Lecture 4: Lexical and Syntax Analysis
Lecture 4A: Lexical and Syntax Analysis II

(The lectures are based on the slides copyrighted by Keith Cooper and Linda Torczon from Rice University.)
Review: Three Pass Compiler

Scanner and Parser collaborate to check the syntax of the input program.

1. Scanner maps stream of characters into words (words are the fundamental unit of syntax)

   Scanner produces a stream of tokens for the parser
   
   Token is <part of speech, word>
   
   “Part of speech” is a unit in the grammar

2. Parser maps stream of words into a sentence in a grammatical model of the input language
Review: Compiler Front End

Scanner and Parser collaborate to check the syntax of the input program.

1. Scanner maps stream of characters into words (words are the fundamental unit of syntax)

   Scanner produces a stream of tokens for the parser
   - Token is <part of speech, word>
     "Part of speech" is a unit in the grammar

2. Parser maps stream of words into a sentence in a grammatical model of the input language
Review: The Compiler’s Front End

- **Scanner** looks at every character
  - Converts stream of characters to stream of tokens, or classified words:
    - `<part of speech, word>`
  - Efficiency & scalability matter
    - Scanner is only part of compiler that looks at every character

- **Parser** looks at every token
  - Determines if the stream of tokens forms a sentence in the source language
  - Fits tokens to some syntactic model, or grammar, for the source language
How can we automate the construction of scanner & parsers?

- **Scanner**
  - Specify syntax with regular expressions (REs)
  - Construct finite automaton & scanner from the regular expression

- **Parser**
  - Specify syntax with context-free grammars (CFGs)
  - Construct push-down automaton & parser from the CFG
We can represent this code for recognizing “not” as an automaton:

```
c ← next character
If c = 'n' then {
    c ← next character
    if c = 'o' then {
        c ← next character
        if c = 't'
            then return <NOT,"not">
        else report error
    }
    else report error
} else report error
```

Transition Diagram for “not”:

States drawn with double lines indicate success.
Review: Automata

To recognize a larger set of keywords is (relatively) easy

For example, the set of words \{ if, int, not, then, to, write \}

one-to-one map from final states to words (maybe parts of speech)

Cost is $O(1)$ per character
Review: Specifying An Automaton

We list the words
Each word’s spelling is unique and concise
We can separate them with the | symbol, read as “or”

The specification:

if int not then to write

is a simple regular expression for the language accepted by our automaton

There is some subtlety here. We introduced notation for concatenation and choice (or alternation).

**Concatenation:**
- ab is a followed by b

**Alternation:**
- a | b is either a or b
Review: Specifying An Automaton

We list the words
Each word’s spelling is unique and can be read as
We can separate them with the $|$ symbol, read as "or"

The specification:

if $\text{int} \ | \ \text{not} \ | \ \text{then} \ | \ \text{to} \ | \ \text{write}$

is a simple regular expression for the language accepted by our automaton

We introduced notation for concatenation and choice (or alternation).

**Concatenation:**
$\text{ab}$ is $a$ followed by $b$

**Alternation:**
$a \ | \ b$ is either $a$ or $b$
Review: Specifying An Automaton

• Every Regular Expression corresponds to an automaton

- We can construct, automatically, an automaton from the RE
- We can construct, automatically, an RE from any automaton
- We can convert an automaton easily and directly into code
- The automaton and the code both have $O(1)$ cost per input character

An RE specification leads directly to an efficient scanner
Review: Unsigned Integers

• The RE corresponds to an automaton and an implementation

The automaton and the code can be generated automatically

```
c ← next character
n ← 0
if c = ‘0’
    then return <CONSTANT,n>
else if (‘1’ ≤ c ≤ ‘9’) then {
    n ← atoi(c)
    c ← next character
    while (‘0’ ≤ c ≤ ‘9’) {
        t ← atoi(c)
        n ← n * 10 + t
    }
    return <CONSTANT,n>
}
else report error
```
Review: Regular Expression

• A formal definition

Regular Expressions over an Alphabet Σ

If \( x \in \Sigma \), then \( x \) is an RE denoting the set \( \{ x \} \) or the language \( L = \{ x \} \)
If \( x \) and \( y \) are REs then

1. \( xy \) is an RE denoting \( L(x)L(y) = \{ pq \mid p \in L(x) \text{ and } q \in L(y) \} \)
2. \( x \mid y \) is an RE denoting \( L(x) \cup L(y) \)
3. \( x^* \) is an RE denoting \( L(x)^* = \cup_{0 \leq k < \infty} L(x)^k \) (Kleene Closure)
   \( \text{Set of all strings that are zero or more concatenations of } x \)
4. \( x^+ \) is an RE denoting \( L(x)^+ = \cup_{1 \leq k < \infty} L(x)^k \) (Positive Closure)
   \( \text{Set of all strings that are one or more concatenations of } x \)

\( \varepsilon \) is an RE denoting the empty set
Review: Regular Expression

- How do these operators help?

Regular Expressions over an Alphabet $\Sigma$

If $x$ is in $\Sigma$, then $x$ is an RE denoting the set $\{x\}$ or the language $L = \{x\}$

*The spelling of any letter in the alphabet is an RE*

If $x$ and $y$ are REs then
1. $xy$ is an RE denoting $L(x)L(y) = \{pq \mid p \in L(x) \text{ and } q \in L(y)\}$
   *If we concatenate letters, the result is an RE (spelling of words)*
2. $x \mid y$ is an RE denoting $L(x) \cup L(y)$
   *Any finite list of words can be written as an RE, ( $w_0 \mid w_1 \mid w_2 \mid \ldots \mid w_n$ )*  
3. $x^*$ is an RE denoting $L(x)^* = \bigcup_{0 \leq k < \infty} L(x)^k$
4. $x^+$ is an RE denoting $L(x)^+ = \bigcup_{1 \leq k < \infty} L(x)^k$

*We can use closure to write finite descriptions of infinite, but countable, sets*

$\varepsilon$ is an RE denoting the empty set

*In practice, the empty string is often useful*
Review: Regular Expression

- Let the notation \([0-9]\) be shorthand for \((0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)\)

**RE Examples:**

\[
\begin{align*}
\text{Non-negative integer} & \quad [0-9] [0-9]^* \\
\text{or} & \quad [0-9]^* \\
\text{No leading zeros} & \quad 0 \mid [1-9] [0-9]^* \\
\text{Algol-style Identifier} & \quad ([a-z] | [A-Z]) (([a-z] | [A-Z]) [0-9])^* \\
\text{Decimal number} & \quad 0 \mid [1-9] [0-9]^* \cdot [0-9]^* \\
\text{Real number} & \quad ((0 \mid [1-9] [0-9]^*) \mid (0 \mid [1-9] [0-9]^* \cdot [0-9]^*) \cdot [0-9]) E [0-9] [0-9]^*
\end{align*}
\]

Each of these **REs** corresponds to an automaton and an implementation.
Review: From RE to Scanner

• Construct scanners directly from REs using automata theory

There are several ways to perform this construction

Classic approach is a two-step method.
- 1. Build automata for each piece of the RE using a simple template-driven method
   - Build a specific variation on an automaton that has transitions on $\varepsilon$ and non-deterministic choice (multiple transitions from a state on the same symbol)
   - This construction is called “Thompson’s construction”

- 2. Convert the newly built automaton into a deterministic automaton
   - Deterministic automaton has no $\varepsilon$-transitions and all choices are single-valued
   - This construction is called the “subset construction”

- Given the deterministic automaton, minimize it to reduce the number of states
  - Minimization is a space optimization. Both the original automaton and the minimal one take $O(1)$ time per character
Review: Thompson’s Construction

• The Key Idea

- For each RE symbols and operator, we have a small template
- Build them, in precedence order, and join them with ε-transitions

Precedence in REs: Parentheses, then closure, then concatenation, then alternation
Review: Thompson’s Construction

• The Key Idea
- For each RE symbols and operator, we have a small template
- Build them, in precedence order, and join them with $\varepsilon$-transitions

Precedence in REs: Parentheses, then closure, then concatenation, then alternation
Review: Thompson’s Construction

• The Key Idea
  - For each RE symbols and operator, we have a small template
  - Build them, in precedence order, and join them with ε-transitions

Precedence in REs: Parentheses, then closure, then concatenation, then alternation
Review: Thompson’s Construction

• The Key Idea

- For each RE symbols and operator, we have a small template
- Build them, in precedence order, and join them with $\varepsilon$-transitions

**Precedence in REs:** Parentheses, then closure, then concatenation, then alternation
Review: Thompson’s Construction

• The Key Idea
  - For each RE symbols and operator, we have a small template
  - Build them, in precedence order, and join them with $\varepsilon$-transitions

![NFA diagrams for a, ab, a | b, a^*]

Precedence in REs: Parentheses, then closure, then concatenation, then alternation
Review: Thompson’s Construction

- The Key Idea
- For each RE symbols and operator, we have a small template
- Build them, in precedence order, and join them with $\varepsilon$-transitions

**Precedence in REs:** Parentheses, then closure, then concatenation, then alternation
Review: Thompson’s Construction

- Let’s build an NFA for \( a ( b \mid c )^* \)

1. \( a, b, \) & \( c \)

2. \( b \mid c \)

3. \( ( b \mid c )' \)

4. \( a ( b \mid c )' \)

Precedence in REs: Parentheses, then closure, then concatenation, then alternation
Review: Thompson’s Construction

- Let’s build an NFA for $a (b | c)^*$

1. $a, b, \& c$
   
   $S_0 \xrightarrow{a} S_1$
   $S_0 \xrightarrow{\varepsilon} S_2$
   $S_1 \xrightarrow{b} S_1$

2. $b | c$

   $S_0 \xrightarrow{b} S_2$
   $S_0 \xrightarrow{\varepsilon} S_4$
   $S_1 \xrightarrow{\varepsilon} S_3$
   $S_4 \xrightarrow{c} S_5$

3. $(b | c)'$

4. $a (b | c)'$

**Precedence in REs:** Parentheses, then closure, then concatenation, then alternation
Review: Thompson’s Construction

• Let’s build an NFA for $a ( b \mid c )^*$

1. $a, b, \& c$

2. $b \mid c$

3. $( b \mid c )'$

4. $a ( b \mid c )'$

**Precedence in REs:** Parentheses, then closure, then concatenation, then alternation
Review: Thompson’s Construction

- Let’s build an NFA for $a (b | c)^*$

1. $a$, $b$, & $c$

2. $b | c$

3. $(b | c)'$

4. $a (b | c)'$

**Precedence in REs:** Parentheses, then closure, then concatenation, then alternation
Subset Construction

• The Concept

- Build a simpler automaton (no ε-transitions, no multi-valued transitions) that simulates the behavior of the more complex automaton.
- Each state in the new automaton represents a set of states in the original.

NFA

DFA

<table>
<thead>
<tr>
<th>DFA</th>
<th>NFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_0$</td>
<td>$n_0$</td>
</tr>
</tbody>
</table>
Subset Construction

• The Concept

- Build a simpler automaton (no ε-transitions, no multi-valued transitions) that simulates the behavior of the more complex automaton
- Each state in the new automaton represents a set of states in the original

NFA

DFA
Subset Construction

• The Concept

- Build a simpler automaton (no ε-transitions, no multi-valued transitions) that simulates the behavior of the more complex automaton
- Each state in the new automaton represents a set of states in the original

\[ a( b \mid c)^* \]

\[
\begin{array}{c}
\text{DFA} \\
\begin{array}{c}
\text{NFA} \\
\hline
\text{DFA} \mid \text{NFA} \\
d_0 \mid n_0 \\
d_1 \mid n_1 \ n_2
\end{array}
\end{array}
\]
Subset Construction

• The Concept

- Build a simpler automaton (no $\varepsilon$-transitions, no multi-valued transitions) that simulates the behavior of the more complex automaton
- Each state in the new automaton represents a set of states in the original

NFA

DFA

<table>
<thead>
<tr>
<th>DFA</th>
<th>NFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_0$</td>
<td>$n_0$</td>
</tr>
<tr>
<td>$d_1$</td>
<td>$n_1 \ n_2 \ n_3$</td>
</tr>
</tbody>
</table>
Subset Construction

• The Concept

- Build a simpler automaton (no ε-transitions, no multi-valued transitions) that simulates the behavior of the more complex automaton
- Each state in the new automaton represents a set of states in the original

NFA

\[ a \ (b \ | \ c)^* \]

DFA

<table>
<thead>
<tr>
<th>DFA</th>
<th>NFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d_0 )</td>
<td>( n_0 )</td>
</tr>
<tr>
<td>( d_1 )</td>
<td>( n_1 \ n_2 \ n_3 \ n_4 )</td>
</tr>
</tbody>
</table>
Subset Construction

• The Concept

- Build a simpler automaton (no ε-transitions, no multi-valued transitions) that simulates the behavior of the more complex automaton
- Each state in the new automaton represents a set of states in the original

NFA

DFA

<table>
<thead>
<tr>
<th>DFA</th>
<th>NFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>(d_0)</td>
<td>(n_0)</td>
</tr>
<tr>
<td>(d_1)</td>
<td>(n_1\ n_2\ n_3\ n_4\ n_6)</td>
</tr>
</tbody>
</table>
Subset Construction

• The Concept
  - Build a simpler automaton (no \(\varepsilon\)-transitions, no multi-valued transitions) that simulates the behavior of the more complex automaton
  - Each state in the new automaton represents a set of states in the original

- NFA

- DFA

\[
\begin{array}{c|c|c}
\text{DFA} & \text{NFA} \\
\hline
\text{d}_0 & \text{n}_0 \\
\text{d}_1 & \text{n}_1 \text{ n}_2 \text{ n}_3 \text{ n}_4 \text{ n}_6 \text{ n}_9 \\
\end{array}
\]
Subset Construction

• The Concept

- Build a simpler automaton (no $\varepsilon$-transitions, no multi-valued transitions) that simulates the behavior of the more complex automaton
- Each state in the new automaton represents a set of states in the original

<table>
<thead>
<tr>
<th>DFA</th>
<th>NFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_0$</td>
<td>$n_0$</td>
</tr>
<tr>
<td>$d_1$</td>
<td>$n_1$ $n_2$ $n_3$ $n_4$ $n_6$ $n_9$</td>
</tr>
<tr>
<td>$d_2$</td>
<td>$n_5$ $n_8$ $n_9$ $n_3$ $n_4$ $n_6$</td>
</tr>
</tbody>
</table>
Subset Construction

• The Concept

- Build a simpler automaton (no \(\varepsilon\)-transitions, no multi-valued transitions) that simulates the behavior of the more complex automaton
- Each state in the new automaton represents a set of states in the original

NFA

\[ a \cdot (b \mid c)^* \]

DFA

<table>
<thead>
<tr>
<th>DFA</th>
<th>NFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>(d_0)</td>
<td>(n_0)</td>
</tr>
<tr>
<td>(d_1)</td>
<td>(n_1 \ n_2 \ n_3 \ n_4 \ n_6 \ n_9)</td>
</tr>
<tr>
<td>(d_2)</td>
<td>(n_5 \ n_8 \ n_9 \ n_3 \ n_4 \ n_6)</td>
</tr>
<tr>
<td>(d_3)</td>
<td>(n_7 \ n_8 \ n_9 \ n_3 \ n_4 \ n_6)</td>
</tr>
</tbody>
</table>
Minimization

• DFA minimization algorithms work by discovering states that are equivalent in their contexts and replacing multiple states with a single one

- Minimization reduces the number of states, but does not change the costs
Minimization

- DFA minimization algorithms work by discovering states that are equivalent in their contexts and replacing multiple states with a single one.

- Minimization reduces the number of states, but does not change the costs.
Minimization

- DFA minimization algorithms work by discovering states that are equivalent in their contexts and replacing multiple states with a single one.

- Minimization reduces the number of states, but does not change the costs.

<table>
<thead>
<tr>
<th>Minimal DFA State</th>
<th>$s_0$</th>
<th>$s_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original DFA State</td>
<td>$d_0$</td>
<td>$d_1, d_2, d_3$</td>
</tr>
</tbody>
</table>
Implementing an Automaton

• A common strategy is to simulate the DFA’s execution
  - Skeleton parser + a table that encodes the automaton
  - The scanner generator constructs the table
  - The skeleton parser does not change

```
state ← s₀
char ← NextChar()
while (char ≠ EOF) {
    state ← δ[state,char]
    char ← NextChar()
}
if (state is a final state)
    then report success
else report an error
```

Simple Skeleton Scanner

### Transition table for our minimal DFA

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₀</td>
<td>S₁</td>
<td>Sₑ</td>
<td>Sₑ</td>
</tr>
<tr>
<td>S₁</td>
<td>Sₑ</td>
<td>S₁</td>
<td>S₁</td>
</tr>
</tbody>
</table>
Automatic Scanner Construction

• **Scanner Generator**
  - Tasks in a specification written as a collection of regular expressions
  - Combines them into one RE using alternation ("|"")
  - Builds the minimal automaton (§ 2.4, 2.6.2 EAC)
  - Emits the tables to drive a skeleton scanner (§ 2.5 EAC)
Automatic Scanner Construction

- **Scanner Generator**
  - As alternative, the generator can produce code rather than tables
  - Direct-coded scanners are ugly, but often faster than table-driven scanners
  - Other than speed, the two are equivalent
Automaton versus Scanner

• Automaton accepts or rejects a word
  - Runs until it exhausts the input and accepts or rejects the stream
• Scanner looks at the whole program and returns all of the tokens
  - Must break the input stream into separate words
  - Must capture and classify the lexeme
  - Must decide when it has looked beyond the end of a word

Scanner generators build the automaton for a set of REs and then convert the automaton into an efficient scanner

Consider the RE:

\[
r \ [0-9]^+
\]

And its minimal DFA:
Table-Driven Scanner

- The scanner does more than recognize a word
  - Categorize the token and capture its spelling
  - Build DFA carefully such that final state maps to a category
  - “Remember” the characters at each transition to capture the spelling
Table-Driven Scanner

• The scanner does more than recognize a word
  - Categorize the token and capture its spelling
  - Build DFA carefully such that final state maps to a category
  - “Remember” the characters at each transition to we capture the spelling

```
char ← next character
state ← s₀
lexeme ← null string
while (char ≠ EOF) {
    lexeme ← lexeme || char
    state ← δ[state, char]
    char ← next character
}
If (state ∈ Sₐ) then result ← <PoS(state), lexeme>
else result ← <invalid,“”>
return result
```

Transition Table (δ)

<table>
<thead>
<tr>
<th>δ</th>
<th>r</th>
<th>0,1,2,3,4,5,6,7,8,9</th>
<th>Any Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₀</td>
<td>S₁</td>
<td>Sₑ</td>
<td>Sₑ</td>
</tr>
<tr>
<td>S₁</td>
<td>Sₑ</td>
<td>Sₑ</td>
<td>Sₑ</td>
</tr>
<tr>
<td>S₂</td>
<td>Sₑ</td>
<td>Sₑ</td>
<td>Sₑ</td>
</tr>
<tr>
<td>Sₑ</td>
<td>Sₑ</td>
<td>Sₑ</td>
<td>Sₑ</td>
</tr>
</tbody>
</table>
Table-Driven Scanner

• The scanner does more than recognize a word
  - Categorize the token and capture its spelling
  - Build DFA carefully such that final state maps to a category
  - “Remember” the characters at each transition to we capture the spelling

• The scanner needs to recognize when one token ends and a new one begins
  - Need to return all the words, in order, but one at a time
  - Do not want force blanks or delimiters everywhere
  - Should “x + y * z” and “x+y*z” scan differently?
Recognizing Token Boundaries

- Two potential solutions

- Require delimiters between every token — might be ugly and painful
- Run the automaton to an error or EOF, and then back up to a final state

```c
// recognize words
char ← next character
state ← s₀
lexeme ← empty string
clear stack
push (bad)
while (state ≠ sₐ) do
  char ← next character
  lexeme ← lexeme || char
  if state ∈ Sₐ
    then clear stack
    push (state)
  state ← δ(state,char)
end

// clean up final state
while (state ∉ Sₐ and state ≠ bad) do
  state ← pop()
  truncate lexeme
  roll back the input one char
end

// report the results
if (state ∈ Sₐ)
  then result = <PoS(state), lexeme>
else result = <invalid, ””>
return result
```
Table-Driven Scanner

• The scanner does more than recognize a word
  - Categorize the token and capture its spelling
  - Build DFA carefully such that final state maps to a category
  - “Remember” the characters at each transition to we capture the spelling

• The scanner needs to recognize when one token ends and a new one begins
  - Need to return all the words, in order, but one at a time
  - Do not want force blanks or delimiters everywhere
    - Should “x + y * z” and “x+y*z” scan differently?

• The scanner needs to deal with the case when multiple matches are possible
  - Two or more categories for the same exact string
  - Two or more matches for a given string
Ambiguous Regular Expressions

• **Ambiguity is not always a bad thing**
  - Sometimes, it is easier to specify an ambiguous RE
    + | ++ | += | * | ** | *|= | < | <= | = | > | >= | …
  - The scanner needs to handle ambiguous REs in an appropriate way

• **Need an approach to specify the expected behavior**
  - Scanner generators assign priority or precedence by the order in which the patterns appear in the specification
  - First matching pattern takes precedence
Example: Recognizing Keywords “for” and “while”

• For an RE such as

  \[ \text{for} \mid \text{while} \mid (\ [a-z] \mid [a-z] \mid [0-9])^* \]

  we want an automata such as

  \[ \text{s4} \Rightarrow \text{“for”} \]
  \[ \text{s9} \Rightarrow \text{“while”} \]
  \[ \text{s10} \Rightarrow \text{general identifier} \]
Ambiguous Regular Expressions

- **Most languages have both keywords and identifiers**
  - “for” might be the RE for the keyword for
  - ([a-z]) ([a-z] | [0-9]) might be the RE for an identifier
  - The input string “for” matches both REs

- **Keywords are sometimes handled as a special case**
  - Use the simpler RE and automaton shown here
  - Build hash table of identifiers & preload keywords
  - Recognize identifiers during the hash lookup

- **The Tradeoff**
  - More states in the scanner versus lookup cost on keywords
  - Encoding keywords in RE has no significant runtime cost
  - Either strategy works & difference in runtime cost is tiny
Automatic Generation of Scanners and Parsers

• Scanner Generation Process
  1. Write down the REs for the input language and connect them with “|”
  2. Build a big, simple NFA
  3. Build the DFA that simulates the NFA
  4. Minimize the number of states in the DFA
  5. Generate an implementation from the minimal DFA

• Scanner Generators
  1. lex, flex, jflex, and such all work along these lines
  2. Algorithms are well-known and well-understood
  3. Key issues:
     finding the longest match rather than first match, and engineering the interface to the parser
Automatic Generation of Scanners and Parsers

- **Design Time**
  Specification as micro syntax and syntax

- **Compiler-build Time**
  Convert specification to actual compiler code

- **Compile Time**
  Translate an application in specified language into an executable form
Automatic Scanner Generation

- **flex is a scanner generator**
  - *flex* follows the input conventions of *lex*
  - Takes a file that includes *REs* for the words
  - Produces a table-driven scanner to recognize words and return tokens
- *flex* works with *bison*: the GNU LR(1) parser generator
  - *bison* produces a file of definitions to
  - connect scanner parts of speech to terminal symbols in the grammar (*.tab.h)
Use flex to Generate Scanners

• A flex input file has three parts
  • Definitions section
    - Holds #include’s, declarations, and definitions (e.g., DIGIT [0-9])
    - Mechanism to insert verbatim code at the top of generated scanner code
  • Rules section
    - Holds a series of rules, which consist of a pattern and an action
      - pattern is just an RE
      - action is a fragment of C code; either a single statement or a block in brackets
    - Rules are prioritized in order of appearance
    - Mechanism to insert arbitrary code into scanning routine
  • User code section
    - Holds code that is copied, verbatim, to the scanner output file
    - This section is optional — a good place to put the main function

Sections are separated by the delimiter %%

Sample Code:
/* Companion source code for "flex & bison", published by O'Reilly */
/* fb1-1 just like unix wc */
%
int chars = 0;
int words = 0;
int lines = 0;
%
%
[a-zA-Z]+ { words++; chars += strlen(yytext); }
\n { chars++; lines++; }
. { chars++; }
%
main()
{
  yylex();
  printf("%8d%8d%8d\n", lines, words, chars);
}
Specifying RE Patterns in *flex*

*flex has a rich specification language*

- `a` matches the character `a`
- `.` matches any character except `\n`
- `[abc]` is a class; it matches `a | b | c`
- `[af-jZ]` matches `a | f | g | h | i | j | Z`
  
  *f-j is a numerical range in collating sequence*
- `[^abc]` is a negated class; it matches any character except `a`, `b`, or `c`
- `[a-z]{[aeiou]}` matches lowercase consonant letters (set subtraction)
- `x*` matches zero or more instances\(^1\) of `x`
- `x+` matches one or more instances of `x`
- `x?` matches zero or one instance of `x`
- `x{3}` matches exactly 3 `x`’s, as would `xxx`
- `x{3,}` matches 3 or more instances of `x`
- `{3,5}` matches 3, 4, or 5 instances of `x`
- `{name}` matches a name defined in the definition section of the file
- `\n` matches the newline character
- `\a, \b, \f, \r, \t, \v` match their C escapes
- `\x` matches `x`; useful for matching operators in the RE notation such as `^`
- `xy` matches `RE x` followed by `RE y`
- `x | y` matches either `RE x` or `RE y`
- `^x` matches `x` at the start of a line
- `x$` matches `RE x` at the end of a line
- `x/y` matches `x` only if it is followed by `RE y`, but does not match `y`. (pushes `y` back into the input stream)

And, there are more options ...
Lecture 4B: Lexical and Syntax Analysis III
Review: Compiler Front End

Scanner and Parser collaborate to check the syntax of the input program.

1. Scanner maps stream of characters into words (words are the fundamental unit of syntax)

Scanner produces a stream of tokens for the parser

Token is <part of speech, word>
“Part of speech” is a unit in the grammar

2. Parser maps stream of words into a sentence in a grammatical model of the input language
Review: The Compiler’s Front End

- **Scanner** looks at every character
  - Converts stream of characters to stream of tokens, or classified words:
    - `<part of speech, word>`
  - Efficiency & scalability matter
  - Scanner is only part of compiler that looks at every character

- **Parser** looks at every token
  - Determines if the stream of tokens forms a *sentence* in the source language
  - Fits tokens to some syntactic model, or grammar, for the source language
How can we automate the construction of scanner & parsers?

- **Scanner**
  - Specify syntax with regular expressions (REs)
  - Construct finite automaton & scanner from the regular expression

- **Parser**
  - Specify syntax with context-free grammars (CFGs)
  - Construct push-down automaton & parser from the CFG

Review: The Compiler’s Front End
The Study of Parsing

Parsing is the process of discovering a *derivation* for some sentence

- Need a mathematical model of syntax — a context-free grammar $G$
- Need an algorithm to test membership in $L(G)$
A **CFG** is a four tuple, \( G = (S, N, T, P) \)

- **S** is the *start symbol* of the grammar
  - \( L(G) \) is the set of sentences that can be derived from **S**
- **N** is a set of *nonterminal symbols* or syntactic variables
  - \{ Goal, List, Pair \}
- **T** is a set of *terminal symbols* or words
  - \{ (, ) \}
- **P** is a set of *productions* or *rewrite rules*

This grammar is written in a variant of Backus–Naur Form (BNF)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal ( \rightarrow ) List</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>List ( \rightarrow ) List Pair</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Pair</td>
</tr>
<tr>
<td>4</td>
<td>Pair ( \rightarrow ) ( List )</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>()</td>
</tr>
</tbody>
</table>

Parentheses Grammar
A **CFG** is a four tuple, \( G = (S, N, T, P) \)

- \( S \) is the **start symbol** of the grammar
- \( L(G) \) is the set of sentences that can be derived from \( S \)
- \( N \) is a set of **nonterminal symbols** or syntactic variables
  - \{ Goal, List, Pair \}
- \( T \) is a set of **terminal symbols** or words
  - \{ ( , ) \}
- \( P \) is a set of **productions** or **rewrite rules**

\[ L(G) \text{ includes sentences such as } “( )”, “(( ))”, \\ & “(( ))” \]

This grammar is written in a variant of Backus–Naur Form (BNF)

<table>
<thead>
<tr>
<th></th>
<th>Goal</th>
<th>( \rightarrow )</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal</td>
<td>( \rightarrow )</td>
<td>List</td>
</tr>
<tr>
<td>2</td>
<td>List</td>
<td>( \rightarrow )</td>
<td>List Pair</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>Pair</td>
</tr>
<tr>
<td>4</td>
<td>Pair</td>
<td>( \rightarrow )</td>
<td>( List )</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>( )</td>
</tr>
</tbody>
</table>

**Parentheses Grammar**
Specifying Syntax: Context-Free Grammars

Why not use regular expressions to specify PL syntax?

• Regular languages, regular expressions, and DFAs are limited
  - Can DFA count? As in $L = \{ p^kq^k \}$ or $L = \{ wcw^R \mid w \in \Sigma^* \}$
Why not use regular expressions to specify PL syntax?

• Regular languages, regular expressions, and DFAs are limited
  - A DFA cannot count, as in $L = \{ p^kq^k \}$ or $L = \{ wcw^R \mid w \in \Sigma^* \}$
Specifying Syntax: Context-Free Grammars

Why not use regular expressions to specify PL syntax?

• Regular languages, regular expressions, and DFAs are limited
  - A DFA cannot count, as in \( L = \{ p^kq^k \} \) or \( L = \{ wcw^R \mid w \in \Sigma^* \} \)
  - Finite states means no counting of unbounded event
  - Matching items in this fashion requires a stack
    - Push ‘p’s onto the stack; pop them off to match ‘q’s
    - Empty stack at the end indicates an equal number of ‘p’s and ‘q’s

• Neither of these is a regular language
Specifying Syntax: Context-Free Grammars

Why not use regular expressions to specify PL syntax?

• Regular languages, regular expressions, and DFAs are limited
  - A DFA cannot count, as in \( L = \{ p^k q^k \} \) or \( L = \{ wcw^R \mid w \in \Sigma^* \} \)
  - Finite states means no counting of unbounded event
  - Matching items in this fashion requires a stack
    - Push ‘p’s onto the stack; pop them off to match ‘q’s
    - Empty stack at the end indicates an equal number of ‘p’s and ‘q’s

• Neither of these is a regular language

Does this limitation matter?
- Since \( L = \{ p^k q^k \} \) is the essence of matching parentheses, brackets, and the like, this limitation is an important one
- Every programming language I know has some matching construct

Bottom line: REs for spelling, CFGs for syntax
The point of parsing is to discover a grammatical derivation for a sentence

A derivation consists of a series of rewrite steps

\[ S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \quad \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \text{ sentence} \]

- \( S \) is the start symbol of the grammar
- Each \( \gamma_i \) is a sentential form
  - If \( \gamma \) contains only terminal symbols, \( \gamma \) is a sentence in \( L(G) \)
  - If \( \gamma \) contains 1 or more non-terminals, \( \gamma \) is a sentential form
- To get \( \gamma_i \) from \( \gamma_{i-1} \), expand some NT \( A \in \gamma_{i-1} \) by using \( A \rightarrow \beta \)
  - Replace the occurrence of \( A \in \gamma_{i-1} \) with \( \beta \) to get \( \gamma_i \)
  - Replacing the leftmost NT at each step, creates a leftmost derivation
  - Replacing the rightmost NT at each step, creates a rightmost derivation

\textbf{NT} means nonterminal symbol
The point of parsing is to discover a grammatical derivation for a sentence.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>List</td>
</tr>
<tr>
<td>3</td>
<td>Pair</td>
</tr>
<tr>
<td>5</td>
<td>( )</td>
</tr>
</tbody>
</table>

Derivation of “( )”

1. Goal → List
2. List → List Pair
3. Pair → ( List )
4. Pair → ( )

Parentheses Grammar
The point of parsing is to discover a grammatical derivation for a sentence.

### Parentheses Grammar

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Derivation of "( )"
The point of parsing is to discover a grammatical derivation for a sentence.
The point of parsing is to discover a grammatical derivation for a sentence.
The point of parsing is to discover a grammatical derivation for a sentence.

Derivation of "( )"
The point of parsing is to discover a grammatical derivation for a sentence.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>→</td>
<td>Goal</td>
</tr>
<tr>
<td>1</td>
<td>List</td>
</tr>
<tr>
<td>3</td>
<td>Pair</td>
</tr>
<tr>
<td>5</td>
<td>( )</td>
</tr>
</tbody>
</table>

Derivation of "( )"
The point of parsing is to discover a grammatical derivation for a sentence.

We denote this particular derivation: \( \text{Goal} \Rightarrow^* \{ ( ) \} ( ) \)

**Parentheses Grammar**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal</td>
</tr>
<tr>
<td>2</td>
<td>List</td>
</tr>
<tr>
<td>3</td>
<td>Pair</td>
</tr>
<tr>
<td>4</td>
<td>Pair \rightarrow { List }</td>
</tr>
<tr>
<td>5</td>
<td>Pair \rightarrow { }</td>
</tr>
</tbody>
</table>
The point of parsing is to discover a grammatical derivation for a sentence.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>List</td>
</tr>
<tr>
<td>2</td>
<td>List Pair</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Pair</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Derivation of \( ((( ))) ( ) \)

1. Goal \( \rightarrow \) List
2. List \( \rightarrow \) List Pair
3. \(| \) Pair
4. Pair \( \rightarrow \) \{ List \}
5. \(| \) \}

Parentheses Grammar

We denote this particular derivation: Goal \( \Rightarrow^* (( )) (( )) \)
The point of parsing is to discover a grammatical derivation for a sentence.

**Parentheses Grammar**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal</td>
</tr>
<tr>
<td>2</td>
<td>List</td>
</tr>
<tr>
<td>3</td>
<td>List Pair</td>
</tr>
<tr>
<td>4</td>
<td>Pair</td>
</tr>
<tr>
<td>5</td>
<td>List { }</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

We denote this particular derivation: $\text{Goal} \Rightarrow^* \{ ( ) \} ( )$
The point of parsing is to discover a grammatical derivation for a sentence.

Parsing

We denote this particular derivation: \( \text{Goal} \Rightarrow^* \{ \{ \} \} \{ \} \)
The point of parsing is to discover a grammatical derivation for a sentence

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Goal</td>
</tr>
<tr>
<td>1</td>
<td>List</td>
</tr>
<tr>
<td>2</td>
<td>List Pair</td>
</tr>
<tr>
<td>3</td>
<td>Pair</td>
</tr>
<tr>
<td>4</td>
<td>{ List }</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Parentheses Grammar

1. Goal → List
2. List → List Pair
3. | Pair
4. Pair → { List }
5. | { }

We denote this particular derivation: Goal ⇒* { { } } { }
The point of parsing is to discover a grammatical derivation for a sentence.

The Parentheses Grammar:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>Goal</td>
</tr>
<tr>
<td>1</td>
<td>List</td>
</tr>
<tr>
<td>2</td>
<td>List Pair</td>
</tr>
<tr>
<td>3</td>
<td>Pair { }</td>
</tr>
<tr>
<td>4</td>
<td>{ List } { }</td>
</tr>
<tr>
<td>5</td>
<td>{ Pair } { }</td>
</tr>
</tbody>
</table>

Derivation of \( ((( ))( )) \)

We denote this particular derivation: \( \text{Goal} \Rightarrow^* ((( ))( )) \)
The point of parsing is to discover a grammatical derivation for a sentence.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>Goal</td>
</tr>
<tr>
<td>1</td>
<td>List</td>
</tr>
<tr>
<td>2</td>
<td>List Pair</td>
</tr>
<tr>
<td>3</td>
<td>Pair ( )</td>
</tr>
<tr>
<td>4</td>
<td>( List ) ( )</td>
</tr>
<tr>
<td>5</td>
<td>( ) ( )</td>
</tr>
</tbody>
</table>

Derivation of ( ( ) ) ( )

We denote this particular derivation: Goal \( \Rightarrow^* ( ( ) ) ( ) \)
A derivation corresponds to a *parse tree* or *syntax tree*.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>List</td>
</tr>
<tr>
<td>2</td>
<td>List Pair</td>
</tr>
<tr>
<td>3</td>
<td>Pair { }</td>
</tr>
<tr>
<td>4</td>
<td>{ List } { }</td>
</tr>
<tr>
<td>5</td>
<td>{ { } } { }</td>
</tr>
</tbody>
</table>

Derivation of `{ { } } { }`
A derivation corresponds to a parse tree or syntax tree

The derivation gives us the grammatical structure of the input sentence, which was missing in the DFA / RE based recognizers.
Two Categories of Systematic Parsers

**Top-down parsers**  
(\textit{LL}(1), recursive descent — see EAC § 3.3)  
- Start with the root of the parse tree and grow toward the leaves  
- At each step, pick a production & try to match the input  
- Bad “pick” ⇒ may need to backtrack  
- Some grammars are “backtrack free”

**Bottom-up parsers**  
(\textit{LR}(1), operator precedence — see EAC § 3.4)  
- Start at the leaves and grow toward the root  
- As input is consumed, encode possibilities in an internal state  
- Start in a state valid for legal first tokens  
- We can make the process deterministic
Two Categories of Systematic Parsers

**Top-down parsers** (LL(1), recursive descent — see EAC § 3.3)
- Start with the root of the parse tree and grow toward the leaves
- At each step, pick a production & try to match the input
- Bad “pick” ⇒ may need to backtrack
- Some grammars are “backtrack free”

**Bottom-up parsers** (LR(1), operator precedence — see EAC § 3.4)
- Start at the leaves and grow toward the root
- As input is consumed, encode possibilities in an internal state
- Start in a state valid for legal first tokens
- We can make the process deterministic
Bottom-up Parsers

How does a bottom-up parser build (discover) a derivation?

• The parser uses a stack to hold grammar symbols, both terminal symbols & non-terminal symbols
• *(In essence)*, it pushes tokens onto a stack until the stack holds the right-hand side *(rhs)* of some production *(rewrite rule)*
• When it finds an rhs at the top of the stack, it rewrites the rhs with the rule’s lhs *(called a “reduction” or a “reduce action”)*

How does it recognize an rhs, or *(more precisely)* the next rhs in the derivation of the input sentence?

• For the moment, assume that we have an oracle to answer that question
• Applying the oracle to the stack yields one of three results:
  • A production <lhs:rhs> whose rhs is at the top of the stack
  • An indication that the stack is consistent with some future res
  • An indication that the stack cannot lead to an rhs
A conceptual bottom-up, shift-reduce parser

push $ onto the stack
word ← NextToken()

repeat until (top of stack = S and word = EOF)
    action ← oracle(stack, word)
    if (action is “future” & word ≠ EOF) then
        push word
        word ← NextToken()
    else if (action is <lhs:rhs>) then
        pop |rhs| symbols off the stack
        push lhs onto the stack
    else
        report an error
        break out of loop

report success

Here:

$ is used as an invalid symbol

S is the goal symbol of the grammar G

push() and pop() implement a simple stack
A conceptual bottom-up, shift-reduce parser

A shift-reduce parser has four kinds of actions:

**Shift:** next word is moved from input to the stack

**Reduce:**
- `TOS` is `rhs` of a reduction
- Pop `rhs` off the stack
- Push `lhs` onto the stack

**Error:** report the problem to user

**Accept:** (normal exit from loop) report success and stop parsing

---

```
push $ onto the stack
word ← NextToken()
repeat until (top of stack = S and word = EOF)
  action ← oracle(stack, word)
  if (action is “future” & word ≠ EOF) then
    push word
    word ← NextToken()
  else if (action is `<lhs:rhs>`) then
    pop |rhs| symbols off the stack
    push lhs onto the stack
  else
    report an error
    break out of loop
report success
```
A conceptual bottom-up, shift-reduce parser

A shift-reduce parser has four kinds of actions:

**Shift**: next word is moved from input to the stack

**Reduce**: TOS is rhs of a reduction
- pop rhs off the stack
- push lhs onto the stack

**Error**: report the problem to user

**Accept**: (normal exit from loop)
- report success and stop parsing

---

push $ onto the stack

word ← NextToken()

repeat until (top of stack = S and word = EOF)

- action ← oracle(stack, word)
- if (action is “future” & word ≠ EOF) then
  - push word
  - word ← NextToken()  
  else if (action is $lhs:rhs$) then
    - pop $|rhs|$ symbols off the stack
    - push lhs onto the stack
  elset
    - report an error
    - break out of loop

report success
A conceptual bottom-up, shift-reduce parser

A shift-reduce parser has four kinds of actions:

**Shift**: next word is moved from input to the stack

**Reduce**: TOS is rhs of a reduction
- Pop rhs off the stack
- Push lhs onto the stack

**Error**: report the problem to user
- Report an error
- Break out of loop

**Accept**: (normal exit from loop)
- Report success and stop parsing

---

`push $ onto the stack`

`word ← NextToken()`

`repeat until (top of stack = S and word = EOF)`

`action ← oracle(stack, word)`

`if (action is “future” & word ≠ EOF) then`

`push word`

`word ← NextToken()`

`else if (action is <lhs:rhs>) then`

`pop |rhs| symbols off the stack`

`push lhs onto the stack`

`else`

`report an error`

`break out of loop`

`report success`
A conceptual bottom-up, shift-reduce parser

A shift-reduce parser has four kinds of actions:

**Shift:** next word is moved from input to the stack

**Reduce:** TOS is rhs of a reduction
- pop rhs off the stack
- push lhs onto the stack

**Error:** report the problem to user

**Accept:** (normal exit from loop)
- report success and stop parsing

---

**push $ onto the stack**

word ← NextToken()

repeat until (top of stack = S and word = EOF)
- action ← oracle(stack, word)
  - if (action is “future” & word ≠ EOF) then
    - push word
    - word ← NextToken()
  - else if (action is < lhs:rhs>) then
    - pop |rhs| symbols off the stack
    - push lhs onto the stack
  - else
    - report an error
    - break out of loop
- report success
A conceptual bottom-up, shift-reduce parser

A shift-reduce parser has four kinds of actions:

**Shift:** next word is moved from input to the stack

**Reduce:** TOS is rhs of a reduction
- pop rhs off the stack
- push lhs onto the stack

**Error:** report the problem to user

**Accept:** (normal exit from loop)
- report success and stop parsing

**Shift, Accept, & Error** are $O(1)$

**Reduce** is $O(|rhs|)$

---

push $\$ onto the stack
word $\leftarrow$ NextToken()

repeat until (top of stack = S and word = EOF)
  action $\leftarrow$ oracle(stack,word)
  if (action is “future” & word ≠ EOF) then
    push word
    word $\leftarrow$ NextToken()
  else if (action is $<\text{lhs:rhs}>$) then
    pop $|\text{rhs}|$ symbols off the stack
    push lhs onto the stack
  else
    report an error
    break out of loop

report success
A conceptual bottom-up, shift-reduce parser

Consider the input stream “( )”

<table>
<thead>
<tr>
<th>Step</th>
<th>Stack</th>
<th>Word</th>
<th>Oracle’s answer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$</td>
<td>(</td>
<td>some future rhs</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Goal $\rightarrow$ List
2. List $\rightarrow$ List Pair
3. Pair $\rightarrow$ ( List )
4. Pair $\rightarrow$ ( )
A conceptual bottom-up, shift-reduce parser

Consider the input stream “( )”

<table>
<thead>
<tr>
<th>Step</th>
<th>Stack</th>
<th>Word</th>
<th>Oracle’s answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$</td>
<td>(</td>
<td>some future rhs</td>
</tr>
<tr>
<td>1</td>
<td>$ (</td>
<td>)</td>
<td>some future rhs</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Goal $\rightarrow$ List
2. List $\rightarrow$ List Pair
3. $|$ Pair
4. Pair $\rightarrow$ ( List )
5. $|$ ( )
A conceptual bottom-up, shift-reduce parser

Consider the input stream “( )”

<table>
<thead>
<tr>
<th>Step</th>
<th>Stack</th>
<th>Word</th>
<th>Oracle’s answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$</td>
<td>(</td>
<td>some future rhs</td>
</tr>
<tr>
<td>1</td>
<td>$(</td>
<td>)</td>
<td>some future rhs</td>
</tr>
<tr>
<td>2</td>
<td>$(</td>
<td>EOF</td>
<td>&lt; Pair : ( ) &gt; 5</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Goal → List
2. List → List Pair
3. | Pair
4. Pair → ( List )
5. | ( )
A conceptual bottom-up, shift-reduce parser

Consider the input stream “( )”

<table>
<thead>
<tr>
<th>Step</th>
<th>Stack</th>
<th>Word</th>
<th>Oracle’s answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$</td>
<td>(</td>
<td>some future rhs</td>
</tr>
<tr>
<td>1</td>
<td>$( )</td>
<td>EOF</td>
<td>&lt; Pair : ( ) &gt; 5</td>
</tr>
<tr>
<td>2</td>
<td>$ Pair</td>
<td>EOF</td>
<td>&lt; List : Pair &gt; 3</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Goal $\rightarrow$ List
2. List $\rightarrow$ List Pair
3. $|$ $\rightarrow$ Pair
4. Pair $\rightarrow$ ( List )
5. $|$ ( )
A conceptual bottom-up, shift-reduce parser

Consider the input stream “( )”

<table>
<thead>
<tr>
<th>Step</th>
<th>Stack</th>
<th>Word</th>
<th>Oracle’s answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>$</td>
<td>(</td>
<td>some future rhs</td>
</tr>
<tr>
<td>1</td>
<td>$ (</td>
<td>)</td>
<td>some future rhs</td>
</tr>
<tr>
<td>2</td>
<td>$ (</td>
<td>EOF</td>
<td>&lt; Pair : ( ) &gt; 5</td>
</tr>
<tr>
<td>3</td>
<td>$ Pair</td>
<td>EOF</td>
<td>&lt; List : Pair &gt; 3</td>
</tr>
<tr>
<td>4</td>
<td>$ List</td>
<td>EOF</td>
<td>&lt; Goal : List &gt; 1</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A conceptual bottom-up, shift-reduce parser

Consider the input stream “( )”

<table>
<thead>
<tr>
<th>Step</th>
<th>Stack</th>
<th>Word</th>
<th>Oracle’s answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>$</td>
<td>(</td>
<td>some future rhs</td>
</tr>
<tr>
<td>1</td>
<td>$ (</td>
<td>)</td>
<td>some future rhs</td>
</tr>
<tr>
<td>2</td>
<td>$ (</td>
<td>EOF</td>
<td>&lt; Pair : ( ) &gt; 5</td>
</tr>
<tr>
<td>3</td>
<td>$ Pair</td>
<td>EOF</td>
<td>&lt; List : Pair &gt; 3</td>
</tr>
<tr>
<td>4</td>
<td>$ List</td>
<td>EOF</td>
<td>&lt; Goal : List &gt; 1</td>
</tr>
<tr>
<td>5</td>
<td>$ Goal</td>
<td>EOF</td>
<td>exits while loop &amp; reports success</td>
</tr>
</tbody>
</table>

1. Goal → List
2. List → List Pair
3.  | Pair
4. Pair → ( List )
5.  | ( )
More Shifts and Reduce

Consider the input stream “(( )) ()”

More complex example
• Key is recognizing the difference between a future, an error, and a and a reduction
• In an LR(1) parser, the oracle is encoded into two parse tables: named ACTION and GOTO

<table>
<thead>
<tr>
<th>Step</th>
<th>Stack</th>
<th>Word</th>
<th>Oracle’s answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$</td>
<td>(</td>
<td>some future rhs</td>
</tr>
<tr>
<td>1</td>
<td>$ (</td>
<td>(</td>
<td>some future rhs</td>
</tr>
<tr>
<td>2</td>
<td>$ ( (</td>
<td>)</td>
<td>some future rhs</td>
</tr>
<tr>
<td>3</td>
<td>$ ( (</td>
<td>)</td>
<td>Pair, ( ) &gt; 5</td>
</tr>
<tr>
<td>4</td>
<td>$ ( Pair</td>
<td>)</td>
<td>List, Pair &gt; 3</td>
</tr>
<tr>
<td>5</td>
<td>$ ( List</td>
<td>)</td>
<td>some future rhs</td>
</tr>
<tr>
<td>6</td>
<td>$ ( List</td>
<td>)</td>
<td>Pair, List &gt; 4</td>
</tr>
<tr>
<td>7</td>
<td>$ Pair</td>
<td>)</td>
<td>List, Pair &gt; 3</td>
</tr>
<tr>
<td>8</td>
<td>$ List</td>
<td>)</td>
<td>some future rhs</td>
</tr>
<tr>
<td>9</td>
<td>$ List (</td>
<td>)</td>
<td>some future rhs</td>
</tr>
<tr>
<td>10</td>
<td>$ List (</td>
<td>EOF</td>
<td>Pair, ( ) &gt; 5</td>
</tr>
<tr>
<td>11</td>
<td>$ List Pair</td>
<td>EOF</td>
<td>List, List Pair &gt; 2</td>
</tr>
<tr>
<td>12</td>
<td>$ List</td>
<td>EOF</td>
<td>Goal, List &gt; 1</td>
</tr>
<tr>
<td>13</td>
<td>$ Goal</td>
<td>EOF</td>
<td>exits while loop &amp; reports success</td>
</tr>
</tbody>
</table>

Consider the input stream “(( )) ()”

More complex example
• Key is recognizing the difference between a future, an error, and a and a reduction
• In an LR(1) parser, the oracle is encoded into two parse tables: named ACTION and GOTO

<table>
<thead>
<tr>
<th>Step</th>
<th>Stack</th>
<th>Word</th>
<th>Oracle’s answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$</td>
<td>(</td>
<td>some future rhs</td>
</tr>
<tr>
<td>1</td>
<td>$ (</td>
<td>(</td>
<td>some future rhs</td>
</tr>
<tr>
<td>2</td>
<td>$ ( (</td>
<td>)</td>
<td>some future rhs</td>
</tr>
<tr>
<td>3</td>
<td>$ ( (</td>
<td>)</td>
<td>Pair, ( ) &gt; 5</td>
</tr>
<tr>
<td>4</td>
<td>$ ( Pair</td>
<td>)</td>
<td>List, Pair &gt; 3</td>
</tr>
<tr>
<td>5</td>
<td>$ ( List</td>
<td>)</td>
<td>some future rhs</td>
</tr>
<tr>
<td>6</td>
<td>$ ( List</td>
<td>)</td>
<td>Pair, List &gt; 4</td>
</tr>
<tr>
<td>7</td>
<td>$ Pair</td>
<td>)</td>
<td>List, Pair &gt; 3</td>
</tr>
<tr>
<td>8</td>
<td>$ List</td>
<td>)</td>
<td>some future rhs</td>
</tr>
<tr>
<td>9</td>
<td>$ List (</td>
<td>)</td>
<td>some future rhs</td>
</tr>
<tr>
<td>10</td>
<td>$ List (</td>
<td>EOF</td>
<td>Pair, ( ) &gt; 5</td>
</tr>
<tr>
<td>11</td>
<td>$ List Pair</td>
<td>EOF</td>
<td>List, List Pair &gt; 2</td>
</tr>
<tr>
<td>12</td>
<td>$ List</td>
<td>EOF</td>
<td>Goal, List &gt; 1</td>
</tr>
<tr>
<td>13</td>
<td>$ Goal</td>
<td>EOF</td>
<td>exits while loop &amp; reports success</td>
</tr>
</tbody>
</table>
How does this work?

An LR(1) parser encodes state information on the stack

- That state information threads together the reduce actions to ensure that the parser builds a derivation.
  - At each point, the parser is in a state that represents all of the possible outcomes
  - From the combination of the state, the stack, and the next word, it can decide whether to **shift**, **reduce**, **accept**, or **throw an error** (We will see examples.)

All of that knowledge is encoded into the ACTION and GOTO tables

The LR(1) Table construction simulates the set of parser states when it builds the ACTION and GOTO tables
See § 3.4 of EAC book
An LR(1) Skeleton Parser

```c
stack.push(INVALID);
stack.push(s0);                // initial state
word ← NextWord();
loop forever {
    s ← stack.top();
    if (ACTION[s,word] == "reduce A → β") then {
        stack.popnum(2 * |β|); // pop RHS off stack
        s ← stack.top();
        stack.push(A);        // push LHS, A
        stack.push(GOTO[s,A]); // push next state
    }
    else if (ACTION[s,word] == "shift s_i") then {
        stack.push(word); stack.push(s_i);
        word ← NextWord();
    }
    else if (ACTION[s,word] == "accept"
             & word == EOF)
        then break;
    else throw a syntax error;
} report success;
```

**The Skeleton LR(1) Parser**

- Follows the basic shift-reduce scheme from previous slides
- Relies on a stack & a scanner
- Stacks `<symbol,state>` pairs
- **ACTION** table encodes the shift, reduce, accept, error decision
- **GOTO** threads reduce actions together to form potential sentences in \( L(G) \)
- Shifts \(|words|\) times
- Reduces \(|derivation|\) times
- Accepts at most once
- Detects errors at the earliest possible point
LR(1) Tables for the Parenthesis Grammar

<table>
<thead>
<tr>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
</tr>
<tr>
<td>$s_0$</td>
</tr>
<tr>
<td>$s_1$</td>
</tr>
<tr>
<td>$s_2$</td>
</tr>
<tr>
<td>$s_3$</td>
</tr>
<tr>
<td>$s_4$</td>
</tr>
<tr>
<td>$s_5$</td>
</tr>
<tr>
<td>$s_6$</td>
</tr>
<tr>
<td>$s_7$</td>
</tr>
<tr>
<td>$s_8$</td>
</tr>
<tr>
<td>$s_9$</td>
</tr>
<tr>
<td>$s_{10}$</td>
</tr>
<tr>
<td>$s_{11}$</td>
</tr>
<tr>
<td>$s_{12}$</td>
</tr>
<tr>
<td>$s_{13}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
</tr>
<tr>
<td>$s_0$</td>
</tr>
<tr>
<td>$s_1$</td>
</tr>
<tr>
<td>$s_2$</td>
</tr>
<tr>
<td>$s_3$</td>
</tr>
<tr>
<td>$s_4$</td>
</tr>
<tr>
<td>$s_5$</td>
</tr>
<tr>
<td>$s_6$</td>
</tr>
<tr>
<td>$s_7$</td>
</tr>
<tr>
<td>$s_8$</td>
</tr>
<tr>
<td>$s_9$</td>
</tr>
<tr>
<td>$s_{10}$</td>
</tr>
<tr>
<td>$s_{11}$</td>
</tr>
<tr>
<td>$s_{12}$</td>
</tr>
<tr>
<td>$s_{13}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action Table Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry</td>
</tr>
<tr>
<td>s</td>
</tr>
<tr>
<td>r</td>
</tr>
<tr>
<td>acc</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

1. Goal $\rightarrow$ List
2. List $\rightarrow$ List Pair
3. $|$ Pair
4. Pair $\rightarrow$ ( List )
5. $|$ ( )
The Parenthesis Language: Parsing "( )"

-Trace of the Skeleton Parser’s Action on "( )"

<table>
<thead>
<tr>
<th>State</th>
<th>Lookahead</th>
<th>Stack</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>(</td>
<td>$0</td>
<td>—none—</td>
<td>—</td>
</tr>
<tr>
<td>0</td>
<td>(</td>
<td>$0</td>
<td>—none—</td>
<td>shift 3</td>
</tr>
<tr>
<td>3</td>
<td>)</td>
<td>$0 { 3 }</td>
<td>—none—</td>
<td>shift 8</td>
</tr>
<tr>
<td>8</td>
<td>EOF</td>
<td>$0 { 3 } 8</td>
<td>Pair → {  }</td>
<td>reduce 5</td>
</tr>
<tr>
<td>2</td>
<td>EOF</td>
<td>$0 Pair 2</td>
<td>List → Pair</td>
<td>reduce 3</td>
</tr>
<tr>
<td>1</td>
<td>EOF</td>
<td>$0 List 1</td>
<td>Goal → List</td>
<td>accept</td>
</tr>
</tbody>
</table>
Summary

- **LR(1) Parsers**
  - The parser generator builds a model of all possible states of the parser
  - If, in each state, the **shift/reduce/accept** decision can be made with just the left context and 1 word of lookahead, the grammar is an **LR(1)** grammar.
    - There are many grammars that are not **LR(1)** grammars
    - There are **LR(1)** grammars for all deterministically parsable languages
  - We will look at the kinds of problems that make a grammar non-**LR(1)**

- **Properties we want in a grammar**
  - Grammar must be unambiguous
  - Grammar must enforce desired structure or meaning
    - For example, precedence in arithmetic expressions
      - $1 + 2 * 3 = 7$, not 9
  - Grammar must have that “odd” 1 word lookahead property
  - Some properties are language design, some are grammar design
Reading

• **Engineering A Compiler (EAC)**
  - Chapter 2.4 (Generation of automaton and the code)
  - Chapter 2.4.2 (Thompson’s Construction)
  - Chapter 2.4.3 (Subset Construction)
  - Chapter 2.4.4 and Chapter 2.6.3 (DFA Minimization)
  - Chapter 2.5 (Implement an Automaton)
  - Chapter 3.2.1 (Use of CFG for PL syntax)
  - Chapter 3.3 (Top-down parsers)
  - Chapter 3.4 (Bottom-up parsers)