokens.
- The problem with this grammar is that it can require an arbitrarily large number of terminals to be “shifted” before placing them into the tree.
A Better Bottom-up Grammar

- This grammar limits the number of “suspended” non-terminals.
  
  \[
  id\_list \rightarrow id\_list\_prefix ; \\
  id\_list\_prefix \rightarrow id\_list\_prefix, id \\
  id\_list\_prefix \rightarrow id
  \]

- However, it cannot be parsed by LL (top-down) parser. Since when the parser discovers an “id” it does not “know” the number of “id\_list\_prefixs”.
A Better Bottom-up Grammar

- Both of these are valid breakdowns, but we don’t know which one. Therefore, is NOT a valid LL grammar, but it is a valid LR grammar.
LL(1) Grammar

program → stmt_list $$
stmt_list → stmt stmt_list | ε
stmt → id:= expr | read id | read expr
expr → term term_tail
term_tail → add_op term term_tail | ε
term → factor factor_tail
factor_tail → mult_op factor factor_tail | ε
factor → (expr) | id | literal
add_op → +| -
mult_op → *| /
LL(1) Grammar

- This grammar (for the calculator) unlike the previous calculator grammar is an LL grammar, because when an “id” is encountered we know exactly where it belongs.
- Like the bottom-up grammar, this one captures associativity and precedence, but most people don't find it as pretty
  - For one thing, the operands of a given operator aren't in a RHS together!
  - The simplicity of the parsing algorithm makes up for this weakness.
- How do we parse a string with this grammar?
  - Build the parse tree incrementally.
- Try parsing \( c := 2*A+B \)
LL Parsing

- Example (average program)
  
  ```
  read A
  read B
  sum := A + B
  write sum
  write sum / 2
  ```

- We start at the top and predict needed productions on the basis of the current left-most non-terminal in the tree and the current input token.
LL Parsing

• Parse tree for the average program (Figure 2.17)
LL Parsing

- Table-driven LL parsing: you have a big loop in which you repeatedly look up an action in a two-dimensional table based on current leftmost non-terminal and current input token.

- The actions are:
  1. Match a terminal.
  2. Predict a production.
  3. Announce a syntax error.
## LL Parsing

- **LL(1) parse table for parsing for calculator language**

<table>
<thead>
<tr>
<th>Top-of-stack nonterminal</th>
<th>id</th>
<th>number</th>
<th>read</th>
<th>Current input token</th>
<th>write</th>
<th>:</th>
<th>( )</th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>$$</th>
</tr>
</thead>
<tbody>
<tr>
<td>program</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td></td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>stmt_list</td>
<td>2</td>
<td>-</td>
<td>2</td>
<td></td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>stmt</td>
<td>4</td>
<td>-</td>
<td>5</td>
<td></td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>expr</td>
<td>7</td>
<td>7</td>
<td>-</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>term_tail</td>
<td>9</td>
<td>-</td>
<td>9</td>
<td></td>
<td>9</td>
<td>-</td>
<td>-</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>term</td>
<td>10</td>
<td>10</td>
<td>-</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>factor_tail</td>
<td>12</td>
<td>-</td>
<td>12</td>
<td></td>
<td>12</td>
<td>-</td>
<td>-</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>factor</td>
<td>14</td>
<td>15</td>
<td>-</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>13</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>add_op</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>
<td>-</td>
</tr>
<tr>
<td>mult_op</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>
<td>-</td>
</tr>
</tbody>
</table>
LL Parsing

initial stack
predict (1) stmt_list $$
predict (2) stmt stmt_list $$
match id
predict (2) id stmt_list $$
match id
predict (5) stmt_list $$
match read
predict (2) stmt $$
match id
predict (5) id stmt_list $$
match read
predict (2) stmt_list $$
match :=
predict (7) term term’ stmt_list $$
predict (10) factor factor’ term’ stmt_list $$
predict (14) id factor’ term’ stmt_list $$
match id
predict (12) term’ stmt_list $$
predict (8) addOp term term’ stmt_list $$

program
read A read..
read A read..
read A read..
read A read.. 
A read B sum..
read B sum :=.. 
read B sum :=.. 
read B sum :=.. 
sum := A + B.. := A + B write.. A + B write.. A + B write.. A + B write.. A + B write.. A + B write.. A + B write.. + B write sum.. + B write sum.. + B write sum.. + B write sum..
LL Parsing

- To keep track of the left-most non-terminal, you push the as-yet-unseen portions of productions onto a stack.
  - For details see Figure 2.20
- The key thing to keep in mind is that the stack contains all the stuff you expect to see between now and the end of the program.
  - What you predict you will see.
LL Parsing

- Problems trying to make a grammar LL(1)
  - Left recursion
    - Example:
      \[
      \text{id} \_ \text{list} \rightarrow \text{id} \mid \text{id} \_ \text{list} \{ \text{, id} \}
      \]
      equivalently
      \[
      \text{id} \_ \text{list} \rightarrow \text{id id} \_ \text{list} \_ \text{tail}
      \]
      \[
      \text{id} \_ \text{list} \_ \text{tail} \rightarrow \{ \text{, id id} \_ \text{list} \_ \text{tail} \mid \varepsilon \}
      \]
    - We can get rid of all left recursion mechanically in any grammar.
LL Parsing

• Problems trying to make a grammar LL(1)
  ▫ Common prefixes: another thing that LL parsers can't handle
    • Solved by "left-factoring"
    • Example:
      stmt → id := expr | id ( arg_list )
      equivalently
      stmt → id id_stmt_tail
      id_stmt_tail → := expr | ( arg_list)
    • We can eliminate left-factor mechanically.
LL Parsing

- Note that eliminating left recursion and common prefixes does NOT make a grammar LL.
  - There are infinitely many non-LL LANGUAGES, and the mechanical transformations work on them just fine.
  - The few that arise in practice, however, can generally be handled with kludges.
LL Parsing

- Problems trying to make a grammar LL(1)
  - The "dangling else" problem prevents grammars from being LL(1) (or in fact LL(k) for any k).
  - The following natural grammar fragment is ambiguous (Pascal):
    ```
    stmt → if cond then_clause else_clause | other_stuff
    then_clause → then stmt
    else_clause → else stmt | epsilon
    ```
LL Parsing

• The less natural grammar fragment can be parsed bottom-up but not top-down:

```plaintext
stmt → balanced_stmt | unbalanced_stmt
balanced_stmt → if cond then balanced_stmt
                         else balanced_stmt
                         | other_stuff
unbalanced_stmt → if cond then stmt
                         | if cond then balanced_stmt
                         else unbalanced_stmt
```
LL Parsing

- The usual approach, whether top-down OR bottom-up, is to use the ambiguous grammar together with a *disambiguating rule* that says:
  - “else” goes with the closest “then” or
  - More generally, the first of two possible productions is the one to predict (or reduce).
LL Parsing

• Better yet, languages (since Pascal) generally employ explicit end-markers, which eliminate this problem.
• In Modula-2, for example, one says:
  
  ```
  if A = B then
    if C = D then E := F end
  else
    G := H
  end
  ```

• Ada says 'end if'; other languages say 'fi'.
LL Parsing

• One problem with end markers is that they tend to bunch up. In Pascal you say:

```pascal
if A = B then ...
else if A = C then ...
else if A = D then ...
else if A = E then ...
else ...;
```

• With end markers this becomes:

```pascal
if A = B then ...
else if A = C then ...
else if A = D then ...
else if A = E then ...
else ...;
end; end; end; end; end;
```
Recursive Descent & LL Parse Table

- There are two ways to build an LL parse table:
  - **Recursive Descent**, which is a recursive program with case statements that correspond to each one-to-one to nonterminals.
  - **LL Parse Table**, which consist of an iterative driver program and a table that contains all of the nonterminals.
LL(1) Grammar Revisited

\[
\begin{align*}
\text{program} & \rightarrow \text{stmt\_list} \ \epsilon \\
\text{stmt\_list} & \rightarrow \text{stmt\_list} \ | \ \epsilon \\
\text{stmt} & \rightarrow \text{id:= expr} \ | \ \text{read id} \ | \ \text{read expr} \\
\text{expr} & \rightarrow \text{term\_tail} \\
\text{term\_tail} & \rightarrow \text{add\_op term term\_tail} \ | \ \epsilon \\
\text{term} & \rightarrow \text{factor factor\_tail} \\
\text{factor\_tail} & \rightarrow \text{mul\_op factor factor\_tail} \ | \ \epsilon \\
\text{factor} & \rightarrow (\ \text{expr}) \ | \ \text{id} \ | \ \text{literal} \\
\text{add\_op} & \rightarrow + \ | \ - \\
\text{mul\_op} & \rightarrow * \ | \ / 
\end{align*}
\]
Recursive Descent

procedure program()
    case in_tok of
        id, read, write, $$:
            stmt_list()
            match($$)
        else
            return error
    end

procedure stmt_list()
    case in_tok of
        id, read, write:
            stmt(); stmt_list();
        $$:
            skip
        else
            return error
    end

procedure stmt()
    case in_tok of
        id:
            match(id); match(:=); expr()
        read:
            match(read); match(id)
        write:
            match(write); expr()
        else
            return error
    end

procedure match(expec)
    if in_tok = expec
        consume in_tok
    else
        return error
Recursive Descent

- **in_tok** is a global variable that is the current token.
- **consume** changes in_tok to the next token.
- The question is how do we label the case statements?
First, Follow, and Predict

- Three functions allow us to label the branches
- **FIRST(a)**: The terminals (and ε) that can be the first tokens of the non-terminal symbol a.
- **FOLLOW(A)**: The terminals that can follow the terminal or nonterminal symbol A.
- **PREDICT(A → a)**: The terminals that can be the first tokens as a result of the production A → a.
LL Parsing

- The algorithm to build predict sets is tedious (for a "real" sized grammar), but relatively simple:
- It consists of three stages:
  - (1) compute FIRST sets for symbols
  - (2) compute FOLLOW sets for non-terminals (this requires computing FIRST sets for some strings)
  - (3) compute predict sets or table for all productions
First Set Rules

1. If X is a terminal then First(X) is just X!
2. If there is a Production X → ε then add ε to first(X).
3. If there is a Production X → Y1Y2..Yk then add first(Y1Y2..Yk) to first(X).
4. First(Y1Y2..Yk) is either:
   1. First(Y1) (if First(Y1) doesn't contain ε)
   2. OR (if First(Y1) does contain ε) then First (Y1Y2..Yk) is everything in First(Y1) <except for ε> as well as everything in First(Y2..Yk)
   3. If First(Y1) First(Y2)..First(Yk) all contain ε then add ε to First(Y1Y2..Yk) as well.
An Example Grammar

E → TE’
E’ → +TE’ | ε
T → FT’
T’ → *FT’ | ε
F → (E)
F → id

<table>
<thead>
<tr>
<th>Non-Terminals</th>
<th>First set</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td></td>
</tr>
<tr>
<td>E’</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
</tr>
<tr>
<td>T’</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>
An Example Grammar

E → TE’
E’ → +TE’
E’ → ε
T → FT’
T’ → *FT’
T’ → ε
F → (E)
F → id

• Applying Rule 2.
An Example Grammar

\[
\begin{align*}
  E & \rightarrow TE' \\
  E' & \rightarrow +TE' \\
  T & \rightarrow FT' \\
  T' & \rightarrow *FT' \\
  F & \rightarrow (E) \\
  F & \rightarrow \text{id}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Non-Terminals</th>
<th>First set</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td></td>
</tr>
<tr>
<td>E'</td>
<td>ε, +</td>
</tr>
<tr>
<td>T</td>
<td></td>
</tr>
<tr>
<td>T'</td>
<td>ε, *</td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

- Applying Rule 3.
An Example Grammar

E → TE’

T → FT’

F → (E)
F → id

• Applying Rule 3.

<table>
<thead>
<tr>
<th>Non-Terminals</th>
<th>First set</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td></td>
</tr>
<tr>
<td>E’</td>
<td>ε, +</td>
</tr>
<tr>
<td>T</td>
<td></td>
</tr>
<tr>
<td>T’</td>
<td>ε, *</td>
</tr>
<tr>
<td>F</td>
<td>id, (</td>
</tr>
</tbody>
</table>
An Example Grammar

\[ E \rightarrow TE' \]

\[ T \rightarrow FT' \]

<table>
<thead>
<tr>
<th>Non-Terminals</th>
<th>First set</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td></td>
</tr>
<tr>
<td>E'</td>
<td>( \varepsilon, + )</td>
</tr>
<tr>
<td>T</td>
<td>( \text{id, (} )</td>
</tr>
<tr>
<td>T'</td>
<td>( \varepsilon, * )</td>
</tr>
<tr>
<td>F</td>
<td>( \text{id, (} )</td>
</tr>
</tbody>
</table>

- Applying Rule 3.
An Example Grammar

$$E \rightarrow TE'$$

- Applying Rule 3, nothing more changes and we are done!

<table>
<thead>
<tr>
<th>Non-Terminals</th>
<th>First set</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>id, (</td>
</tr>
<tr>
<td>E'</td>
<td>ε, +</td>
</tr>
<tr>
<td>T</td>
<td>id, (</td>
</tr>
<tr>
<td>T'</td>
<td>ε, *</td>
</tr>
<tr>
<td>F</td>
<td>id, (</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>ε</td>
<td>ε</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>(</td>
<td>(</td>
</tr>
<tr>
<td>)</td>
<td>)</td>
</tr>
<tr>
<td>id</td>
<td>id</td>
</tr>
</tbody>
</table>
Follow Set Rules

1. First put $ (the end of input marker) in Follow(S) (S is the start symbol).
2. If there is a production $A \rightarrow aBb$, (where $a$ can be a whole string) then everything in FIRST(b) except for $\varepsilon$ is placed in FOLLOW(B).
3. If there is a production $A \rightarrow aB$, then everything in FOLLOW(A) is in FOLLOW(B).
4. If there is a production $A \rightarrow aBb$, where FIRST(b) contains $\varepsilon$, then everything in FOLLOW(A) is in FOLLOW(B).
An Example Grammar

E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E)
F → id

- The first thing we do is Add $ to the start Symbol 'E'.

<table>
<thead>
<tr>
<th>Non-Terminals</th>
<th>Follow set</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>$</td>
</tr>
<tr>
<td>E'</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
</tr>
<tr>
<td>T'</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>
An Example Grammar

\[
\begin{align*}
E & \rightarrow TE' \\
E' & \rightarrow +TE' \\
E' & \rightarrow \varepsilon \\
T & \rightarrow FT' \\
T' & \rightarrow *FT' \\
T' & \rightarrow \varepsilon \\
F & \rightarrow (E) \\
F & \rightarrow id
\end{align*}
\]

<table>
<thead>
<tr>
<th>Non-Terminals</th>
<th>Follow set</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>$$</td>
</tr>
<tr>
<td>$E'$</td>
<td></td>
</tr>
<tr>
<td>$T$</td>
<td>+</td>
</tr>
<tr>
<td>$T'$</td>
<td></td>
</tr>
<tr>
<td>$F$</td>
<td></td>
</tr>
</tbody>
</table>

- Apply rule 2 to $E' \rightarrow +TE'$. This says that everything in $\text{First}(E')$ except for $\varepsilon$ should be in $\text{Follow}(T)$. 
An Example Grammar

\[ E \rightarrow TE' \]
\[ E' \rightarrow +TE' \]
\[ E' \rightarrow \varepsilon \]
\[ T \rightarrow FT' \]
\[ T' \rightarrow *FT' \]
\[ T' \rightarrow \varepsilon \]
\[ F \rightarrow (E) \]
\[ F \rightarrow \text{id} \]

<table>
<thead>
<tr>
<th>Non-Terminals</th>
<th>Follow set</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>$</td>
</tr>
<tr>
<td>E'</td>
<td>$</td>
</tr>
<tr>
<td>T</td>
<td>+</td>
</tr>
<tr>
<td>T'</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

• Apply rule 3 to \( E \rightarrow TE' \). This says that we should add everything in \( \text{Follow}(E) \) into \( \text{Follow}(E') \).
An Example Grammar

E → TE’
E’ → +TE’
E’ → ε
T → FT’
T’ → *FT’
T’ → ε
F → (E)
F → id

<table>
<thead>
<tr>
<th>Non-Terminals</th>
<th>Follow set</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>$</td>
</tr>
<tr>
<td>E’</td>
<td>$</td>
</tr>
<tr>
<td>T</td>
<td>+</td>
</tr>
<tr>
<td>T’</td>
<td>+</td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

• Apply rule 3 to T → FT’. This says that we should add everything in Follow(T) into Follow(T’).
An Example Grammar

\[
\begin{align*}
E & \rightarrow TE' \\
E' & \rightarrow +TE' \\
E' & \rightarrow \epsilon \\
T & \rightarrow FT' \\
T' & \rightarrow *FT' \\
T' & \rightarrow \epsilon \\
F & \rightarrow (E) \\
F & \rightarrow \text{id}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Non-Terminals</th>
<th>Follow set</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>$</td>
</tr>
<tr>
<td>E'</td>
<td>$</td>
</tr>
<tr>
<td>T</td>
<td>+</td>
</tr>
<tr>
<td>T'</td>
<td>+</td>
</tr>
<tr>
<td>F</td>
<td>*</td>
</tr>
</tbody>
</table>

- Apply rule 2 to $T' \rightarrow *FT'$. This says that everything in First(T') except for $\epsilon$ should be in Follow(F).
An Example Grammar

E → TE’
E’ → +TE’
E’ → ε
T → FT’
T’ → *FT’
T’ → ε
F → (E)
F → id

<table>
<thead>
<tr>
<th>Non-Terminals</th>
<th>Follow set</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>$, )</td>
</tr>
<tr>
<td>E’</td>
<td>$</td>
</tr>
<tr>
<td>T</td>
<td>+</td>
</tr>
<tr>
<td>T’</td>
<td>+</td>
</tr>
<tr>
<td>F</td>
<td>*</td>
</tr>
</tbody>
</table>

- Apply rule 2 to F → (E). This says that everything in First(')') should be in Follow(E).
An Example Grammar

E → TE’
E’ → +TE’
E’ → ε
T → FT’
T’ → *FT’
T’ → ε
F → (E)
F → id

Apply rule 3 to E → TE’. This says that we should add everything in Follow(E) into Follow(E’).
An Example Grammar

E → TE’
E’ → +TE’
E’ → ε
T → FT’
T’ → *FT’
T’ → ε
F → (E)
F → id

- Apply rule 4 to E’ → +TE’. This says that we should add everything in Follow(E’) into Follow(T) (because First(E’) contains ε.)
An Example Grammar

E → TE'
E’ → +TE'
E’ → ε
T → FT'
T’ → *FT'
T’ → ε
F → (E)
F → id

- Apply rule 3 to T → FT’. This says that we should add everything in Follow(T) into Follow(T’).
An Example Grammar

\[
\begin{align*}
E & \rightarrow TE' \\
E' & \rightarrow +TE' \\
E' & \rightarrow \varepsilon \\
T & \rightarrow FT' \\
T' & \rightarrow *FT' \\
T' & \rightarrow \varepsilon \\
F & \rightarrow (E) \\
F & \rightarrow \text{id}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Non-Terminals</th>
<th>Follow set</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>$, \rangle$</td>
</tr>
<tr>
<td>E'</td>
<td>$, \rangle$</td>
</tr>
<tr>
<td>T</td>
<td>$, +, \rangle$</td>
</tr>
<tr>
<td>T'</td>
<td>$, +, \rangle$</td>
</tr>
<tr>
<td>F</td>
<td>$, +, *, \rangle$</td>
</tr>
</tbody>
</table>

- Apply rule 4 to \( T' \rightarrow *FT' \). This says that we should add everything in \text{Follow}(T')\ into \text{Follow}(F).
Predict Set

- Predict: $(A \rightarrow X_1 \ldots X_m) = (\text{FIRST}(X_1 \ldots X_m) - \{\varepsilon\}) \cup (\text{if } X_1, \ldots, X_m \rightarrow^* \varepsilon \text{ then } \text{FOLLOW}(A) \text{ ELSE } \text{NULL})$
LL Parsing

\[
\begin{align*}
\text{program} & \rightarrow \text{stmt_list}$
\text{stmt_list} & \rightarrow \text{stmt \ stmt_list} \mid \varepsilon \\
\text{stmt} & \rightarrow \text{id:= expr} \mid \text{read id} \mid \text{read expr} \\
\text{expr} & \rightarrow \text{term \ term_tail} \\
\text{term_tail} & \rightarrow \text{add_op \ term \ term_tail} \mid \varepsilon \\
\text{term} & \rightarrow \text{factor \ factor_tail} \\
\text{factor_tail} & \rightarrow \text{mul_op \ factor \ factor_tail} \mid \varepsilon \\
\text{factor} & \rightarrow (\text{expr}) \mid \text{id} \mid \text{literal} \\
\text{add_op} & \rightarrow + \mid - \\
\text{mul_op} & \rightarrow * \mid / \\
\end{align*}
\]

- FIRST(program) = \{id, read, write, $$\}
- FOLLOW(program) = \{\varepsilon\}
- PREDICT(program \rightarrow \text{stmt_list $$}) = \{\text{id, read, write, $$}\}
- FOLLOW(id) = \{+, *, /, ):, =, \text{id, read, write, $$}\}
- FIRST(factor_tail) = \{*, /, \varepsilon\}
- FOLLOW(factor_tail) = \{+, -, \), id, read, write, $$\}
- PREDICT(factor_tail \rightarrow m_op factor \ factor_tail) = \{*, /\}
- PREDICT(factor_tail \rightarrow \varepsilon) = \{+, -, \), id, read, write, $$\}
LL Parsing

- LL(1) parse table for parsing for calculator language

<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>
<th>$$</th>
</tr>
</thead>
<tbody>
<tr>
<td>program</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>-</td>
<td>-</td>
</tr>
<tr>
<td>stmt_list</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>-</td>
<td>-</td>
</tr>
<tr>
<td>stmt</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>
<td>-</td>
</tr>
<tr>
<td>expr</td>
<td>7</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td></td>
<td>-</td>
<td>-</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>term_tail</td>
<td>9</td>
<td>-</td>
<td>9</td>
<td>9</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>term</td>
<td>10</td>
<td>10</td>
<td>-</td>
<td>-</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>factor_tail</td>
<td>12</td>
<td>-</td>
<td>12</td>
<td>12</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>factor</td>
<td>14</td>
<td>15</td>
<td>-</td>
<td>-</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>13</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>add_op</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>16</td>
<td>17</td>
<td>-</td>
</tr>
<tr>
<td>mult_op</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>

2016/9/25
LL Parsing

- It is conventional in general discussions of grammars to use:
  - Lower case letters near the beginning of the alphabet for terminals.
  - Lower case letters near the end of the alphabet for strings of terminals.
  - Upper case letters near the beginning of the alphabet for non-terminals.
  - Upper case letters near the end of the alphabet for arbitrary symbols.
  - Greek letters for arbitrary strings of symbols.
LL Parsing

• Algorithm First/Follow/Predict:

  - \( \text{FIRST}(\alpha) = \{ a : \alpha \rightarrow^* a \beta \} \)
    \( \cup \) (if \( \alpha \Rightarrow^* \varepsilon \) THEN \( \{ \varepsilon \} \) ELSE NULL)

  - \( \text{FOLLOW}(A) = \{ a : S \rightarrow^+ \alpha A a \beta \} \)
    \( \cup \) (if \( S \rightarrow^* \alpha A \) THEN \( \{ \varepsilon \} \) ELSE NULL)

  - \( \text{Predict} (A \rightarrow X_1 \ldots X_m) = (\text{FIRST}(X_1 \ldots X_m) - \{ \varepsilon \}) \cup \) (if \( X_1, \ldots, X_m \rightarrow^* \varepsilon \) then \( \text{FOLLOW}(A) \) ELSE NULL)

• A good demonstration site:

program → stmt_list $$

stmt_list → stmt stmt_list
stmt_list → ε
stmt → id := expr
stmt → read id
stmt → write expr
expr → term term_tail
term_tail → add_op term term_tail
term_tail → ε
term → factor factor_tail
factor_tail → mult_op factor factor_tail
factor_tail → ε
factor → ( expr )
factor → id
factor → number
add_op → +
add_op → -
mult_op → *
mult_op → /

$$ ∈ \text{FOLLOW}(\text{stmt_list}),
ε ∈ \text{FOLLOW}($$), \text{and} \ ε ∈ \text{FOLLOW}(\text{program})

ε ∈ \text{FIRST}(\text{stmt_list})
id ∈ \text{FIRST}(\text{stmt}) \text{ and} \ := ∈ \text{FOLLOW}(\text{id})
read ∈ \text{FIRST}(\text{stmt}) \text{ and} \ id ∈ \text{FOLLOW}(\text{read})
write ∈ \text{FIRST}(\text{stmt})

ε ∈ \text{FIRST}(\text{term_tail})

ε ∈ \text{FIRST}(\text{factor_tail})
( ∈ \text{FIRST}(\text{factor}) \text{ and} \ ) ∈ \text{FOLLOW}(\text{expr})
id ∈ \text{FIRST}(\text{factor})
number ∈ \text{FIRST}(\text{factor})

+ ∈ \text{FIRST}(\text{add_op})
- ∈ \text{FIRST}(\text{add_op})
* ∈ \text{FIRST}(\text{mult_op})
/ ∈ \text{FIRST}(\text{mult_op})

Figure 2.21: “Obvious” facts about the LL(1) calculator grammar.
FIRST

\[
\begin{align*}
\text{program} & \rightarrow \{id, \text{read}, \text{write}, \$$\} \\
\text{stmt} & \rightarrow \text{stmt_list} \{id, \text{read}, \text{write}, \$$\} \\
\text{expr} & \rightarrow \{\text{\_}, \text{id, number}\} \\
\text{term} & \rightarrow \text{term_tail} \{\text{\_}, \text{id, number}\} \\
\text{factor} & \rightarrow \text{factor_tail} \{\text{\_}, \text{id, number}\} \\
\text{add_op} & \rightarrow \{\text{\_}, \text{\_}, \text{id, number}\} \\
\text{mult_op} & \rightarrow \{\text{\_}, \text{\_}, \text{id, number}\}
\end{align*}
\]

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

FOLLOW

\[
\begin{align*}
\text{id} & \rightarrow \{\text{\_}, \text{id, number}\} \\
\text{number} & \rightarrow \{\text{\_}, \text{id, number}\} \\
\text{read} & \rightarrow \{\text{\_}, \text{id, number}\} \\
\text{write} & \rightarrow \{\text{\_}, \text{id, number}\} \\
\text{expr} & \rightarrow \{\text{\_}, \text{id, number}\} \\
\text{term} & \rightarrow \{\text{\_}, \text{id, number}\} \\
\text{factor} & \rightarrow \{\text{\_}, \text{id, number}\} \\
\text{add_op} & \rightarrow \{\text{\_}, \text{id, number}\} \\
\text{mult_op} & \rightarrow \{\text{\_}, \text{id, number}\}
\end{align*}
\]

PREDICT

\[
\begin{align*}
1 & \quad \text{program} \rightarrow \text{stmt_list} \{\text{id, read, write, \$$}\} \\
2 & \quad \text{stmt_list} \rightarrow \text{stmt} \text{stmt_list} \{\text{id, read, write}\} \\
3 & \quad \text{stmt_list} \rightarrow \epsilon \{\$$\} \\
4 & \quad \text{stmt} \rightarrow \text{id} \:= \text{expr} \{\text{id}\} \\
5 & \quad \text{stmt} \rightarrow \text{read id} \{\text{read}\} \\
6 & \quad \text{stmt} \rightarrow \text{write expr} \{\text{write}\} \\
7 & \quad \text{expr} \rightarrow \text{term term_tail} \{\text{\_}, \text{id, number}\} \\
8 & \quad \text{term_tail} \rightarrow \text{add_op term term_tail} \{\text{\_}, \text{id, number}\} \\
9 & \quad \text{term_tail} \rightarrow \epsilon \{\text{id, number}\} \\
10 & \quad \text{term} \rightarrow \text{factor factor_tail} \{\text{\_}, \text{id, number}\} \\
11 & \quad \text{factor_tail} \rightarrow \text{mult_op factor factor_tail} \{\text{\_}, \text{id, number}\} \\
12 & \quad \text{factor_tail} \rightarrow \epsilon \{\} \\
13 & \quad \text{factor} \rightarrow \{\text{expr} \} \{\} \\
14 & \quad \text{factor} \rightarrow \text{id} \{\text{id}\} \\
15 & \quad \text{factor} \rightarrow \text{number} \{\text{number}\} \\
16 & \quad \text{add_op} \rightarrow \{\text{\_}, \text{id, number}\} \\
17 & \quad \text{add_op} \rightarrow \{\text{\_}, \text{id, number}\} \\
18 & \quad \text{mult_op} \rightarrow \{\text{\_}, \text{id, number}\} \\
19 & \quad \text{mult_op} \rightarrow \{\text{\_}, \text{id, number}\}
\end{align*}
\]

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

- If any token belongs to the predict set of more than one production with the same LHS, then the grammar is not LL(1).
- A conflict can arise because:
  - The same token can begin more than one RHS.
  - It can begin one RHS and can also appear after the LHS in some valid program, and one possible RHS is $\varepsilon$. 
LR Parsing

- LR parsers are almost always table-driven:
  - Like a table-driven LL parser, an LR parser uses a big loop in which it repeatedly inspects a two-dimensional table to find out what action to take.
  - Unlike the LL parser, however, the LR driver has non-trivial state (like a DFA), and the table is indexed by current input token and current state.
  - The stack contains a record of what has been seen SO FAR (NOT what is expected).
LR Parsing

- A scanner is a DFA.
  - It can be specified with a state diagram.
- An LL or LR parser is a PDA,
  - Early's & CYK algorithms do NOT use PDAs.
  - A PDA can be specified with a state diagram and a stack
    - The state diagram looks just like a DFA state diagram, except the arcs are labeled with `<input symbol, top-of-stack symbol>` pairs, and in addition to moving to a new state the PDA has the option of pushing or popping a finite number of symbols onto/off the stack.
LR Parsing

- An LL(1) PDA has only one state!
  - Actually two; it needs a second one to accept with, but that's all (it's pretty simple).
  - All the arcs are self loops; the only difference between them is the choice of whether to push or pop.
  - The final state is reached by a transition that sees EOF on the input and the stack.
LR Parsing

- An SLR/LALR/LR PDA has multiple states
  - it is a "recognizer," not a "predictor"
  - it builds a parse tree from the bottom up
  - the states keep track of which productions we might be in the middle
- The parsing of the Characteristic Finite State Machine (CFSM) is based on
  - Shift
  - Reduce
LR Parsing

- To illustrate LR parsing, consider the grammar (Figure 2.24, Page 73):

```
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 → * | /
term → factor | term mult_op factor
factor → ( expr ) | id | number
add op → + | -
mult op → * | /
```
LR Parsing

• This grammar is SLR(1), a particularly nice class of bottom-up grammar:
  ▫ It isn't exactly what we saw originally.
  ▫ We've eliminated the epsilon production to simplify the presentation.
• For details on the table driven SLR(1) parsing please note the following slides.
### State Transitions

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

---

Figure 2.25: CFSM 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.
| Top-of-stack state | s | e | t | f | ao | mo | Current input symbol | id | lit | r | w | := | ( | ) | + | - | * | / | $$ |
|-------------------|---|---|---|---|----|----|---------------------|----|----|---|---|----|---|----|---|---|---|---|---|---|
| 0                 | s2| b3| --| --| --| --| s3                  | s1| s4| --| --| --| --| --| --| --| --| --| --| --| --| --| --|
| 1                 | --| --| --| --| --| --| b5                  | --| --| --| --| --| --| --| --| --| --| --| --| --| --| --| --| --|
| 2                 | --| b2| --| --| --| --| s3                  | s1| s4| --| --| --| --| --| b1| --| --| --| --| --| --| --| --|
| 3                 | --| --| --| --| --| --| s5                  | --| --| --| --| --| --| --| --| --| --| --| --| --| --| --| --| --|
| 4                 | --| s6| s7| b9| --| --| b12                 | b13| s8| --| --| --| --| --| r6| r6| r6| b14| b15| --| --| r6|
| 5                 | --| s9| s7| b9| --| --| b12                 | b13| s8| --| --| --| --| --| r6| r7| r7| r7| r7| r7| b16| b17| r7|
| 6                 | --| --| --| --| --| --| s10                 | r6| r6| r6| b14| b15| --| --| r6| r6| b14| b15| --| --| r6|
| 7                 | --| s12| s7| b9| --| --| b12                | b13| s8| --| --| --| --| --| r7| r7| r7| r7| r7| b16| b17| r7|
| 8                 | --| s12| s7| b9| --| --| b12                | b13| s8| --| --| --| --| --| r7| r7| r7| r7| b16| b17| r7|
| 9                 | --| --| --| --| --| --| s10                 | r4| r4| r4| b14| b15| --| --| r4| r4| b14| b15| --| --| r4|
| 10                | --| --| s13| b9| --| --| b12                | b13| s8| --| --| --| --| --| r7| r7| r7| r7| b16| b17| r7|
| 11                | --| --| --| b10| --| --| b12                | b13| s8| --| --| --| --| --| r8| r8| r8| r8| r8| r8| b16| b17| r8|
| 12                | --| --| --| --| s10| --| --| s11               | r8| r8| r8| b11| b14| b15| --| --| --| --| --| --| --| --|
| 13                | --| --| --| --| --| s11| --| r8| r8| r8| --| b11| b14| b15| --| --| --| --| --| --| --| --| --|

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

- SLR parsing is based on:
  - Shift
  - Reduce
  - Shift & Reduce (for optimization).

---

**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 → stmt_list $$
\text{stmt_list}$$.