CSE 4251: Compiler Construction

Ryan Stansifer
Department of Computer Sciences
Florida Institute of Technology
Melbourne, Florida USA 32901

http://www.cs.fit.edu/~ryan/

computer-presentation edition

February 14, 2013
Formal Languages

See my notes on Programming Languages
Chapter 2: Lexical Analysis
Chapter 2: Lexical Analysis

2.1. Lexical Tokens

See my notes on Programming Languages
Chapter 2: Lexical Analysis

2.2 Regular Expressions

See my notes on Programming Languages
Chapter 2: Lexical Analysis

2.3 Finite Automata
Deterministic Finite Automata

A good property of regular expressions is that they can be easily recognized. This means that a computer program can be written for every regular expression which when given a string over the alphabet can efficiently decide if the string is or is not in the language described by the regular expression.

A simple variation of this program can be used convert a string of characters to a string of token when every kind of token is described by a regular expression.

The abstract algorithm that does the recognition is called a deterministic finite automaton.
Deterministic Finite Automata

Our plan is to algorithmically convert any regular expression to a finite automaton.
Deterministic Finite Automata

A finite automaton, has a finite set of states, edges each leading from one state to another, and each edges is labeled with a symbol. One state is the start state, and a certain set of states are distinguished as final states.

Examples: Figure 2.3, page 21.

An edge labeled with more than one character is shorthand for many parallel edges; so in the ID machine there are really 26 edges each leading from state 1 to 2, each labeled by a different letter.
Deterministic Finite Automata

In a deterministic finite automaton (DFA), no two edges leaving from the same state are labeled with the same symbol.

A DFA recognizes an input string (an hence defines a formal language) by computing on the input string until it reaches an accept state or a reject state. A DFA accepts or rejects a string as follows. Starting in the start state, for each character in the input string the automaton follows exactly one edge to get to the next state. The edge must be labeled with the input character. After making $n$ transitions for an $n$-character string, if the automaton is in a final state, then it accepts the string. If it is not in a final state, or if at some point there was not an appropriately labeled edge to follow, it rejects the string.

The language recognized by an automaton is the set of strings that it accepts.
Encoding Finite Automata

We can encode an automaton as a transition matrix; a two-dimensional array indexed by state number and input character. There will be a “dead” state (state 0) that loops to itself on all characters; we use this state to encode the absence of an edge.

```c
int edges[][ ] = { /* ... 0 1 2 ... e f g h i j ... */
/* state 0 */ {0,0,... 0,0,0,... 0,0,0,0,0,0,...},
/* state 1 */ {0,0,... 7,7,7,... 4,4,4,4,2,4,...},
/* state 2 */ {0,0,... 4,4,4,... 4,3,4,4,4,4,...},
/* state 3 */ {0,0,... 4,4,4,... 4,4,4,4,4,4,...},
/* state 4 */ {0,0,... 4,4,4,... 4,4,4,4,4,4,...},
/* state 5 */ {0,0,... 6,6,6,... 0,0,0,0,0,0,...},
/* state 6 */ {0,0,... 6,6,6,... 0,0,0,0,0,0,...},
/* state 8 */ {0,0,... 8,8,8,... 0,0,0,0,0,0,...},
and so on
}
```

There must also be an array indicating which states are final.
current_state := S;
while not endof input stream loop
    current_input_character := next character in input stream
    current_state := edges[current_state][current_input_character]
end loop;
if (final_state (current_state)) then
    Accept
else
    Reject
end if;
In Practice

Lexical analysis is not pure and simple language recognition.

One, we want to find the next token in the input stream, i.e., a prefix of the remaining input stream and not the whole input stream. Two, there is not just one kind of token, rather many kinds of token each defined as a language of characters using a regular expression.
How can separate automata be combined into a single automaton that can serve as a lexical analyzer?

We have a single start state (big disjunction of all the individual automata), and each final state must be labeled with the token-type that it accepts.
These rules are bit ambiguous. For example, does if8 match as a single identifier or as the two tokens if and 8? Does the string if 89 begin with an identifier or a reserved word?

So, traditionally:

**longest match** The longest initial sub-string of the input that can be matched is taken as the next token.

**rule priority** For a particular longest initial sub-string, the first regular expression determines its token type.
Recognizing the longest automata.

Appel, 2nd, Figure 2.5, page 24.
last_final_state := 0
token_start := 0;
token_end := 0;

current_state := S;
buffer_index :=0; -- begining of buffer
while buffer_index < length(input buffer) loop
  current_input_character := input_buffer[buffer_index++];
  current_state := edges[current_state][current_input_character];
  if current_state=0 then
    report token kind(last_final_state) input_buffer[start..end
    last_final_state := 0;
    buffer_index := token_end + 1;
    token_start := buffer_index;
    token_end := buffer_index;
    current_state := S;
  elsif (final_state (current_state)) then
    last_final_state = current_state;
    token_end = buffer_index-1;
end if;
end loop;
Finally, a lexical specification should always be *complete*, always matching some initial sub-string of the input; we can always achieve this by having a rule that matches any single character (and in this case, prints an “illegal character” error message and continues.
Deterministic Finite Automata

Our plan is to algorithmically convert any regular expression to a finite automaton hits a snag, and we find it simpler to break the problem into two parts: converting a regular expression to an NFA and convert an NFA to a DFA.
Chapter 2: Lexical Analysis

2.4 Non-deterministic Finite Automata
Non-deterministic Finite Automata

A non-deterministic finite automaton, or NFA for short, is a DFA that may have a choice of edges—labeled with the same symbol—to follow out of a state, or it may have a edges labeled with $\epsilon$ (the Greek letter epsilon) that can be followed without consuming any symbol from the input.

NFAs are useful because it is easy to convert a regular expression to a NFA.
NFA

An example NFA. Louden, Example 2.10, page 58.
An example NFA. Galles, Figure B.5, page 336. A NFA that accepts all strings over \( \{a, b, c\} \) that do not contain an \( a \), or do not contain a \( b \), or do not contain a \( c \).
Appel, exercise 2.3(a), page 36. Explain in informal English what this DFA over $\Sigma = \{0, 1\}$ recognizes.
Exercise 2.3(b), page 37. Explain in informal English what this FSA recognizes.
Converting a Regular Expression to an NFA

Thompson construction. Appel does it slightly differently with a modified NFA—this saves a few needless states.
Modified NFA

A modified NFA has a labeled “tail” and a unique “ending” state.

A modified NFA can be converted to a NFA by adding a unique start state and choosing the ending state as the final state.
Thompson’s Construction

A recursive construction of a NFA (actually a modified NFA).

- $a$ a rule for a symbol of the alphabet
- $\epsilon$
- $r_1 \cdot r_2$
- $r_1 \mid r_2$
- $r^*$
- $r^+$ equivalent to $r \cdot r^*$
- $r?$ equivalent to $r \mid \epsilon$
- $[abc]$ equivalent to $a \mid b \mid c$
Thompson’s Construction: Case $a$
Thompson’s Construction: Case $\epsilon$
Thompson’s Construction: Case $r_1 | r_2$

Start with the modified NFAs for $r_1$ and for $r_2$ (created recursively).
Thompson’s Construction: Case $r_1 \mid r_2$

Add a state to the new NFA labeled 1.
Thompson’s Construction: Case $r_1 \mid r_2$

Connect both tails the modified NFAs to the state labeled 1.
Thompson’s Construction: Case $r_1 \mid r_2$

Add the state labeled $n$ to the new NFA.
Thompson’s Construction: Case $r_1 \mid r_2$

Add two edges labeled $\epsilon$ from the designated ends of the NFAs for $r_1$ and $r_2$ to the state labeled $n$. 
Thompson’s Construction: Case $r_1 \mid r_2$

Pick state $n$ as the designated end of the modified NFA for $r_1 \mid r_2$. 

![Diagram of Thompson's Construction](image)
Thompson’s Construction: Case $r_1 \mid r_2$

Add a tail labeled $\epsilon$ for the modified NFA for $r_1 \mid r_2$. This completes the modified NFA for $r_1 \mid r_2$. 

![Diagram of NFA with labeled transitions]
Thompson’s Construction: Case $r_1 \cdot r_2$
Thompson’s Construction: Case $r^*$

Start with the modified NFA for $r$ (created recursively).
Thompson’s Construction: Case $r^*$

Create a node labeled $n$ for the new NFA.
Thompson’s Construction: Case $r^*$

Make an edge labeled $\epsilon$ from the designated end of the modified NFA for $r$ to the state labeled $n$. 
Thompson’s Construction: Case $r^*$

The tail of the modified NFA for $r$ is set to come from state $n$. 

![Diagram](image_url)
Thompson’s Construction: Case $r^*$

Pick state $n$ as the designated end of the modified NFA for $r^*$. 
Thompson’s Construction: Case $r^*$

Add tail labeled $\epsilon$ for modified NFA for $r^*$. This completes the modified NFA for $r^*$. 
NFA construction

Java applet: excellent visualization of NFA construction.

http://www.cs.kent.ac.uk/people/staff/smk/regexp/gui.html

Use abc|+*% and don’t forget to enter return.
Appel, 2nd, Exercise 2.4(b), page 34. Convert \( a((b \mid b^*c)d)^* \mid d^*a \) to a NFA.
Exercise 2.4(b), page 37. Convert \( a((b \mid b^*c)d)^* \mid d^*a \) to a NFA. Step 1 apply the rule for \( \mid \).
Exercise 2.4(b), page 37. Convert $a((b \mid b^*c)d)^* \mid d^*a$ to a NFA.

Step 2 apply the rule for $\cdot$. 

![Diagram of NFA](image-url)
Exercise 2.4(b), page 37. Convert $a((b \mid b^*c)d)^* \mid d^*a$ to a NFA. Step 3 apply the rule for $a$. 

\[ a((b \mid b^*c)d)^* \mid d^*a \]
Exercise 2.4(b), page 37. Convert $a((b | b^*c)d)^* \mid d^*a$ to a NFA. Step 4 apply the rule for $\cdot$. 
Exercise 2.4(b), page 37. Convert $a((b \mid b^*c)d)^* \mid d^*a$ to a NFA. Step 5 apply the rule for $a$. 

![Diagram of NFA for the given expression]
Exercise 2.4(b), page 37. Convert \( a((b | b^*c)d)^* | d^*a \) to a NFA.
Step 6 apply the rule for \( * \).
Exercise 2.4(b), page 37. Convert \( a((b \mid b^* c)d)^* \mid d^* a \) to a NFA. Step 7 apply the rule for \( d \).
Exercise 2.4(b), page 37. Convert $a((b \mid b^*c)d)^* \mid d^*a$ to a NFA. Step 8 apply the rule for \(*\).
Exercise 2.4(b), page 37. Convert $a((b \mid b^*c)d)^* \mid d^*a$ to a NFA. 
Step 9 apply the rule for $\cdot$. 
Exercise 2.4(b), page 37. Convert $a((b \mid b^*c)d)^* \mid d^*a$ to a NFA.
Step 10 apply the rule for $d$. 
Exercise 2.4(b), page 37. Convert \(a((b \mid b^*c)d)^* \mid d^*a\) to a NFA. Step 11 apply the rule for \(|\).
Exercise 2.4(b), page 37. Convert $a((b \mid b^*c)d)^* \mid d^*a$ to a NFA.

Step 12 apply the rule for $b$. 
Exercise 2.4(b), page 37. Convert $a((b \mid b^*c)d)^* \mid d^*a$ to a NFA. 
Step 13 apply the rule for $\cdot$. 

![Diagram of NFA](image_url)
Exercise 2.4(b), page 37. Convert $a((b \mid b^*c)d)^* \mid d^*a$ to a NFA. Step 14 apply the rule for $c$. 

![Diagram of NFA](image-url)
Exercise 2.4(b), page 37. Convert $a((b \mid b^* c)d)^* \mid d^* a$ to a NFA. Step 15 apply the rule for $\ast$. 

![NFA Diagram]
Exercise 2.4(b), page 37. Convert $a((b | b^*c)d)^* d^*a$ to a NFA.
Step 16 apply the rule for $b$. 

![NFA Diagram]
Appel, Exercise 2.4(b), page 37. Convert $a((b \mid b^*c)d)^* \mid d^*a$ to a NFA. Step 17: add the initial state and make the “ending” state as final.
We are almost done.

We have a finite-automaton/scanner engine—the same engine for every table. We can convert a regular-expression to an NFA. All that is left is converting an NFA to a DFA. (And then encode the graph into a table.)
Converting an NFA to a DFA
Converting an NFA to a DFA

Key: Imagine a DFA that simulates an NFA by being in many of the states of the NFA simultaneously. The states of a DFA correspond to sets of states of the NFS.

Definition of $\varepsilon$-closure.

Definition of DFAedge.
Converting an NFA to a DFA

Definition of $\epsilon$-closure.

Let $\text{edge}(s,c)$ be the set of all NFA states reachable by following a single edge with label $c$ from state $s$ in the NFA.

We define $\epsilon$—closure($S$) to be the set $T$ where $T$ satisfies

$$T = S \cup \left( \bigcup_{s \in T} \text{edge}(s, \epsilon) \right)$$

$T := S$

repeat

$T' := T$

$T := T' \cup \left( \bigcup_{s \in T'} \text{edge}(s, \epsilon) \right)$

until $T = T'$
Why does this algorithm work? $T$ can only grow in each iteration, so the final $T$ must include $S$. If $T = T'$ after an iteration step, then $T$ must also include $\bigcup_{s \in T'} \text{edge}(s, \epsilon)$. Finally, the algorithm must terminate, because there are only a finite number of distinct states in the NFA.

\[
\text{DFAedge}(D, c) = \epsilon-\text{closure}(\bigcup_{s \in D} \text{edge}(s, c))
\]
Converting an NFA to a DFA

Algorithm to convert an NFA to a DFA.

\[ X := \text{e-closure of } \{s\} \]

add \( X \) as the start state to the DFA

add \( X \) to list

while list not empty

remove element from list, call it \( S \)

for each symbol \( \sigma \in \Sigma \)

\[ X := \text{DFAedge } (S, \sigma) \]

add a transition from \( S \) to \( X \) to the DFA

if \( X \) not in DFA then

add \( X \) to DFA

add \( X \) to list

end for

end while
The algorithm does not visit unreachable states of the DFA. This is extremely important, because in principle the DFA has $2^n$ states, but in practice we usually find the only about $n$ of them are reachable from the start state. It is important to avoid an exponential blowup in the size of the DFA interpreter’s transition tables, which will form part of the working compiler.
First example
Louden, exercise 2-14
Convert an NFA to DFA

Lounden, Exercise 2.14, page 92. Convert the NFA of Example 2.10 into a DFA using the subset construction.

An example NFA. Lounden, Example 2.10, page 58.
Convert an NFA to DFA

Louneden, Exercise 2.14, page 92. Convert the NFA of Example 2.10 into a DFA using the subset construction.

Each state of the DFA is a set of states of the NFA. The initial state of the DFA is the $\epsilon$-closure of the initial state of the NFA.

$$\text{CLOSE}\{0\} = \{0, 1, 3\} = S_0$$
Convert an NFA to DFA

Determine the transition function of the DFA on all inputs. Begin with the initial state $S_0$, and determine the transition on input $a$.

\[
\text{CLOSE}\{0\} = \{0, 1, 3\} = S_0
\]
\[
\delta(S_0, a) = \text{CLOSE}\{1, 2\}
\]
\[
\delta(S_0, b) = \]

\[
0, 1, 3
\]
The $\epsilon$-closure of the set $\{1, 2\}$ is $\{1, 2, 3\}$. This is a new state in the DFA, call it $S_1$.

\[
\text{CLOSE}\{0\} = \{0, 1, 3\} = S_0 \\
\delta(S_0, a) = \text{CLOSE}\{1, 2\} = \{1, 2, 3\} = S_1 \\
\delta(S_0, b) = 
\]
Convert an NFA to DFA

With the initial state $S_0$, determine the transition on input $b$.

\[
\text{CLOSE}\{0\} = \{0, 1, 3\} = S_0
\]
\[
\delta(S_0, a) = \text{CLOSE}\{1, 2\} = \{1, 2, 3\} = S_1
\]
\[
\delta(S_0, b) = \text{CLOSE}\{3\}
\]
Convert an NFA to DFA

The $\epsilon$-closure of the set $\{3\}$ is $\{1, 3\}$. This is a new state in the DFA, call it $S_2$.

$$\text{CLOSE}\{0\} = \{0, 1, 3\} = S_0$$
$$\delta(S_0, a) = \text{CLOSE}\{1, 2\} = \{1, 2, 3\} = S_1$$
$$\delta(S_0, b) = \text{CLOSE}\{3\} = \{1, 3\} = S_2$$
Determine the transition function of the DFA from state $S_1$ on inputs $a$ and $b$. On $a$ there is no where to go in the NFA, so we create a “sink” state for DFA.

$$
\begin{align*}
\delta(S_0, a) &= \text{CLOSE}\{1, 2\} = \{1, 2, 3\} = S_1 \\
\delta(S_0, b) &= \text{CLOSE}\{3\} = \{1, 3\} = S_2 \\
\delta(S_1, a) &= \text{CLOSE}\{} = \emptyset = S_3 \\
\delta(S_1, b) &=
\end{align*}
$$
Determine the transition function of the DFA from state $S_1$ on inputs $a$ and $b$. On $b$ we happen to transition to an existing state $S_2$.

$$\delta(S_0, a) = \text{CLOSE}\{1, 2\} = \{1, 2, 3\} = S_1$$
$$\delta(S_0, b) = \text{CLOSE}\{3\} = \{1, 3\} = S_2$$
$$\delta(S_1, a) = \text{CLOSE}\{\} = \emptyset = S_3$$
$$\delta(S_1, b) = \text{CLOSE}\{3\} = \{1, 3\} = S_2$$
Determine the transition from state $S_2$ on inputs $a$ and $b$.

$$\delta(S_0, a) = \text{CLOSE}\{1, 2\} = \{1, 2, 3\} = S_1$$
$$\delta(S_0, b) = \text{CLOSE}\{3\} = \{1, 3\} = S_2$$
$$\delta(S_1, a) = \text{CLOSE}\{\} = \emptyset = S_3$$
$$\delta(S_1, b) = \text{CLOSE}\{3\} = \{1, 3\} = S_2$$
$$\delta(S_2, a) = \text{CLOSE}\{\} = \emptyset = S_3$$
$$\delta(S_2, b) = \text{CLOSE}\{3\} = \{1, 3\} = S_2$$
Convert an NFA to DFA

Determining the transition from state $S_2$ on inputs $a$ and $b$ is easy; from the empty set of states there are no transitions in the NFA. In the DFA this is represented by a transition from the empty set back to itself.

![Diagram of NFA to DFA conversion]
Convert an NFA to DFA

The final states of the DFA are determined from the final states of the NFA. State 3 was the only final state in the NFA. Any set of NFA states containing a final state is a final state in the DFA.
Convert an NFA to DFA

The transition function of the constructed DFA is summarized as follows:

\[
\begin{align*}
\text{CLOSE}\{0\} &= \{0, 1, 3\} = S_0 \\
\delta(S_0, a) &= \text{CLOSE}\{1, 2\} = \{1, 2, 3\} = S_1 \\
\delta(S_0, b) &= \text{CLOSE}\{3\} = \{1, 3\} = S_2 \\
\delta(S_1, a) &= \text{CLOSE}\{} = \emptyset = S_3 \\
\delta(S_1, b) &= \text{CLOSE}\{3\} = \{1, 3\} = S_2 \\
\delta(S_2, a) &= \text{CLOSE}\{} = \emptyset = S_3 \\
\delta(S_2, b) &= \text{CLOSE}\{3\} = \{1, 3\} = S_2 \\
\delta(S_3, a) &= \text{CLOSE}\{} = \emptyset = S_3 \\
\delta(S_3, b) &= \text{CLOSE}\{} = \emptyset = S_3 
\end{align*}
\]
Louden, Exercise 2.14, page 92. Convert the NFA of Example 2.10 into a DFA using the subset construction. The resulting DFA is shown below on the right.
Second example
Convert an NFA to DFA

Convert this NFA into a DFA using the subset construction.
Convert an NFA to DFA

Each state of the DFA is a set of states of the NFA. The initial state of the DFA is the $\epsilon$-closure of the initial state of the NFA.

\[
\text{CLOSE}\{0\} = \{0, 2\} = S_0
\]

![Diagram of NFA to DFA conversion](image-url)
Convert an NFA to DFA

Determine the transition function of the DFA on all inputs. Begin with the initial state $S_0$, and determine the transition on input $a$.

\[
\text{CLOSE}\{0\} = \{0, 2\} = S_0
\]
\[
\delta(S_0, a) = \text{CLOSE}\{1\}
\]
\[
\delta(S_0, b) =
\]
Convert an NFA to DFA

The $\epsilon$-closure of the set $\{1\}$ is $\{1, 3\}$. This is a new state in the DFA, call it $S_1$.

\[
\text{CLOSE}\{0\} = \{0, 2\} = S_0 \\
\delta(S_0, a) = \text{CLOSE}\{1\} = \{1, 3\} = S_1 \\
\delta(S_0, b) =
\]
Convert an NFA to DFA

With the initial state $S_0$, determine the transition on input $b$.

$$\text{CLOSE}\{0\} = \{0, 2\} = S_0$$
$$\delta(S_0, a) = \text{CLOSE}\{1\} = \{1, 3\} = S_1$$
$$\delta(S_0, b) = \text{CLOSE}\{3\}$$
Convert an NFA to DFA

The $\epsilon$-closure of the set $\{3\}$ is $\{3\}$. This is a new state in the DFA, call it $S_2$.

\[
\text{CLOSE}\{0\} = \{0, 2\} = S_0
\]

\[
\delta(S_0, a) = \text{CLOSE}\{1\} = \{1, 3\} = S_1
\]

\[
\delta(S_0, b) = \text{CLOSE}\{3\} = \{3\} = S_2
\]
Determine the transition function of the DFA from state $S_1$ on inputs $a$ and $b$. On $a$ the DFA goes to the $\epsilon$-closure of state 1, which has already been computed and named state $S_1$. Since $S_1$ is also the starting state, we add a loop to the DFA.

\[
\begin{align*}
\delta(S_0, a) &= \text{CLOSE}\{1\} = \{1, 3\} = S_1 \\
\delta(S_0, b) &= \text{CLOSE}\{3\} = \{3\} = S_2 \\
\delta(S_1, a) &= \text{CLOSE}\{1\} = \{1, 3\} = S_1 \\
\delta(S_1, b) &= \\
\end{align*}
\]
With the state $S_1$, determine the transition on input $b$.

\[
\begin{align*}
\delta(S_0, a) &= \text{CLOSE}\{1\} = \{1, 3\} = S_1 \\
\delta(S_0, b) &= \text{CLOSE}\{3\} = \{3\} = S_2 \\
\delta(S_1, a) &= \text{CLOSE}\{1\} = \{1, 3\} = S_1 \\
\delta(S_1, b) &= \text{CLOSE}\{4\}
\end{align*}
\]
With the state $S_1$, determine the transition on input $b$.

\[
\delta(S_0, a) = \text{CLOSE}\{1\} = \{1, 3\} = S_1
\]
\[
\delta(S_0, b) = \text{CLOSE}\{3\} = \{3\} = S_2
\]
\[
\delta(S_1, a) = \text{CLOSE}\{1\} = \{1, 3\} = S_1
\]
\[
\delta(S_1, b) = \text{CLOSE}\{4\} = \{4\} = S_4
\]
With the state $S_2$, determine the transition on input $a$.

\[
\begin{align*}
\delta(S_0, a) &= \text{CLOSE}\{1\} = \{1, 3\} = S_1 \\
\delta(S_0, b) &= \text{CLOSE}\{3\} = \{3\} = S_2 \\
\delta(S_1, a) &= \text{CLOSE}\{1\} = \{1, 3\} = S_1 \\
\delta(S_1, b) &= \text{CLOSE}\{4\} = \{4\} = S_4 \\
\delta(S_2, a) &= \text{CLOSE}\{1\} = \{1, 3\} = S_1 \\
\delta(S_2, b) &= \\
\end{align*}
\]
With the state $S_2$, determine the transition on input $b$.

\[
\begin{align*}
\delta(S_0, a) &= \text{CLOSE}\{1\} = \{1, 3\} = S_1 \\
\delta(S_0, b) &= \text{CLOSE}\{3\} = \{3\} = S_2 \\
\delta(S_1, a) &= \text{CLOSE}\{1\} = \{1, 3\} = S_1 \\
\delta(S_1, b) &= \text{CLOSE}\{4\} = \{4\} = S_4 \\
\delta(S_2, a) &= \text{CLOSE}\{1\} = \{1, 3\} = S_1 \\
\delta(S_2, b) &= \text{CLOSE}\{4\} = \{4\} = S_4
\end{align*}
\]
With the state $S_4 = S_3$, determine the transition on inputs $a$ and $b$. 

\[ \delta(S_1, a) = \text{CLOSE}\{1\} = \{1, 3\} = S_1 \]
\[ \delta(S_1, b) = \text{CLOSE}\{4\} = \{4\} = S_4 \]
\[ \delta(S_2, a) = \text{CLOSE}\{1\} = \{1, 3\} = S_1 \]
\[ \delta(S_2, b) = \text{CLOSE}\{4\} = \{4\} = S_4 \]
\[ \delta(S_4, a) = \text{CLOSE}\{2\} = \{2\} = S_5 \]
\[ \delta(S_4, b) = \]

![Diagram](image_url)
With the state $S_4$, determine the transition on inputs $a$ and $b$.

\[
\begin{align*}
\delta(S_1, a) &= \text{CLOSE}\{1\} = \{1, 3\} = S_1 \\
\delta(S_1, b) &= \text{CLOSE}\{4\} = \{4\} = S_4 \\
\delta(S_2, a) &= \text{CLOSE}\{1\} = \{1, 3\} = S_1 \\
\delta(S_2, b) &= \text{CLOSE}\{4\} = \{4\} = S_4 \\
\delta(S_4, a) &= \text{CLOSE}\{2\} = \{2\} = S_5 \\
\delta(S_4, b) &= \text{CLOSE}\{\} = \emptyset = S_3
\end{align*}
\]
With the state $S_5$, determine the transition on inputs $a$ and $b$.

\[
\begin{align*}
\delta(S_2, a) &= \text{CLOSE}\{1\} = \{1, 3\} = S_1 \\
\delta(S_2, b) &= \text{CLOSE}\{4\} = \{4\} = S_4 \\
\delta(S_4, a) &= \text{CLOSE}\{2\} = \{2\} = S_5 \\
\delta(S_4, b) &= \text{CLOSE}\{\} = \emptyset = S_3 \\
\delta(S_5, a) &= \text{CLOSE}\{\} = \emptyset = S_3 \\
\delta(S_5, b) &= \emptyset = S_3
\end{align*}
\]
With the state $S_5$, determine the transition on inputs $a$ and $b$.

\[
\delta(S_2, a) = \text{CLOSE}\{1\} = \{1, 3\} = S_1 \\
\delta(S_2, b) = \text{CLOSE}\{4\} = \{4\} = S_4 \\
\delta(S_4, a) = \text{CLOSE}\{2\} = \{2\} = S_5 \\
\delta(S_4, b) = \text{CLOSE}\{\} = \emptyset = S_3 \\
\delta(S_5, a) = \text{CLOSE}\{\} = \emptyset = S_3 \\
\delta(S_5, b) = \text{CLOSE}\{3\} = \{3\} = S_2
\]
\[\delta(S_2, a) = \text{CLOSE}\{1\} = \{1, 3\} = S_1\]
\[\delta(S_2, b) = \text{CLOSE}\{4\} = \{4\} = S_4\]
\[\delta(S_4, a) = \text{CLOSE}\{2\} = \{2\} = S_5\]
\[\delta(S_4, b) = \text{CLOSE}\{\} = \emptyset = S_3\]
\[\delta(S_5, a) = \text{CLOSE}\{\} = \emptyset = S_3\]
\[\delta(S_5, b) = \text{CLOSE}\{3\} = \{3\} = S_2\]
Third example
[not completed]
Convert an NFA to DFA

Appel, 2nd, Exercise 2.5a, page 35. Convert the NFA into a DFA using the subset construction.
Convert an NFA to DFA

Each state of the DFA is a set of states of the NFA. The initial state of the DFA is the $\epsilon$-closure of the initial state of the NFA.

$$\text{CLOSE}\{1\} = \{1, 2, 3, 4\} = S_0$$
Chapter 2: Lexical Analysis

2.5 Lexical-Analyzer Generators
JLex
Jlex Input File

user code
%%% 
JLex directives
%%% 
rules

- User code: Java code, copied into the output file
- JLex directives: options, macros, state names
- rules: regular expressions for each token and associated Java action
Options for character and line counting.

// Character counting is turned off by default, but can be turned
// the "%char" directive. The zero-based character index of the
// character in the matched region of text is then placed in the
// variable "yychar".
%char

// Line counting is turned off by default, but can be turned on with
// the "%line" directive. The zero-based line index at the begin
// the matched region of text is then placed in the integer vari
%line

// Allow 8-bit characters.
%full
Jlex Directives

Internal code for the lexical analyzer class

\%

\{

\textit{code}

\%

\}

\textbf{class Yylex \{}

\ \ ...

\ \ \textit{code}

\}

\textbf{110}
Jlex Directives

Macro definitions

{name} = {definition}

name is a valid identifier, a definition is a regular expression that may contain other macro names (not recursive)
State declarations

%state state list

Example:

%state COMMENT, STRING
Jlex Rules

[states ] r-expression {action }

    {LF}        { newline(); }  
<YYINITIAL>  {WHITE}     {}  
<YYINITIAL>  "if"        { return tok (TokenConstants.IF, null); }  
<YYINITIAL>  "/*"        { commentLevel=1; yybegin(COMMENT); }  
<YYINITIAL>  {ID}        { return tok (TokenConstants.ID, yytext()); }  
<COMMENT>  "/*"        { commentLevel--; if (commentLevel==0) yybegin(YYINITIAL . }  
<COMMENT>  "."         {}  
<STRING>  "\\\"     { stringVal.append ('"'); }  
<STRING>  \{CR\}    { err ("Illegal CR (0x0D) in string"); }  
<STRING>  .      { err ("Illegal character in string"); }

No state list matches all state lists.
The rules given in a JLex specification should match all possible input. If the generated lexical analyzer receives input that does not match any of its rules, an error will be raised.

Therefore, all input should be matched by at least one rule. This can be guaranteed by placing the following rule at the end of the JLex rules:

```java
. { System.out.println("Unmatched input: " + yytext()); }
```

The dot will match any input except for the newline.
Using JavaCC for a scanner
Chapter 3: Parsing
Overview

• Context-Free Grammars. Productions, derivations, parse trees, ambiguity.

• Predictive Parsing. Recursive descent, LL

• LR Parsing. LR(0), SLR, LR(1), LALR(1)

• Parser Generators.
Chapter 3: Parsing

3.1. Context-Free Grammars
Formal language

“We say that a language is a set of strings; each string is a finite sequence of symbols taken from a finite alphabet.” Page 42.

In the context of programming languages, the strings are the source programs.

<table>
<thead>
<tr>
<th></th>
<th>scanner</th>
<th>parser</th>
</tr>
</thead>
<tbody>
<tr>
<td>alphabet</td>
<td>Latin-1</td>
<td>collection of tokens</td>
</tr>
<tr>
<td>symbols</td>
<td>characters</td>
<td>tokens</td>
</tr>
<tr>
<td>string</td>
<td>tokens</td>
<td>programs</td>
</tr>
</tbody>
</table>
Context-Free Grammar

A context-free grammar describes a language. A grammar has a set of productions of the form

\[ \text{symbol} \rightarrow \text{symbol symbol} \ldots \text{symbol} \]

where there are zero or more symbols on the RHS. Each symbol is either a terminal, a token, or a nonterminal, a symbol on the LHS of some production. No token can ever appear on the LHS.
Grammar 3.1


\[ S \rightarrow S ; S \]
\[ S \rightarrow \text{id} := E \]
\[ S \rightarrow \text{print} \ (L) \]
\[ E \rightarrow \text{id} \]
\[ E \rightarrow \text{num} \]
\[ E \rightarrow E + E \]
\[ E \rightarrow (S, E) \]
\[ L \rightarrow E \]
\[ L \rightarrow L , E \]
Definition of Derivation

To show that a sentence is in the language of a grammar, we construct a *derivation*: start with the start symbol, then repeatedly replace any nonterminal by one of its RHS. *Rightmost derivation* always chooses the rightmost nonterminal to expand. *Leftmost derivation* is analogous.
Example Derivation

We can derive the sentence

\[
\text{id := num ; id := num + id}
\]

using productions from Grammar 3.1. An overview of the derivation is given below.

\[
\begin{align*}
S \\
S ; S \\
S ; \text{id := } E \\
\text{id := } E ; \text{id := } E \\
\text{id := num ; id := } E \\
\text{id := num ; id := } E + E \\
\text{id := num ; id := } E + \text{id} \\
\text{id := num ; id := num + id}
\end{align*}
\]
Example Derivation

A derivation of the sentence \( \text{id} := \text{num} ; \text{id} := \text{num} + \text{id} \) using productions from Grammar 3.1. Begin with the distinguished start symbol of the grammar: \( S \). We choose to expand it first (we have no other choice).

<table>
<thead>
<tr>
<th>step</th>
<th>sentential form</th>
<th>production</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( S )</td>
<td></td>
</tr>
</tbody>
</table>
A derivation of the sentence \texttt{id := num ; id := num + id} using productions from Grammar 3.1. Select a production with the appropriate LHS nonterminal; we choose \(S \rightarrow S ; S\).

<table>
<thead>
<tr>
<th>step</th>
<th>sentential form</th>
<th>production</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(S)</td>
<td>(S \rightarrow S ; S)</td>
</tr>
</tbody>
</table>
A derivation of the sentence \texttt{id := num ; id := num + id} using productions from Grammar 3.1. Replace the LHS nonterminal with the RHS of the production.

<table>
<thead>
<tr>
<th>step</th>
<th>sentential form</th>
<th>production</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\texttt{S}</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>\texttt{S ; S}</td>
<td>\texttt{S \rightarrow S ; S}</td>
</tr>
</tbody>
</table>
A derivation of the sentence `id := num ; id := num + id` using productions from Grammar 3.1. Select a nonterminal to expand; we choose the rightmost $S$.

<table>
<thead>
<tr>
<th>step</th>
<th>sentential form</th>
<th>production</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$S$</td>
<td>$S \rightarrow S ; S$</td>
</tr>
<tr>
<td>1</td>
<td>$S ; S$</td>
<td></td>
</tr>
</tbody>
</table>
A derivation of the sentence $\text{id} := \text{num} ; \text{id} := \text{num} + \text{id}$ using productions from Grammar 3.1. Select a production with the appropriate LHS nonterminal.

<table>
<thead>
<tr>
<th>step</th>
<th>sentential form</th>
<th>production</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$S$</td>
<td>$S \rightarrow S ; S$</td>
</tr>
<tr>
<td>1</td>
<td>$S ; S$</td>
<td>$S \rightarrow \text{id} := E$</td>
</tr>
</tbody>
</table>
A derivation of the sentence `id := num ; id := num + id` using productions from Grammar 3.1. Replace the LHS nonterminal with the RHS of the production.

<table>
<thead>
<tr>
<th>step</th>
<th>sentential form</th>
<th>production</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>S</code></td>
<td><code>S → S ; S</code></td>
</tr>
<tr>
<td>1</td>
<td><code>S ; S</code></td>
<td><code>S → id := E</code></td>
</tr>
<tr>
<td>2</td>
<td><code>S ; id := E</code></td>
<td></td>
</tr>
</tbody>
</table>
A derivation of the sentence `id := num; id := num + id` using productions from Grammar 3.1. Select a nonterminal to expand.

<table>
<thead>
<tr>
<th>step</th>
<th>sentential form</th>
<th>production</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>S</code></td>
<td><code>S → S ; S</code></td>
</tr>
<tr>
<td>1</td>
<td><code>S ; S</code></td>
<td><code>S → id := E</code></td>
</tr>
<tr>
<td>2</td>
<td><code>S ; id := E</code></td>
<td></td>
</tr>
</tbody>
</table>
A derivation using productions from Grammar 3.1. Select a production with the appropriate LHS nonterminal.

<table>
<thead>
<tr>
<th>step</th>
<th>sentential form</th>
<th>production</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$S$</td>
<td>$S \rightarrow S ; S$</td>
</tr>
<tr>
<td>1</td>
<td>$S ; S$</td>
<td>$S \rightarrow \text{id} := E$</td>
</tr>
<tr>
<td>2</td>
<td>$S ; \text{id} := E$</td>
<td>$S \rightarrow \text{id} := E$</td>
</tr>
</tbody>
</table>
A derivation using productions from Grammar 3.1. Replace the LHS nonterminal with the RHS of the production.

<table>
<thead>
<tr>
<th>step</th>
<th>sentential form</th>
<th>production</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$S$</td>
<td>$S \rightarrow S ; ; ; S$</td>
</tr>
<tr>
<td>1</td>
<td>$S ; ; ; S$</td>
<td>$S \rightarrow \text{id} ; := ; E$</td>
</tr>
<tr>
<td>2</td>
<td>$S ; ; ; \text{id} ; := ; E$</td>
<td>$S \rightarrow \text{id} ; := ; E$</td>
</tr>
<tr>
<td>3</td>
<td>$\text{id} ; := ; E ; ; ; \text{id} ; := ; E$</td>
<td>$S \rightarrow \text{id} ; := ; E$</td>
</tr>
</tbody>
</table>
A derivation using productions from Grammar 3.1. Select a nonterminal to expand.

<table>
<thead>
<tr>
<th>step</th>
<th>sentential form</th>
<th>production</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$S$</td>
<td>$S \rightarrow S \ ; \ S$</td>
</tr>
<tr>
<td>1</td>
<td>$S \ ; \ S$</td>
<td>$S \rightarrow \text{id} := E$</td>
</tr>
<tr>
<td>2</td>
<td>$S \ ; \ \text{id} := E$</td>
<td>$S \rightarrow \text{id} := E$</td>
</tr>
<tr>
<td>3</td>
<td>$\text{id} := E \ ; \ \text{id} := E$</td>
<td>$S \rightarrow \text{id} := E$</td>
</tr>
</tbody>
</table>
A derivation using productions from Grammar 3.1. Select a production with the appropriate LHS nonterminal.

<table>
<thead>
<tr>
<th>step</th>
<th>sentential form</th>
<th>production</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$S$</td>
<td>$S \rightarrow S ; S$</td>
</tr>
<tr>
<td>1</td>
<td>$S ; S$</td>
<td>$S \rightarrow \text{id} := E$</td>
</tr>
<tr>
<td>2</td>
<td>$S ; \text{id} := E$</td>
<td>$S \rightarrow \text{id} := E$</td>
</tr>
<tr>
<td>3</td>
<td>\text{id} := $E$ ; \text{id} := $E$</td>
<td>$E \rightarrow \text{num}$</td>
</tr>
</tbody>
</table>
A derivation using productions from Grammar 3.1. Replace the LHS nonterminal with the RHS of the production.

<table>
<thead>
<tr>
<th>step</th>
<th>sentential form</th>
<th>production</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$S$</td>
<td>$S \rightarrow S ; S$</td>
</tr>
<tr>
<td>1</td>
<td>$S ; S$</td>
<td>$S \rightarrow \text{id} := E$</td>
</tr>
<tr>
<td>2</td>
<td>$S ; \text{id} := E$</td>
<td>$S \rightarrow \text{id} := E$</td>
</tr>
<tr>
<td>3</td>
<td>$\text{id} := E ; \text{id} := E$</td>
<td>$E \rightarrow \text{num}$</td>
</tr>
<tr>
<td>4</td>
<td>$\text{id} := \text{num} ; \text{id} := E$</td>
<td></td>
</tr>
</tbody>
</table>
A derivation using productions from Grammar 3.1. Select a nonterminal to expand. Select a production with the appropriate LHS nonterminal. Replace the LHS nonterminal with the RHS of the production.

<table>
<thead>
<tr>
<th>step</th>
<th>sentential form</th>
<th>production</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$S$</td>
<td>$S \to S ; S$</td>
</tr>
<tr>
<td>1</td>
<td>$S ; S$</td>
<td>$S \to \text{id} := E$</td>
</tr>
<tr>
<td>2</td>
<td>$S ; \text{id} := E$</td>
<td>$S \to \text{id} := E$</td>
</tr>
<tr>
<td>3</td>
<td>$\text{id} := E ; \text{id} := E$</td>
<td>$E \to \text{num}$</td>
</tr>
<tr>
<td>4</td>
<td>$\text{id} := \text{num} ; \text{id} := E$</td>
<td>$E \to E + E$</td>
</tr>
<tr>
<td>5</td>
<td>$\text{id} := \text{num} ; \text{id} := E + E$</td>
<td></td>
</tr>
</tbody>
</table>
A derivation using productions from Grammar 3.1. Select a nonterminal to expand. Select a production with the appropriate LHS nonterminal. Replace the LHS nonterminal with the RHS of the production.

<table>
<thead>
<tr>
<th>step</th>
<th>sentential form</th>
<th>production</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$S$</td>
<td>$S \rightarrow S ; S$</td>
</tr>
<tr>
<td>1</td>
<td>$S ; S$</td>
<td>$S \rightarrow \text{id} := E$</td>
</tr>
<tr>
<td>2</td>
<td>$S ; \text{id} := E$</td>
<td>$S \rightarrow \text{id} := E$</td>
</tr>
<tr>
<td>3</td>
<td>$\text{id} := E ; \text{id} := E$</td>
<td>$E \rightarrow \text{num}$</td>
</tr>
<tr>
<td>4</td>
<td>$\text{id} := \text{num} ; \text{id} := E$</td>
<td>$E \rightarrow E + E$</td>
</tr>
<tr>
<td>5</td>
<td>$\text{id} := \text{num} ; \text{id} := E + E$</td>
<td>$E \rightarrow \text{id}$</td>
</tr>
<tr>
<td>6</td>
<td>$\text{id} := \text{num} ; \text{id} := E + \text{id}$</td>
<td>$E \rightarrow \text{id}$</td>
</tr>
</tbody>
</table>
Select a nonterminal to expand. Select a production with the appropriate LHS nonterminal. Replace the LHS nonterminal with the RHS of the production. When the sentential form has no nonterminals a sentence has been derived.

<table>
<thead>
<tr>
<th>step</th>
<th>sentential form</th>
<th>production</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$S$</td>
<td>$S \rightarrow S ; S$</td>
</tr>
<tr>
<td>1</td>
<td>$S ; S$</td>
<td>$S \rightarrow \text{id} := E$</td>
</tr>
<tr>
<td>2</td>
<td>$S ; \text{id} := E$</td>
<td>$S \rightarrow \text{id} := E$</td>
</tr>
<tr>
<td>3</td>
<td>$\text{id} := E ; \text{id} := E$</td>
<td>$E \rightarrow \text{num}$</td>
</tr>
<tr>
<td>4</td>
<td>$\text{id} := \text{num} ; \text{id} := E$</td>
<td>$E \rightarrow E + E$</td>
</tr>
<tr>
<td>5</td>
<td>$\text{id} := \text{num} ; \text{id} := E + E$</td>
<td>$E \rightarrow \text{id}$</td>
</tr>
<tr>
<td>6</td>
<td>$\text{id} := \text{num} ; \text{id} := E + \text{id}$</td>
<td>$E \rightarrow \text{num}$</td>
</tr>
<tr>
<td>7</td>
<td>$\text{id} := \text{num} ; \text{id} := \text{num} + \text{id}$</td>
<td></td>
</tr>
</tbody>
</table>
Example Derivation

We have exhibited

\[ S \Rightarrow^* \text{id := num ; id := num + id} \]

A summary is given below.

\[
\begin{align*}
S & \quad S \rightarrow S ; S \\
S ; S & \quad S \rightarrow \text{id := E} \\
S ; \text{id := E} & \quad S \rightarrow \text{id := E} \\
\text{id := E} ; \text{id := E} & \quad \text{E \rightarrow num} \\
\text{id := num ; id := E} & \quad \text{E \rightarrow E + E} \\
\text{id := num ; id := E + E} & \quad \text{E \rightarrow id} \\
\text{id := num ; id := E + id} & \quad \text{E \rightarrow num} \\
\text{id := num ; id := num + id} & 
\end{align*}
\]
A formal definition of a parse tree for a context-free grammar is possible, but it is not illuminating. But the trees in the figures are suggestive. Each production used in the derivation of a string appears as a subtree in the diagram. The left-hand-side nonterminal appears as a node, and all the grammar symbols in the right-hand side of the production appear as children of this node.
Parse Trees

```
S ::= id ;
    | id := E
    | E num

S ::= id := E
    | E + E
    | E num
    | num
```
A derivation of the sentence \texttt{id := num ; id := num + id} using productions from Grammar 3.1. Begin with the distinguished start symbol of the grammar: \textit{S}.

\begin{center}
\begin{tabular}{c c c}
\hline
	\text{sentential form} & \text{production} & \text{parse tree} \\
\hline
	\texttt{S} & \texttt{S} & \texttt{S} \\
\end{tabular}
\end{center}
A derivation of the sentence $\text{id} := \text{num} ; \text{id} := \text{num} + \text{id}$ using productions from Grammar 3.1. Select a production with the appropriate LHS nonterminal.

<table>
<thead>
<tr>
<th>sentential form</th>
<th>production</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$S \rightarrow S ; S$</td>
</tr>
</tbody>
</table>

parse tree
A derivation of the sentence \texttt{id := num ; id := num + id} using productions from Grammar 3.1. Replace the LHS nonterminal with the RHS of the production.

<table>
<thead>
<tr>
<th>sentential form</th>
<th>production</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$S \rightarrow S ; S$</td>
</tr>
<tr>
<td>$S ; S$</td>
<td></td>
</tr>
</tbody>
</table>

Parse tree

```
S
  ;
  S
```

parse tree
A derivation of the sentence \texttt{id := num ; id := num + id} using productions from Grammar 3.1. Select a nonterminal to expand.

<table>
<thead>
<tr>
<th>sentential form</th>
<th>production</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$S \rightarrow S ; S$</td>
</tr>
<tr>
<td>$S ; S$</td>
<td>parse tree</td>
</tr>
</tbody>
</table>

\[ S \rightarrow S ; S \]
A derivation of the sentence \( \text{id} := \text{num} ; \text{id} := \text{num} + \text{id} \) using productions from Grammar 3.1. Select a production with the appropriate LHS nonterminal.

<table>
<thead>
<tr>
<th>sentential form</th>
<th>production</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>( S \rightarrow S ; S )</td>
</tr>
<tr>
<td>( S ; S )</td>
<td>( S \rightarrow \text{id} := E )</td>
</tr>
</tbody>
</table>

parse tree
A derivation of the sentence \texttt{id} := \texttt{num} ; \texttt{id} := \texttt{num} + \texttt{id} using productions from Grammar 3.1. Replace the LHS nonterminal with the RHS of the production.

<table>
<thead>
<tr>
<th>Sentential Form</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{S}</td>
<td>\texttt{S} \rightarrow \texttt{S} ; \texttt{S}</td>
</tr>
<tr>
<td>\texttt{S} ; \texttt{S}</td>
<td>\texttt{S} \rightarrow \texttt{id} := \texttt{E}</td>
</tr>
<tr>
<td>\texttt{S} ; \texttt{id} := \texttt{E}</td>
<td>\texttt{parse tree}</td>
</tr>
</tbody>
</table>

parse tree
<table>
<thead>
<tr>
<th>sentential form</th>
<th>production</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$S \rightarrow S ; S$</td>
</tr>
<tr>
<td>$S ; S$</td>
<td>$S \rightarrow \text{id} := E$</td>
</tr>
<tr>
<td>$S ; \text{id} := E$</td>
<td>$S \rightarrow \text{id} := E$</td>
</tr>
<tr>
<td>$\text{id} := E ; \text{id} := E$</td>
<td>parse tree</td>
</tr>
</tbody>
</table>

S → S ; S
S → id := E
S → id := E
S

parse tree
sentential form | production
---|---
$S$ | $S \rightarrow S ; S$
$S ; S$ | $S \rightarrow \text{id} := E$
$S ; \text{id} := E$ | $S \rightarrow \text{id} := E$
$\text{id} := E ; \text{id} := E$ | $E \rightarrow \text{num}$
$\text{id} := \text{num} ; \text{id} := E$ |
<table>
<thead>
<tr>
<th>Sentential Form</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$S \rightarrow S ; S$</td>
</tr>
<tr>
<td>$S ; S$</td>
<td>$S \rightarrow id := E$</td>
</tr>
<tr>
<td>$S ; id := E$</td>
<td>$E \rightarrow num$</td>
</tr>
<tr>
<td>$id := E ; id := E$</td>
<td>$E \rightarrow E + E$</td>
</tr>
<tr>
<td>$id := num ; id := E$</td>
<td>$E \rightarrow E + E$</td>
</tr>
<tr>
<td>$id := num ; id := E + E$</td>
<td>$E \rightarrow E + E$</td>
</tr>
</tbody>
</table>

Parse tree
sentential form               production

\[ S \]

\[ S ; S \]

\[ S ; \text{id} := E \]

\[ \text{id} := E ; \text{id} := E \]

\[ \text{id} := \text{num} ; \text{id} := E \]

\[ \text{id} := \text{num} ; \text{id} := E + E \]

\[ \text{id} := \text{num} ; \text{id} := E + \text{id} \]
When the sentential form has no nonterminals a sentence has been derived.

sentential form production
\[ S \]
\[ S ; S \]
\[ S ; \text{id} := E \]
\[ \text{id} := E ; \text{id} := E \]
\[ \text{id} := \text{num} ; \text{id} := E \]
\[ \text{id} := \text{num} ; \text{id} := E + E \]
\[ \text{id} := \text{num} ; \text{id} := E + \text{id} \]
\[ \text{id} := \text{num} ; \text{id} := \text{num} + \text{id} \]
Other Examples

Other derivable sentences ...

\[
\text{print ( num )}
\]

\[
\text{print ( id, num )}
\]

\[
id := \text{num} + \text{id} + \text{num}
\]

\[
\text{print ( id, num + id )}
\]

\[
\text{print ( print ( id ), num )}
\]
A grammar is *ambiguous* if it can derive a sentence with two different parse trees. We can usually eliminate ambiguity by transforming the grammar, though some formal languages are *inherently ambiguous*.

An *augmented grammar* is one that a special symbol $\$\$ to represent end-of-file.
Grammar 3.1 Is Ambiguous

One parse tree for

\[ id := \text{num} + \text{num} + \text{num} \]
Grammar 3.1 Is Ambiguous

Another parse tree for

\[
\text{id} := \text{num} + \text{num} + \text{num}
\]
Two Parse Trees

Two parse trees for

if $C_1$ then if $C_2$ then $S_1$ else $S_2$
Two Parse Trees

Two parse trees for

\[ \text{if } C_1 \text{ then if } C_2 \text{ then } S_1 \text{ else } S_2 \]
Two parse trees for

\begin{equation*}
    \text{if } C_1 \text{ then if } C_2 \text{ then } S_1 \text{ else } S_2
\end{equation*}
Two parse trees for

\[ \text{if } C_1 \text{ then if } C_2 \text{ then } S_1 \text{ else } S_2 \]
Two parse trees for

if \( C_1 \) then if \( C_2 \) then \( S_1 \) else \( S_2 \)
Equivalent Grammar

Just because a grammar is ambiguous does not mean that all grammars for the same language are ambiguous.

$$S \rightarrow \text{if } C \text{ then } S \mid \text{if } C \text{ then } S \text{ else } S \mid S'$$

This grammar for the same language is not.

$$S \rightarrow MS \mid UMS$$

$$MS \rightarrow \text{if } C \text{ then } MS \text{ else } MS \mid S'$$

$$UMS \rightarrow \text{if } C \text{ then } S \mid \text{if } C \text{ then } MS \text{ else } UMS$$
Equivalent Grammar

\[ S \]
\[ U M S \]

Now only one parse tree for

\[
\text{if } C_1 \text{ then if } C_2 \text{ then } S_1 \text{ else } S_2
\]
Equivalent Grammar

Now only one parse tree for

\[
\text{if } C_1 \text{ then if } C_2 \text{ then } S_1 \text{ else } S_2
\]
Equivalent Grammar

Now only one parse tree for

\[
\text{if } C_1 \text{ then if } C_2 \text{ then } S_1 \text{ else } S_2
\]
Equivalent Grammar

Now only one parse tree for

\[
\text{if } C_1 \text{ then if } C_2 \text{ then } S_1 \text{ else } S_2
\]
Alternative Language

If the “natural” grammar is ambiguous, could it mean that the construct is confusing to programmers? Is another approach possible?

Many languages have redesigned the syntax of the if statement:

\[
S \rightarrow \text{if } C \text{ then } S \text{ end if } \mid \\
\text{if } C \text{ then } S \text{ else } S \text{ end if } \mid \\
S'
\]

The two statements with different structure now have different syntax.

if \(C_1\) then if \(C_2\) then \(S_1\) end if else \(S_2\) end if

if \(C_1\) then if \(C_2\) then \(S_1\) else \(S_2\) end if end if
Grammar 3.8

A grammar for expressions (same languages as Grammar 3.5) separated into precedence levels. Appel, 2nd, Grammar 3.8, page 44.

\[
\begin{align*}
E & \rightarrow E + T \\
E & \rightarrow E - T \\
E & \rightarrow T \\
T & \rightarrow T \* F \\
T & \rightarrow T / F \\
T & \rightarrow F \\
F & \rightarrow \text{id} \\
F & \rightarrow \text{num} \\
F & \rightarrow (E)
\end{align*}
\]
Grammar 3.10

The Grammar 3.8 augmented with the production $S \rightarrow E \$$.  

\[ 
\begin{align*} 
S & \rightarrow E \$ \\
E & \rightarrow E + T \\
E & \rightarrow E - T \\
E & \rightarrow T \\
T & \rightarrow T * F \\
T & \rightarrow T / F \\
T & \rightarrow F \\
F & \rightarrow \text{id} \\
F & \rightarrow \text{num} \\
F & \rightarrow (E) 
\end{align*} \]
Chapter 3: Parsing

3.2. Predictive Parsing
Recursive Descent Parser

Grammars in the right form permit a parser of simple recursive functions. Each nonterminal turns into a recursive function forming a set of mutually recursive functions. Each function does a case analysis on the input to determine which production/RHS to follow. Terminals in the RHS are matched (consumed) and nonterminals turned into recursive calls.

Java Applet demo:

[link]
http://cswebsrv.cs.binghamton.edu/~zdu/parsdemo/recframe.html

[link]
http://www.uni-paderborn.de/fachbereich/AG/agkastens/compiler/pars
Grammar 3.11


\[
S \rightarrow \text{if } E \text{ then } S \text{ else } S \\
S \rightarrow \text{begin } S \text{ L} \\
S \rightarrow \text{print } E \\
L \rightarrow \text{end} \\
L \rightarrow ; S \text{ L} \\
E \rightarrow \text{num} = \text{num}
\]
We can build a recursive-descent, or predictive, parser for Appel, 2nd, Grammar 3-11, page 45 [page 165 of these notes].

/* Token definitions */
final int IF=1, THEN=2, ELSE=3, BEGIN=4,
            END=5, PRINT=6, SEMI=7, NUM=8, EQ=9;

int tok = getToken(); // Current token (non-local)

/* Convenience routine for matching a token */
private void match (int t) {
    if (tok==t) tok=getToken(); else error();
}

void S() {
    switch(tok) {
    case IF:    match(IF); E(); match(THEN); S(); match(ELSE); S(); break;
    case BEGIN: match(BEGIN); S(); L(); break;
    case PRINT: match(PRINT); E(); break;
    default:    error();
    }
}

void L() {
    switch(tok) {
    case END: match(END); break;
    case SEMI: match(SEMI); S(); L(); break;
    default: error();
    }
}

void E() { match(NUM); match(EQ); match(NUM); }
Grammar 3.10

The Grammar 3.8 augmented with the production $S \rightarrow E \$$. Appel, 2nd, Grammar 3.10, page 45.

$$S \rightarrow E \$
$$E \rightarrow E + T$$
$$E \rightarrow E - T$$
$$E \rightarrow T$$
$$T \rightarrow T \ast F$$
$$T \rightarrow T / F$$
$$T \rightarrow F$$
$$F \rightarrow \text{id}$$
$$F \rightarrow \text{num}$$
$$F \rightarrow (E)$$
We cannot build a predictive parser for Appel, 2nd, Grammar 3-10, page 45.

Consider the strings \((1*2-3)+4\) and \((1*2-3)\). In the first case, the initial call to \(E\) should use the production \(E \rightarrow E + T\), but in the second case it should use \(E \rightarrow T\).
Prediction

To predict, we need to know that a symbol uniquely determines the RHS of a production. So certainly it must be in the set of symbols than can begin a RHS. And, if the RHS is nullable then the symbol must be able to follow the LHS nonterminal in a sentential form. These considerations led us to the following definitions:

- $\text{nullable}[X]$: true iff $X$ can derive the empty string
- $\text{FIRST}[X]$: set of terminals that can begin strings derived from $X$
- $\text{FLW}[X]$: set of terminals that can follow $X$
Again, FIRST[$X$] is the set of terminals that can begin strings derived from $X$. Some authors put $\epsilon$ (the empty string) in FIRST[$X$], if $X$ is nullable. I don’t like this.
Grammar 3.12

A grammar used in illustrating computing FIRST and FLW. Apple, 2nd, Grammar 3.12, page 47.

\[
\begin{align*}
X \rightarrow Y \\
X \rightarrow a \\
Y \rightarrow \\
Y \rightarrow c \\
Z \rightarrow d \\
Z \rightarrow X Y Z
\end{align*}
\]

(The order is different than in the book. The order of the productions is important in the algorithms to compute nullable, first, and follow.)
Examples

For example, in Appel, Grammar 3.12, page 49:

- $Y$ is nullable,
- $\text{FIRST}[Z]$ contains $d$, and
- $\text{FLW}[X]$ contains $c$.

We now get more precise ....
**Nullable**

**Nullable.** Given a string $\gamma$ of terminal and nonterminal symbols, Nullable[$\gamma$] is true if $\gamma$ derives the empty string ($\gamma \Rightarrow^* \epsilon$).

In Grammar 3.12, for example, $XY \Rightarrow YY \Rightarrow Y \Rightarrow \epsilon$, so Nullable[$XY$] is true.

Obviously, no string with a terminal in it can be nullable.
Nullable

Algorithm to compute Nullable\([X]\) for nonterminals \(X\).

 Nullable\([X]\) := \textit{False} for all nonterminals \(X\)

repeat

for each production \(X \rightarrow Y_1 Y_2 \cdots Y_k\)

for each symbol \(Y_i\) where \(1 \leq i \leq k\)

if \(Y_i\) terminal or not Nullable\([Y_i]\) then

continue with next production

end if;

end;

end;

-- The RHS is all nullable, so is \(X\)

Nullable\([X]\) := \textit{True}

end;

until no changes;

In particular, note that if \(k = 0\), then \(X \rightarrow \epsilon\) and \(X\) is immediately nullable.
Example of Nullable

Consider Appel, 2nd, Grammar 3.12, page 47.

<table>
<thead>
<tr>
<th></th>
<th>step 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>no</td>
</tr>
<tr>
<td>Y</td>
<td>no</td>
</tr>
<tr>
<td>Z</td>
<td>no</td>
</tr>
</tbody>
</table>
Example of Nullable

Consider Appel, 2nd, Grammar 3.12, page 47.

<table>
<thead>
<tr>
<th></th>
<th>step 0</th>
<th>step 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Y</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Z</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

$Y \rightarrow \epsilon$ means $Y$ is nullable.
Example of Nullable

Consider Appel, 2nd, Grammar 3.12, page 47.

<table>
<thead>
<tr>
<th></th>
<th>step 0</th>
<th>step 1</th>
<th>step 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Y</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Z</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

$Y \rightarrow \epsilon$ means $Y$ is nullable. $X \rightarrow Y$ and $Y$ nullable means $X$ is nullable.
Example of Nullable

Consider Appel, 2nd, Grammar 3.12, page 47.

<table>
<thead>
<tr>
<th></th>
<th>step 0</th>
<th>step 1</th>
<th>step 2</th>
<th>nullable</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>$Y$</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>$Z$</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

$Y \rightarrow \epsilon$ means $Y$ is nullable. $X \rightarrow Y$ and $Y$ nullable means $X$ is nullable. No further changes.
Nullable

To compute if a string $\gamma$ of terminal and nonterminal symbols is nullable given $\text{Nullable}[X]$ for all nonterminals, we observed the following:

$\text{Nullable}[t]$ is false for all terminals $t$.

$\text{Nullable}[X_1X_2\cdots X_k]$ is true iff $\text{Nullable}[X_i]$ is true for all $1 \leq i \leq k$ (this is vacuously true in the case where there are no symbols in the string, $k = 0$).

So, we can compute nullable for any sentential form $\gamma$, given just a table of the nullable nonterminals.
**FIRST**

**FIRST.** Given a string $\gamma$ of terminal and nonterminal symbols, $t \in \text{FIRST}[\gamma]$ if $\gamma \Rightarrow^* t\omega$.

In Grammar 3.10, for example, $\text{FIRST}[T*F] = \{\text{id, num, (}\}$. (Some authors, not Appel, include $\epsilon$ in FIRST, if the terminal is nullable. Please do not do this.)
FIRST

Algorithm to compute FIRST[X] for nonterminals X.

FIRST[X] := ∅ for all nonterminals X
repeat
    for each production X → Y₁Y₂⋯Yₖ
        for each symbol Yᵢ where 1 ≤ i ≤ k
            FIRST[X] := FIRST[X] ∪ FIRST[Yᵢ]
            if Yᵢ terminal or not Nullable[Yᵢ] then
                continue with next production
            end if
        end for
    end for
until no changes
Example of FIRST

Consider Appel, Grammar 3.12, page 49.

<table>
<thead>
<tr>
<th></th>
<th>step 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$Y$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$Z$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
Example of FIRST

Consider Appel, Grammar 3.12, page 49.

<table>
<thead>
<tr>
<th></th>
<th>step 0</th>
<th>step 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>$\emptyset$</td>
<td>$a$</td>
</tr>
<tr>
<td>$Y$</td>
<td>$\emptyset$</td>
<td>$c$</td>
</tr>
<tr>
<td>$Z$</td>
<td>$\emptyset$</td>
<td>$a, c, d$</td>
</tr>
</tbody>
</table>
Example of FIRST

Consider Appel, Grammar 3.12, page 49.

<table>
<thead>
<tr>
<th></th>
<th>step 0</th>
<th>step 1</th>
<th>step 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>$\emptyset$</td>
<td>$a$</td>
<td>$a, c$</td>
</tr>
<tr>
<td>$Y$</td>
<td>$\emptyset$</td>
<td>$c$</td>
<td>$c$</td>
</tr>
<tr>
<td>$Z$</td>
<td>$\emptyset$</td>
<td>$a, c, d$</td>
<td>$a, c, d$</td>
</tr>
</tbody>
</table>
Example of FIRST

Consider Appel, Grammar 3.12, page 49.

<table>
<thead>
<tr>
<th></th>
<th>step 0</th>
<th>step 1</th>
<th>step 2</th>
<th>FIRST</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>$\emptyset$</td>
<td>a</td>
<td>$a, c$</td>
<td>$a, c$</td>
</tr>
<tr>
<td>$Y$</td>
<td>$\emptyset$</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>$Z$</td>
<td>$\emptyset$</td>
<td>$a, c, d$</td>
<td>$a, c, d$</td>
<td>$a, c, d$</td>
</tr>
</tbody>
</table>

180-c
FIRST of Strings

To compute FIRST[γ] for some string γ of terminal and nonterminal symbols, we add the following definitions.

FIRST[ε] = ∅. For all terminals t, FIRST[t] is \{t\}.

If \(X_1\) is a terminal or not nullable, then

\[
\text{FIRST}[X_1X_2 \cdots X_k] = \text{FIRST}[X_1]
\]

else

\[
\text{FIRST}[X_1X_2 \cdots X_k] = \text{FIRST}[X_1] \cup \text{FIRST}[X_2 \cdots X_k]
\]
Follow

**Follow.** Given a nonterminal $X$, a terminal $t$ is in $\text{FLW}[X]$, if $N \Rightarrow^* \alpha X t \beta$ for some $\alpha$, $\beta$, and $N$. In other words, $t$ follows $X$ in some sentential form.

In Grammar 3.10, for example,

$$\text{FLW}[E] = \{+, -, *, /, \), $\}$$

[Is this right?]
Follow

Algorithm to compute $\text{FLW}[X]$ for nonterminals of a given grammar.

$\text{FLW}[X] := \emptyset$ for all nonterminals $X$

repeat
    for each production $X \rightarrow Y_1 Y_2 \cdots Y_k$
        for each symbol $Y_i$ where $1 \leq i \leq k$
            if $Y_i$ is a nonterminal then
                $\text{FLW}[Y_i] := \text{FLW}[Y_i] \cup \text{FIRST}[Y_{i+1} \cdots Y_k]$
                if $\text{Nullable}[Y_{i+1} \cdots Y_k]$ then
                    $\text{FLW}[Y_i] := \text{FLW}[Y_i] \cup \text{FLW}[X]$
                end if
            end if
        end for
    end for
until no changes
Note in particular that when \( i = k \), we consider 
\( \text{Nullable}[Y_{i+1} \cdots Y_k] \) to be true, and

\[
\text{FLW}[Y_k] := \text{FLW}[Y_k] \cup \text{FLW}[X]
\]

(if \( Y_k \) is a nonterminal). In other words, for every production 
\( X \rightarrow \cdots Y_k \) ending in a nonterminal \( Y_k \), we add \( \text{FLW}[X] \) to the 
\( \text{FLW}[Y_k] \).

<table>
<thead>
<tr>
<th></th>
<th>step 0</th>
<th>step 1</th>
<th>FLW</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>$\emptyset$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$Y$</td>
<td>$\emptyset$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$Z$</td>
<td>$\emptyset$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

185

<table>
<thead>
<tr>
<th></th>
<th>step 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$Y$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$Z$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Because $\text{FIRST}[YZ]$ is equal to $\text{FIRST}[Y] \cup \text{FIRST}[Z] = \{c\} \cup \{a, c, d\} = \{a, c, d\}$. 

<table>
<thead>
<tr>
<th></th>
<th>step 0</th>
<th>step 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>$\emptyset$</td>
<td>$a, c, d$</td>
</tr>
<tr>
<td>$Y$</td>
<td>$\emptyset$</td>
<td>$a, c, d$</td>
</tr>
<tr>
<td>$Z$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Because $\text{FIRST}[YZ]$ is equal to
$\text{FIRST}[Y] \cup \text{FIRST}[Z] = \{c\} \cup \{a, c, d\} = \{a, c, d\}$. 

<table>
<thead>
<tr>
<th></th>
<th>step 0</th>
<th>step 1</th>
<th>FLW</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>$\emptyset$</td>
<td>$a, c, d$</td>
<td>$a, c, d$</td>
</tr>
<tr>
<td>$Y$</td>
<td>$\emptyset$</td>
<td>$a, c, d$</td>
<td>$a, c, d$</td>
</tr>
<tr>
<td>$Z$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Because $\text{FIRST}[YZ]$ is equal to
$\text{FIRST}[Y] \cup \text{FIRST}[Z] = \{c\} \cup \{a, c, d\} = \{a, c, d\}$. 
### Nullable, First, Follow, Grammar 3.12

<table>
<thead>
<tr>
<th></th>
<th>nullable</th>
<th>FIRST</th>
<th>FLW</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>yes</td>
<td>$a,c$</td>
<td>$a,c,d$</td>
</tr>
<tr>
<td>$Y$</td>
<td>yes</td>
<td>$c$</td>
<td>$a,c,d$</td>
</tr>
<tr>
<td>$Z$</td>
<td>no</td>
<td>$a,c,d$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
Predictive Parsing

The production $A \rightarrow \alpha_k$ is predicted when the look-ahead token is in the set $\text{FIRST}[\alpha_k]$. If $\text{Nullable}[\alpha_k]$, then the production is also predicted when the look-ahead token is in the set $\text{FLW}[A]$.

Predictive parsing is like a DFA with recursive calls. Each production is its own DFA in which an arc with a nonterminal on it requires a recursive call. If we simulate the recursive stack, we get a state or table-driven parser.
Predictive Parser

Construction of a predictive parsing table

\[ M : \text{nonterminals} \times \text{terminals} \]

for an augmented grammar. For each production \( A \rightarrow \alpha \) do the following two steps.

1. For each terminal \( a \) in \( \text{FIRST}[\alpha] \), add \( A \rightarrow \alpha \) to \( M[A, a] \).

2. If \( \text{Nullable}[\alpha] \), add \( A \rightarrow \alpha \) to \( M[A, b] \) for each terminal \( b \) of \( \text{FLW}[A] \) (including \( b = \$ \)).
Observe that we have *two* conflicting entries in the entry for nonterminal $X$ and input terminal $a$ (as well as for $Y$, $c$, and $Z$, $d$).
LL Parsing Engine

See Aho, Sethi, Ullman, Figure 4.13, page 186. See Grune et al, Figure 2.67, page 135.

A particular kind of pushdown automaton.

The machine examines the input (with $ at the end) token by token. If the machine matches all the input and empties the stack without finding any syntax errors, then the machine accepts the input.

This machine assumes an augmented grammar which makes it slightly easier.
LL Parsing Engine

input

\[ a + b \]

stack

$X$

$Y$

$Z$

$\$

LL parsing engine

parsing table M

left-most derivation

192
Push \((S_0)\); -- push start symbol into empty stack

t := GetNextToken(); -- first token in input

while not StackEmpty loop
    X := pop symbol off top of stack
    if X a nonterminal
        if \(M[X, t] = X \Rightarrow Y_1 \ldots Y_n\) then
            Push \((Y_n)\); \ldots \; \text{Push} \;(Y_1);
        else
            Syntax Error -- no entry in table
        end if;
    else
        if \((X = \$)\) and \((t = \$)\) then
            Accept
        elsif \((X = t)\) then
            t := GetNextToken();
        else
            Syntax Error -- input not expected
        end if;
    end if;
end loop;
Predictive Parsing

Compute the LL parsing table.

1. Use the augmented grammar.

2. Compute Nullable, FIRST, FLW

3. Compute FIRST of every RHS and FLW as appropriate

4. Build the table according to the rules

5. Check for multiple entries
Grammar 3.15

An example grammar for use in LL parsing; Appel, 2nd, Grammar 3.15, page 52.

\[
S \rightarrow E \$
\]

\[
E \rightarrow TE'
\]

\[
E' \rightarrow +TE'
\]

\[
E' \rightarrow \epsilon
\]

\[
T \rightarrow FT'
\]

\[
T' \rightarrow *FT'
\]

\[
T' \rightarrow \epsilon
\]

\[
F \rightarrow \text{id}
\]

\[
F \rightarrow (E)
\]
Let us build the LL parsing table for Appel, 2nd, Grammar 3.15, page 52.

Computing nullable is quite easy.

Computing FIRST for the nonterminals:

\[
\begin{align*}
\text{FIRST}[F] &= \{ (, \text{id}) \} \\
\text{FIRST}[S] &= \text{FIRST}[E] = \text{FIRST}[T] = \text{FIRST}[F] \\
\text{FIRST}[E'] &= \{ + \} \\
\text{FIRST}[T'] &= \{ \ast \}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Rule</th>
<th>Action</th>
<th>FLW Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow E$</td>
<td>$$ \in \text{FLW}[E]$</td>
<td></td>
</tr>
<tr>
<td>$E \rightarrow TE'$</td>
<td>add $\text{FIRST}[E']$ to $\text{FLW}[T]$</td>
<td>add $\text{FLW}[E]$ to $\text{FLW}[T]$</td>
</tr>
<tr>
<td>$E' \rightarrow +TE'$</td>
<td>(superfluous)</td>
<td></td>
</tr>
<tr>
<td>$E' \rightarrow \epsilon$</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>$T \rightarrow FT'$</td>
<td>add $\text{FIRST}[T']$ to $\text{FLW}[F]$</td>
<td>add $\text{FLW}[T]$ to $\text{FLW}[F]$</td>
</tr>
<tr>
<td>$T' \rightarrow ^*FT'$</td>
<td>(superfluous)</td>
<td></td>
</tr>
<tr>
<td>$T' \rightarrow \epsilon$</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>$F \rightarrow \text{id}$</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>$F \rightarrow (E)$</td>
<td>$) \in \text{FLW}[E]$</td>
<td></td>
</tr>
</tbody>
</table>
Computing FOLLOW
(continued)

), $ \in FLW[E]
FLW[E'] = FLW[E] = \{), $\}
FLW[T] = FLW[E'] \cup \text{FIRST}[E'] = \{+, ), $\}
FLW[T'] = FLW[T] = \{+, ), $\}
FLW[F] = FLW[T'] \cup \text{FIRST}[T'] = \{*, +, ), $\}
**Nullable, FIRST, FOLLOW**

Summary of nullable, first, and follow for Grammar 3.15, page 54.

<table>
<thead>
<tr>
<th></th>
<th>Nullable</th>
<th>FIRST</th>
<th>FLW</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>no</td>
<td>${ (, \text{id} ) }$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$E$</td>
<td>no</td>
<td>${ (, \text{id} ) }$</td>
<td>$(), $}$</td>
</tr>
<tr>
<td>$E'$</td>
<td>yes</td>
<td>${ + }$</td>
<td>$(), $}$</td>
</tr>
<tr>
<td>$T$</td>
<td>no</td>
<td>${ (, \text{id} ) }$</td>
<td>$(+, ), $}$</td>
</tr>
<tr>
<td>$T'$</td>
<td>yes</td>
<td>${ * }$</td>
<td>$(+, ), $}$</td>
</tr>
<tr>
<td>$F$</td>
<td>no</td>
<td>${ (, \text{id} ) }$</td>
<td>$(*, +, ), $}$</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c|c|c|c}
A \rightarrow \alpha & \text{FIRST}[\alpha] & \text{n}[\alpha] & \text{FLW}[A] \\
\hline
S \rightarrow E$ & \{(, \text{id}\} & \text{no} & \{\right, \}$ \\
E \rightarrow TE' & \{(, \text{id}\} & \text{no} & \{\right, \}$ \\
E' \rightarrow +TE' & \{+\} & \text{no} & \{\right, \}$ \\
E' \rightarrow \epsilon & \emptyset & \text{yes} & \{\right, \}$ \\
T \rightarrow FT' & \{(, \text{id}\} & \text{no} & \{\right, \}, \$\} \\
T' \rightarrow FT' & \{(, \text{id}\} & \text{no} & \{\right, \}, \$\} \\
T' \rightarrow *FT' & \{*\} & \text{no} & \{\right, \}$ \\
T' \rightarrow \epsilon & \emptyset & \text{yes} & \{+ , \}$ \\
F \rightarrow \text{id} & \{\text{id}\} & \text{no} & \{\right, \}$ \\
F \rightarrow (E) & \{\right\} & \text{no} & \{\right, \}$ \\
\end{array}
\]
Add to …

<table>
<thead>
<tr>
<th>Production</th>
<th>Push Stack 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow E$$</td>
<td>$M[S, (], M[S, \text{id}]$</td>
</tr>
<tr>
<td>$E \rightarrow TE'$</td>
<td>$M[E, (], M[E, \text{id}]$</td>
</tr>
<tr>
<td>$E' \rightarrow +TE'$</td>
<td>$M[E', +]$</td>
</tr>
<tr>
<td>$E' \rightarrow \epsilon$</td>
<td>$M[E', )], M[E', $$</td>
</tr>
<tr>
<td>$T \rightarrow FT'$</td>
<td>$M[T, (], M[T, \text{id}]$</td>
</tr>
<tr>
<td>$T' \rightarrow *FT'$</td>
<td>$M[T', *]$</td>
</tr>
<tr>
<td>$T' \rightarrow \epsilon$</td>
<td>$M[T', +], M[T', )], M[T', $$</td>
</tr>
<tr>
<td>$F \rightarrow \text{id}$</td>
<td>$M[F, \text{id}]$</td>
</tr>
<tr>
<td>$F \rightarrow (E)$</td>
<td>$M[F, (]$</td>
</tr>
</tbody>
</table>
## Predictive Parsing Table for Grammar 3.15

Predictive Parsing Table from Appel, 2nd, Table 3.17, page 53, for Grammar 3.15, page 52 [page 195 of these notes].

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>id</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$S \rightarrow E$</td>
<td>$S \rightarrow E$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E$</td>
<td>$E \rightarrow TE'$</td>
<td>$E \rightarrow TE'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E'$</td>
<td>$E' \rightarrow +TE'$</td>
<td>$E' \rightarrow \epsilon$</td>
<td>$E' \rightarrow \epsilon$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T$</td>
<td>$T \rightarrow FT'$</td>
<td>$T \rightarrow FT'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T'$</td>
<td>$T' \rightarrow \epsilon$</td>
<td>$T' \rightarrow *FT'$</td>
<td>$T' \rightarrow \epsilon$</td>
<td>$T' \rightarrow \epsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$F$</td>
<td>$F \rightarrow (E)$</td>
<td>$F \rightarrow \text{id}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example of LL parsing $id + id^*id$ using Appel, 2nd, Grammar 3.15, page 52 [page 195 of these notes]. Begin by pushing start symbol into the stack.

<table>
<thead>
<tr>
<th>step</th>
<th>← stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$S$</td>
<td>$id_1 + id_2^*id_3$</td>
<td></td>
</tr>
</tbody>
</table>
Example of LL parsing $\text{id} + \text{id}^*\text{id} \, \$ using Appel, 2nd, Grammar 3.15, page 52 [page 195 of these notes]. Look up action in parsing table.

<table>
<thead>
<tr>
<th>step</th>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$S$</td>
<td>$\text{id}_1 + \text{id}_2^*\text{id}_3 , $</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$S \to E$</td>
</tr>
</tbody>
</table>
Predictive Parsing

Predictive Parsing table for Appel, 2nd, Grammar 3.15, page 52 [page 195].

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>id</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>S → E$</td>
<td>S → E$</td>
<td>E → TE'</td>
<td>E → TE'</td>
<td>E' → +TE'</td>
<td>E' → ε</td>
</tr>
<tr>
<td>E</td>
<td>E → TE'</td>
<td>E → TE'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E'</td>
<td>E' → +TE'</td>
<td>E' → ε</td>
<td>E' → ε</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T → FT'</td>
<td>T → FT'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T'</td>
<td>T' → ε</td>
<td>T' → *FT'</td>
<td>T' → ε</td>
<td>T' → ε</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F → (E)</td>
<td>F → id</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

205
Example of LL parsing \( \text{id} + \text{id} \ast \text{id} \) \$ using grammar 3.15, page 54. Pop stack, push RHS.

<table>
<thead>
<tr>
<th>step</th>
<th>( \leftrightarrow ) stack</th>
<th>input ( \text{id}_1 + \text{id}_2 \ast \text{id}_3 $</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( S )</td>
<td>( \text{id}_1 + \text{id}_2 \ast \text{id}_3 $</td>
<td>( S \rightarrow E $</td>
</tr>
<tr>
<td>1</td>
<td>( E $ )</td>
<td>( \text{id}_1 + \text{id}_2 \ast \text{id}_3 $</td>
<td></td>
</tr>
</tbody>
</table>
Example of LL parsing \( \text{id} + \text{id}*\text{id} \) using grammar 3.15, page 54. Look up action.

<table>
<thead>
<tr>
<th>step</th>
<th>← stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( S )</td>
<td>( \text{id}_1 + \text{id}_2*\text{id}_3 ) $</td>
<td>( S \rightarrow E)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>( E ) $</td>
<td>( \text{id}_1 + \text{id}_2*\text{id}_3 ) $</td>
<td>( E \rightarrow T E' )</td>
</tr>
</tbody>
</table>
Predictive Parsing

Predictive Parsing table for Appel, 2nd, Grammar 3.15, page 52 [page 195].

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>*</th>
<th>(</th>
<th></th>
<th>id</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$S \rightarrow E$$</td>
<td>$S \rightarrow E$$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E$</td>
<td>$E \rightarrow TE'$</td>
<td>$E \rightarrow TE'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E'$</td>
<td>$E' \rightarrow +TE'$</td>
<td>$E' \rightarrow \epsilon$</td>
<td>$E' \rightarrow \epsilon$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T$</td>
<td>$T \rightarrow FT'$</td>
<td>$T \rightarrow FT'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T'$</td>
<td>$T' \rightarrow \epsilon$</td>
<td>$T' \rightarrow *FT'$</td>
<td>$T' \rightarrow \epsilon$</td>
<td>$T' \rightarrow \epsilon$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$F$</td>
<td>$F \rightarrow (E)$</td>
<td>$F \rightarrow id$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example of LL parsing $\text{id} + \text{id}^*\text{id}$ using grammar 3.15, page 54. Pop LHS, push RHS.

<table>
<thead>
<tr>
<th>step</th>
<th>(\leftarrow) stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(S) (\text{id}_1 + \text{id}_2^*\text{id}_3) ($)</td>
<td>(\text{id}_1 + \text{id}_2^*\text{id}_3) ($)</td>
<td>(S \rightarrow E)</td>
</tr>
<tr>
<td>1</td>
<td>(E) ($) (\text{id}_1 + \text{id}_2^*\text{id}_3) ($)</td>
<td>(\text{id}_1 + \text{id}_2^*\text{id}_3) ($)</td>
<td>(E \rightarrow TE')</td>
</tr>
<tr>
<td>2</td>
<td>(TE') ($) (\text{id}_1 + \text{id}_2^*\text{id}_3) ($)</td>
<td>(\text{id}_1 + \text{id}_2^*\text{id}_3) ($)</td>
<td></td>
</tr>
</tbody>
</table>
Example of LL parsing $id + id*id$ using grammar 3.15, page 54. Look up action.

<table>
<thead>
<tr>
<th>step</th>
<th>&lt;- stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$S$</td>
<td>$id_1 + id_2*id_3$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$S \rightarrow E$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$E$</td>
<td>$id_1 + id_2*id_3$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$E \rightarrow TE'$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$TE'$</td>
<td>$id_1 + id_2*id_3$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$T \rightarrow FT'$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example of LL parsing $id + id*id$ using grammar 3.15, page 54. Pop LHS, push RHS.

<table>
<thead>
<tr>
<th>step</th>
<th>← stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$S$</td>
<td>$id_1 + id_2*id_3$</td>
<td>$S \rightarrow E$ $</td>
</tr>
<tr>
<td></td>
<td>$E$ $</td>
<td>$</td>
<td>$id_1 + id_2*id_3$</td>
</tr>
<tr>
<td>1</td>
<td>$T E'$ $</td>
<td>$</td>
<td>$id_1 + id_2*id_3$</td>
</tr>
<tr>
<td>2</td>
<td>$FT' E'$ $</td>
<td>$</td>
<td>$id_1 + id_2*id_3$</td>
</tr>
<tr>
<td>3</td>
<td>$FT' E'$ $</td>
<td>$</td>
<td>$id_1 + id_2*id_3$</td>
</tr>
</tbody>
</table>
Example of LL parsing \( \text{id} + \text{id*id} \) $ using grammar 3.15, Appel, 2nd, page 52. Lookup action.

<table>
<thead>
<tr>
<th>step</th>
<th>← stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( S )</td>
<td>( \text{id}_1 + \text{id}_2*\text{id}_3 ) $</td>
<td>( S \rightarrow E $ )</td>
</tr>
<tr>
<td>1</td>
<td>( E $</td>
<td>( \text{id}_1 + \text{id}_2*\text{id}_3 ) $</td>
<td>( E \rightarrow T E' )</td>
</tr>
<tr>
<td>2</td>
<td>( T E' $</td>
<td>( \text{id}_1 + \text{id}_2*\text{id}_3 ) $</td>
<td>( T \rightarrow F T' )</td>
</tr>
<tr>
<td>3</td>
<td>( F T' E' $</td>
<td>( \text{id}_1 + \text{id}_2*\text{id}_3 ) $</td>
<td>( F \rightarrow \text{id} )</td>
</tr>
</tbody>
</table>
Example of LL parsing $id + id*id$ using grammar 3.15, Appel, 2nd, page 52. Continuing ...

<table>
<thead>
<tr>
<th>step</th>
<th>$\rightarrow$ stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
</table>
| 0    | $S$                | id$_1$ + id$_2*id$_3$ | $S \rightarrow E\$
<p>|      | $S \rightarrow E$ |             |                 |
| 1    | $E $              | id$_1$ + id$_2<em>id$_3$ | $E \rightarrow T E'$ |
|      | $E \rightarrow T E'$ |             |                 |
| 2    | $T E' $           | id$_1$ + id$_2</em>id$_3$ | $T \rightarrow F T'$ |
|      | $T \rightarrow F T'$ |             |                 |
| 3    | $F T' E' $       | id$_1$ + id$_2<em>id$_3$ | $F \rightarrow id$  |
|      | $F \rightarrow id$  |             |                 |
| 4    | $id T' E' $      | id$_1$ + id$_2</em>id$_3$ | match $id$     |
|      | $T' \rightarrow \epsilon$ |             |                 |
| 5    | $T' E' $         | +id$_2*id$_3$ | $T' \rightarrow \epsilon$ |
|      | $T' \rightarrow \epsilon$ |             |                 |</p>
<table>
<thead>
<tr>
<th>step</th>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T' \ E'$</td>
<td>$+id_2*id_3$</td>
<td></td>
<td>$T' \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$E'$</td>
<td>$+id_2*id_3$</td>
<td></td>
<td>$E' \rightarrow +T E'$</td>
</tr>
<tr>
<td>$+T \ E'$</td>
<td>$+id_2*id_3$</td>
<td></td>
<td>match $+$</td>
</tr>
<tr>
<td>$T \ E'$</td>
<td>$id_2*id_3$</td>
<td></td>
<td>$T \rightarrow F T'$</td>
</tr>
<tr>
<td>$F \ T' \ E'$</td>
<td>$id_2*id_3$</td>
<td></td>
<td>$F \rightarrow id$</td>
</tr>
<tr>
<td>$id \ T' \ E'$</td>
<td>$id_2*id_3$</td>
<td></td>
<td>match $id$</td>
</tr>
<tr>
<td>$T' \ E'$</td>
<td>*$id_3$</td>
<td></td>
<td>$T' \rightarrow *F T'$</td>
</tr>
<tr>
<td>$E'$</td>
<td>$+id_2*id_3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>step</td>
<td>stack</td>
<td>input</td>
<td>action</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>---------</td>
<td>------------------</td>
</tr>
<tr>
<td></td>
<td>$T'\ E'\ $</td>
<td>$^{*}\text{id}_3\ $</td>
<td>$T' \rightarrow^{*} FT'$</td>
</tr>
<tr>
<td></td>
<td>$^{*} FT'\ E'\ $</td>
<td>$^{*}\text{id}_3\ $</td>
<td>$F \rightarrow \text{id}$</td>
</tr>
<tr>
<td></td>
<td>$FT'\ E'\ $</td>
<td>$\text{id}_3\ $</td>
<td>$F \rightarrow \text{id}$</td>
</tr>
<tr>
<td></td>
<td>$\text{id}\ T'\ E'\ $</td>
<td>$\text{id}_3\ $</td>
<td>$E \rightarrow \epsilon$</td>
</tr>
<tr>
<td></td>
<td>$T'\ E'\ $</td>
<td>$\ $</td>
<td>$T' \rightarrow \epsilon$</td>
</tr>
<tr>
<td></td>
<td>$E'\ $</td>
<td>$\ $</td>
<td>$E' \rightarrow \epsilon$</td>
</tr>
<tr>
<td></td>
<td>$\ $</td>
<td>$\ $</td>
<td>$\text{match}\ $$</td>
</tr>
</tbody>
</table>
Summary of the left-most derivation discovered by the parsing.

\[
S \Rightarrow E$
\[
\Rightarrow TE'$
\[
\Rightarrow FT'E'
\[
\Rightarrow id_1T'E'$
\[
\Rightarrow id_1E'$
\[
\Rightarrow id_1+TE'$
\[
\Rightarrow id_1+FT'E'$
\[
\Rightarrow id_1+id_2T'E'$
\[
\Rightarrow id_1+id_2*FT'E'$
\[
\Rightarrow id_1+id_2*id_3T'E'$
\[
\Rightarrow id_1+id_2*id_3E'$
\[
\Rightarrow id_1+id_2*id_3$
\]
The grammar $G$ is LL($k$)—left-to-right, leftmost-derivation, $k$-symbol lookahead—if the three conditions

1. $S \Rightarrow_{lm}^* wA\alpha \Rightarrow_{lm} w\beta\alpha \Rightarrow^* wx$

2. $S \Rightarrow_{lm}^* wA\alpha \Rightarrow_{lm} w\gamma\alpha \Rightarrow^* wy$

3. $FIRST_k(x) = FIRST_k(y)$

imply that $\beta = \gamma$. 
Eliminating left Recursion. \( E \rightarrow E \ldots \) will certainly cause an LL first/first conflict; the first set of \( E \) will contain anything the non-recursive alternatives have.

\[
E \rightarrow E + T \\
E \rightarrow T
\]

Left Factoring. Also an LL first/first conflict.

\[
S \rightarrow \text{if } E \text{ then } S \text{ else } S \\
S \rightarrow \text{if } E \text{ then } S
\]

\[
S \rightarrow \text{if } E \text{ then } S X \\
X \rightarrow \\
X \rightarrow \text{else } S
\]
Left Recursion

1. $X \rightarrow X \gamma_1$
2. $X \rightarrow X \gamma_2$
3. $X \rightarrow \alpha_1$
4. $X \rightarrow \alpha_2$

1. $X \rightarrow \alpha_1 X'$
2. $X \rightarrow \alpha_2 X'$
3. $X' \rightarrow \epsilon$
4. $X' \rightarrow \gamma_1 X'$
5. $X' \rightarrow \gamma_2 X'$
Chapter 3: Parsing

3.3. LR Parsing
LR parsing was invented by D. Knuth, 1965. LR\((k)\)—left-to-right, rightmost-derivation, \(k\)-symbol lookahead

1. LR(0) Parsing
2. SLR Parsing
3. LR(1) Parsing
4. LALR(1) Parsing
LR Parsing

Consider the following grammar.

\[ S \rightarrow aABe \]
\[ A \rightarrow Abc \]
\[ A \rightarrow b \]
\[ B \rightarrow d \]

The sentence \( abbcde \) can be reduced to \( S \) by the following steps:

\[ abbcde \rightarrow aAbcde \rightarrow aAde \rightarrow aABe \rightarrow S \]

A rightmost derivation in reverse.
LR Parsing Actions

Here are the actions of the LR parsing machine and the customary representations of these actions in parsing tables.

- $sn$ Shift into state $n$; advance to next token in input
- $gn$ Goto state $n$;
- $rk$ Reduce by rule $k$;
- $a$ Accept (shifting the end-of-file marker);
- Error (represented by a blank entry in the parsing table)
LR Parsing Engine

input

a + b $

s_{m-1} s_m$

stack

goto action

right-most derivation
Push \((S_0)\); -- push start state

\(t := \text{GetNextToken}()\);

loop

\(S := \text{state at top of stack}\)

\(\text{case Action } [S, t] \text{ is}\)

when Error => Syntax Error;
when Accept => exit;
when Shift \(S' \Rightarrow\)
    Push \((S')\);
    \(t := \text{GetNextToken}()\);
when Reduce \(X \Rightarrow Y_1 \ldots Y_n \Rightarrow\)
    for each \(i \in n, \ldots, 1\) Pop(); end for;
    \(S' := \text{state at top of stack}\)
    Push (GoTo \([S', X]\));

end case

end loop
Appel, 2nd, Figure 3.18, page 56, an example of parsing the input
\[ a := 7; \quad b := c + (d := 5 + 6, d) $\]

using Grammar 3.1, page 40. The LR parsing table for Grammar 3.1 appears in Table 3.19, page 57.

1. Grammar 3.1, page 40
2. Table 3.19, page 57; the parsing table for Grammar 3.1
3. actions of the parsing engine on the input
Parsing Example: Grammar 3.1

Syntax for straight-line programs—Appel, 2nd, Grammar 3.1, page 40. The grammar is not augmented; the productions are numbered so that we may refer to them by number.

1. $S \rightarrow S ; S$
2. $S \rightarrow \text{id} := E$
3. $S \rightarrow \text{print} ( L )$
4. $E \rightarrow \text{id}$
5. $E \rightarrow \text{num}$
6. $E \rightarrow E + E$
7. $E \rightarrow ( S , E )$
8. $L \rightarrow E$
9. $L \rightarrow L , E$
Parsing Example: Parsing Table

Appel, 2nd, Figure 3.19, page 57. We do not yet say where we get this sort of parsing table.

<table>
<thead>
<tr>
<th>state</th>
<th>i</th>
<th>n</th>
<th>p</th>
<th>;</th>
<th>,</th>
<th>+</th>
<th>:=</th>
<th>(</th>
<th>)</th>
<th>$</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s4</td>
<td>s7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g2</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>s3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s4</td>
<td>s7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g5</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>r1</td>
<td>r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

228
Appel, 2nd, Figure 3.18, page 56, an example of parsing the input

\[ a := 7; \ b := c + (d := 5 + 6, d) \]

with the LR parsing engine. The LR parsing table for Grammar 3.1 appears in Table 3.19, page 57.

Begin by pushing start state 1 into the stack.

<table>
<thead>
<tr>
<th>step</th>
<th>stack (\rightarrow)</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>(a := 7; \ b := c + (d := 5 + 6, d))</td>
<td></td>
</tr>
</tbody>
</table>
Figure 3.18, page 58, an example of LR parsing.

The next action is determined by the state on the top of the stack 1, and by the first input symbol \( a \). Look up entry 1, \( a \) in the parsing table. The entry is “s4” meaning “shift 4.”

<table>
<thead>
<tr>
<th>step</th>
<th>stack ( \rightarrow )</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>( a := 7; \ b := c + (d := 5 + 6, d) $</td>
<td>shift 4</td>
</tr>
</tbody>
</table>
Figure 3.18, page 58, an example of LR parsing.

Shift the input token and push the new state into the stack.

<table>
<thead>
<tr>
<th>step</th>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>$a:=7; b:=c+(d:=5+6,d)$</td>
<td>shift 4</td>
</tr>
<tr>
<td>1</td>
<td>1 $4_{id}$ :=7; $b:=c+(d:=5+6,d)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 3.18, page 58, an example of LR parsing.

The next action is determined by the state on the top of the stack 4, and by the first input symbol :=. Look up entry 4, := in the parsing table. The entry is “s6” meaning “shift 6.”

<table>
<thead>
<tr>
<th>step</th>
<th>stack →</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>a:=7; b:=c+(d:=5+6,d)$</td>
<td>shift 4</td>
</tr>
<tr>
<td>1</td>
<td>4id</td>
<td>:=7; b:=c+(d:=5+6,d)$</td>
<td>shift 6</td>
</tr>
</tbody>
</table>
Figure 3.18, page 58, an example of LR parsing.

Shift the input token and push the new state into the stack.

<table>
<thead>
<tr>
<th>step</th>
<th>stack →</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>a:=7; b:=c+(d:=5+6,d)$</td>
<td>shift 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1 4_id</td>
<td>:=7; b:=c+(d:=5+6,d)$</td>
<td>shift 6</td>
</tr>
<tr>
<td>2</td>
<td>1 4_id 6:=</td>
<td>7; b:=c+(d:=5+6,d)$</td>
<td></td>
</tr>
</tbody>
</table>
Figure 3.18, page 58, an example of LR parsing.

The next action is determined by the state on the top of the stack 6, and by the first input symbol 7. Look up entry 6, 7 in the parsing table. The entry is “s10” meaning “shift 10.”

<table>
<thead>
<tr>
<th>step</th>
<th>stack ↦</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 4:id</td>
<td>a:=7; b:=c+(d:=5+6,d)$</td>
<td>shift 4</td>
</tr>
<tr>
<td>1</td>
<td>1 4:id</td>
<td>:=7; b:=c+(d:=5+6,d)$</td>
<td>shift 6</td>
</tr>
<tr>
<td>2</td>
<td>1 4:id 6:= 7; b:=c+(d:=5+6,d)$</td>
<td>shift 10</td>
<td></td>
</tr>
</tbody>
</table>
Figure 3.18, page 58, an example of LR parsing.

Shift the input token and push the new state into the stack.

<table>
<thead>
<tr>
<th>step</th>
<th>stack $\mapsto$</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>$a:=7; b:=c+(d:=5+6,d)$</td>
<td>shift 4</td>
</tr>
<tr>
<td>1</td>
<td>1 4 id</td>
<td>$:=7; b:=c+(d:=5+6,d)$</td>
<td>shift 6</td>
</tr>
<tr>
<td>2</td>
<td>1 4 id 6:=</td>
<td>7; $b:=c+(d:=5+6,d)$</td>
<td>shift 10</td>
</tr>
<tr>
<td>3</td>
<td>1 4 id 6:= 10 num</td>
<td>; $b:=c+(d:=5+6,d)$</td>
<td></td>
</tr>
</tbody>
</table>
Figure 3.18, page 58, an example of LR parsing.

The next action is determined by the state on the top of the stack \(10\), and by the first input symbol \(;\). Look up entry \(10, ;\) in the parsing table. The entry is “r5” meaning “reduce by production 5.” Lookup action.

<table>
<thead>
<tr>
<th>step</th>
<th>stack →</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>(a:=7; \ b:=c+(d:=5+6, d))$</td>
<td>shift 4</td>
</tr>
<tr>
<td>1</td>
<td>1 4 id</td>
<td>(:=7; \ b:=c+(d:=5+6, d))$</td>
<td>shift 6</td>
</tr>
<tr>
<td>2</td>
<td>1 4 id 6:=</td>
<td>7; \ b:=c+(d:=5+6, d))$</td>
<td>shift 10</td>
</tr>
<tr>
<td>3</td>
<td>1 4 id 6:=10 num</td>
<td>; \ b:=c+(d:=5+6, d))$</td>
<td>(r \ E \rightarrow \text{num})</td>
</tr>
</tbody>
</table>
Reduce using production $E \rightarrow \text{num}$. First, pop the RHS.

<table>
<thead>
<tr>
<th>step</th>
<th>stack $\rightarrow$ input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 \text{a:=7; b:=c+(d:=5+6,d)$}</td>
<td>shift 4</td>
</tr>
<tr>
<td>1</td>
<td>1 4\text{id} :=7; b:=c+(d:=5+6,d)$</td>
<td>shift 6</td>
</tr>
<tr>
<td>2</td>
<td>1 4\text{id} 6:= 7; b:=c+(d:=5+6,d)$</td>
<td>shift 10</td>
</tr>
<tr>
<td>3</td>
<td>1 4\text{id} 6:= 10\text{num}; b:=c+(d:=5+6,d)$</td>
<td>r $E \rightarrow \text{num}$</td>
</tr>
</tbody>
</table>

|     | 1 4\text{id} 6:= ; b:=c+(d:=5+6,d)$ | pop RHS |
Reduce using production $E \rightarrow \text{num}$. First, pop the RHS. Second, look in goto table with top of stack and LHS.

<table>
<thead>
<tr>
<th>step</th>
<th>stack $\mapsto$</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>a:=7; b:=c+(d:=5+6,d)$</td>
<td>shift 4</td>
</tr>
<tr>
<td>1</td>
<td>1 4\text{id}</td>
<td>:=7; b:=c+(d:=5+6,d)$</td>
<td>shift 6</td>
</tr>
<tr>
<td>2</td>
<td>1 4\text{id} 6:=</td>
<td>7; b:=c+(d:=5+6,d)$</td>
<td>shift 10</td>
</tr>
<tr>
<td>3</td>
<td>1 4\text{id} 6:= 10\text{num}</td>
<td>; b:=c+(d:=5+6,d)$</td>
<td>r $E \rightarrow \text{num}$</td>
</tr>
<tr>
<td></td>
<td>1 4\text{id} 6:=</td>
<td>; b:=c+(d:=5+6,d)$</td>
<td>pop RHS</td>
</tr>
<tr>
<td></td>
<td>1 4\text{id} 6:=</td>
<td>; b:=c+(d:=5+6,d)$</td>
<td>goto[6,E]=11</td>
</tr>
</tbody>
</table>
Reduce using production $E \rightarrow \text{num}$. First, pop the RHS. Second, look in goto table with top of stack and LHS. Third, push new state in stack.

<table>
<thead>
<tr>
<th>step</th>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>$a:=7; \ b:=c+(d:=5+6,d)$</td>
<td>shift 4</td>
</tr>
<tr>
<td></td>
<td>1 4 id</td>
<td>:=7; $b:=c+(d:=5+6,d)$</td>
<td>shift 6</td>
</tr>
<tr>
<td>2</td>
<td>1 4 id 6:=</td>
<td>7; $b:=c+(d:=5+6,d)$</td>
<td>shift 10</td>
</tr>
<tr>
<td>3</td>
<td>1 4 id 6:= 10 num</td>
<td>; $b:=c+(d:=5+6,d)$</td>
<td>$r \ E \rightarrow \text{num}$</td>
</tr>
<tr>
<td></td>
<td>1 4 id 6:=</td>
<td>; $b:=c+(d:=5+6,d)$</td>
<td>pop RHS</td>
</tr>
<tr>
<td></td>
<td>1 4 id 6:=</td>
<td>; $b:=c+(d:=5+6,d)$</td>
<td>goto[$6, E] = 11</td>
</tr>
<tr>
<td>4</td>
<td>1 4 id 6:= 11 $E$</td>
<td>; $b:=c+(d:=5+6,d)$</td>
<td></td>
</tr>
</tbody>
</table>
Lookup action.

<table>
<thead>
<tr>
<th>step</th>
<th>stack →</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>a:=7; b:=c+(d:=5+6,d)$</td>
<td>shift 4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4id :=7; b:=c+(d:=5+6,d)$</td>
<td>shift 6</td>
</tr>
<tr>
<td>2</td>
<td>1 4id 6:=</td>
<td>7; b:=c+(d:=5+6,d)$</td>
<td>shift 10</td>
</tr>
<tr>
<td>3</td>
<td>1 4id 6:=</td>
<td>10num ; b:=c+(d:=5+6,d)$</td>
<td>r $E \rightarrow \text{num}$</td>
</tr>
<tr>
<td>4</td>
<td>1 4id 6:=</td>
<td>11E ; b:=c+(d:=5+6,d)$</td>
<td>r $S \rightarrow \text{id := }E$</td>
</tr>
</tbody>
</table>
Reduce by production \( S \rightarrow \text{id} := E \).

<table>
<thead>
<tr>
<th>step</th>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>( a := 7; \ b := c + (d := 5 + 6, d) $</td>
<td>shift 4</td>
</tr>
<tr>
<td>1</td>
<td>( 4 \text{id} ) := 7; \ b := c + (d := 5 + 6, d) $</td>
<td>shift 6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( 4 \text{id} 6 := ) 7; \ b := c + (d := 5 + 6, d) $</td>
<td>shift 10</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>( 4 \text{id} 6 := 10 \text{num} ) ; \ b := c + (d := 5 + 6, d) $</td>
<td>r ( E \rightarrow \text{num} )</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>( 4 \text{id} 6 := 11_E ) ; \ b := c + (d := 5 + 6, d) $</td>
<td>r ( S \rightarrow \text{id} := E )</td>
<td></td>
</tr>
</tbody>
</table>
| 5    | \( 2_S \) ; \ b := c + (d := 5 + 6, d) \$ | }
## Lookup action.

<table>
<thead>
<tr>
<th>step</th>
<th>stack $\mapsto$ input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 $a:=7; ; b:=c+(d:=5+6,d)$</td>
<td>shift 4</td>
</tr>
<tr>
<td>1</td>
<td>1 4_id :=7; ; b:=c+(d:=5+6,d)$</td>
<td>shift 6</td>
</tr>
<tr>
<td>2</td>
<td>1 4_id 6:= 7; ; b:=c+(d:=5+6,d)$</td>
<td>shift 10</td>
</tr>
<tr>
<td>3</td>
<td>1 4_id 6:= 10_num ; b:=c+(d:=5+6,d)$</td>
<td>r $E \rightarrow \text{num}$</td>
</tr>
<tr>
<td>4</td>
<td>1 4_id 6:= 11_E ; b:=c+(d:=5+6,d)$</td>
<td>r $S \rightarrow \text{id} := E$</td>
</tr>
<tr>
<td>5</td>
<td>1 2_S ; b:=c+(d:=5+6,d)$</td>
<td>shift 3</td>
</tr>
</tbody>
</table>

LR parsing of input

Figure 3.18 — 14 of 45
Shift the input token and push the new state into the stack.

<table>
<thead>
<tr>
<th>step</th>
<th>stack →</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>a:=7; b:=c+(d:=5+6,d)$</td>
<td>shift 4</td>
</tr>
<tr>
<td>1</td>
<td>1 4id</td>
<td>:=7; b:=c+(d:=5+6,d)$</td>
<td>shift 6</td>
</tr>
<tr>
<td>2</td>
<td>1 4id 6:=</td>
<td>7; b:=c+(d:=5+6,d)$</td>
<td>shift 10</td>
</tr>
<tr>
<td>3</td>
<td>1 4id 6:= 10num</td>
<td>; b:=c+(d:=5+6,d)$</td>
<td>r E → num</td>
</tr>
<tr>
<td>4</td>
<td>1 4id 6:= 11E</td>
<td>; b:=c+(d:=5+6,d)$</td>
<td>r S → id := E</td>
</tr>
<tr>
<td>5</td>
<td>1 2S</td>
<td>; b:=c+(d:=5+6,d)$</td>
<td>shift 3</td>
</tr>
<tr>
<td>6</td>
<td>1 2S 3;</td>
<td>b:=c+(d:=5+6,d)$</td>
<td></td>
</tr>
</tbody>
</table>
Lookup action.

<table>
<thead>
<tr>
<th>step</th>
<th>stack →</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>a:=7; b:=c+(d:=5+6,d)$</td>
<td>shift 4</td>
</tr>
<tr>
<td>1</td>
<td>1 4 id :=7; b:=c+(d:=5+6,d)$</td>
<td>shift 6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1 4 id 6:= 7; b:=c+(d:=5+6,d)$</td>
<td>shift 10</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1 4 id 6:= 10 num ; b:=c+(d:=5+6,d)$</td>
<td>r $E \rightarrow \text{num}$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1 4 id 6:= 11 E ; b:=c+(d:=5+6,d)$</td>
<td>r $S \rightarrow \text{id} := E$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1 2 S ; b:=c+(d:=5+6,d)$</td>
<td>shift 3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1 2 S 3; b:=c+(d:=5+6,d)$</td>
<td>shift 4</td>
<td></td>
</tr>
</tbody>
</table>
Shift the input token and push the new state into the stack.

<table>
<thead>
<tr>
<th>step</th>
<th>stack $\rightarrow$</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>$a := 7; b := c + (d := 5 + 6, d)$</td>
<td>shift 4</td>
</tr>
<tr>
<td>1</td>
<td>1 4 id 7; b := c + (d := 5 + 6, d)$</td>
<td>shift 6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1 4 id 6 := 7; b := c + (d := 5 + 6, d)$</td>
<td>shift 10</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1 4 id 6 := 10 num ; b := c + (d := 5 + 6, d)$</td>
<td>$r \ E \rightarrow \ num$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1 4 id 6 := 11 E ; b := c + (d := 5 + 6, d)$</td>
<td>$r \ S \rightarrow \ id := E$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1 2 S ; b := c + (d := 5 + 6, d)$</td>
<td>shift 3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1 2 S 3 ; b := c + (d := 5 + 6, d)$</td>
<td>shift 4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1 2 S 3 ; 4 id := c + (d := 5 + 6, d)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lookup action.

<table>
<thead>
<tr>
<th>step</th>
<th>stack $\mapsto$</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>a:=7; b:=c+(d:=5+6,d)$</td>
<td>shift 4</td>
</tr>
<tr>
<td>1</td>
<td>1 4 id</td>
<td>:=7; b:=c+(d:=5+6,d)$</td>
<td>shift 6</td>
</tr>
<tr>
<td>2</td>
<td>1 4 id 6:=</td>
<td>7; b:=c+(d:=5+6,d)$</td>
<td>shift 10</td>
</tr>
<tr>
<td>3</td>
<td>1 4 id 6:= 10 num; b:=c+(d:=5+6,d)$</td>
<td>r $E \rightarrow$ num</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1 4 id 6:= 11E  ; b:=c+(d:=5+6,d)$</td>
<td>r $S \rightarrow$ id := $E$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1 2S           ; b:=c+(d:=5+6,d)$</td>
<td>shift 3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1 2S 3;        b:=c+(d:=5+6,d)$</td>
<td>shift 4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1 2S 3; 4 id   :=c+(d:=5+6,d)$</td>
<td>shift 6</td>
<td></td>
</tr>
</tbody>
</table>
Shift the input token and push the new state into the stack.

<table>
<thead>
<tr>
<th>step</th>
<th>stack $\mapsto$ input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 $a:=7; \ b:=c+(d:=5+6,d)$</td>
<td>shift 4</td>
</tr>
<tr>
<td>1</td>
<td>$1\ 4\text{id} \ :=7; \ b:=c+(d:=5+6,d)$</td>
<td>shift 6</td>
</tr>
<tr>
<td>2</td>
<td>$1\ 4\text{id} \ 6:= \ 7; \ b:=c+(d:=5+6,d)$</td>
<td>shift 10</td>
</tr>
<tr>
<td>3</td>
<td>$1\ 4\text{id} \ 6:= \ 10\text{num}; \ b:=c+(d:=5+6,d)$</td>
<td>$r \ E \to \text{num}$</td>
</tr>
<tr>
<td>4</td>
<td>$1\ 4\text{id} \ 6:= \ 11_E; \ b:=c+(d:=5+6,d)$</td>
<td>$r \ S \to \text{id} := E$</td>
</tr>
<tr>
<td>5</td>
<td>$1\ 2_S; \ b:=c+(d:=5+6,d)$</td>
<td>shift 3</td>
</tr>
<tr>
<td>6</td>
<td>$1\ 2_S \ 3; \ b:=c+(d:=5+6,d)$</td>
<td>shift 4</td>
</tr>
<tr>
<td>7</td>
<td>$1\ 2_S \ 3; \ 4\text{id} \ :=c+(d:=5+6,d)$</td>
<td>shift 6</td>
</tr>
<tr>
<td>8</td>
<td>$1\ 2_S \ 3; \ 4\text{id} \ 6:= \ c+(d:=5+6,d)$</td>
<td></td>
</tr>
</tbody>
</table>
Lookup action.

<table>
<thead>
<tr>
<th>step</th>
<th>stack $\mapsto$</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$1,4,id$</td>
<td>$:=7;\ b:=c+(d:=5+6,d)$</td>
<td>shift 6</td>
</tr>
<tr>
<td>2</td>
<td>$1,4,id\ 6:=</td>
<td>7;\ b:=c+(d:=5+6,d)$</td>
<td>shift 6</td>
</tr>
<tr>
<td>3</td>
<td>$1,4,id\ 6:=</td>
<td>10\text{num};\ b:=c+(d:=5+6,d)$</td>
<td>shift 10</td>
</tr>
<tr>
<td>4</td>
<td>$1,4,id\ 6:=</td>
<td>11_E;\ b:=c+(d:=5+6,d)$</td>
<td>r $E \rightarrow \text{num}$</td>
</tr>
<tr>
<td>5</td>
<td>$1,2_S$</td>
<td>$;\ b:=c+(d:=5+6,d)$</td>
<td>r $S \rightarrow \text{id} := E$</td>
</tr>
<tr>
<td>6</td>
<td>$1,2_S\ 3$</td>
<td>$;\ b:=c+(d:=5+6,d)$</td>
<td>shift 3</td>
</tr>
<tr>
<td>7</td>
<td>$1,2_S\ 3;\ 4,id$</td>
<td>$:=c+(d:=5+6,d)$</td>
<td>shift 4</td>
</tr>
<tr>
<td>8</td>
<td>$1,2_S\ 3;\ 4,id\ 6:=</td>
<td>c+(d:=5+6,d)$</td>
<td>shift 6</td>
</tr>
</tbody>
</table>

LR parsing of input
Shift the input token and push the new state into the stack.

<table>
<thead>
<tr>
<th>step</th>
<th>stack →</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 4 id =7; b:=c+(d:=5+6,d)$</td>
<td>shift 6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1 4 id 6:= 7; b:=c+(d:=5+6,d)$</td>
<td>shift 10</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1 4 id 6:= 10 num ; b:=c+(d:=5+6,d)$</td>
<td>r $E \rightarrow$ num</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1 4 id 6:= 11 e ; b:=c+(d:=5+6,d)$</td>
<td>r $S \rightarrow$ id := E</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1 2 s ; b:=c+(d:=5+6,d)$</td>
<td>shift 3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1 2 s 3; b:=c+(d:=5+6,d)$</td>
<td>shift 4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1 2 s 3; 4 id :=c+(d:=5+6,d)$</td>
<td>shift 6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1 2 s 3; 4 id 6:= c+(d:=5+6,d)$</td>
<td>shift 20</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1 2 s 3; 4 id 6:= 20 id + (d:=5+6,d)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lookup action.

<table>
<thead>
<tr>
<th>step</th>
<th>stack $\mapsto$</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1 4 id 6 := 7</td>
<td>b := c + (d := 5 + 6, d) $</td>
<td>shift 10</td>
</tr>
<tr>
<td>3</td>
<td>1 4 id 6 := 10 num</td>
<td>b := c + (d := 5 + 6, d) $</td>
<td>r $ E \rightarrow$ num</td>
</tr>
<tr>
<td>4</td>
<td>1 4 id 6 := 11 $E$</td>
<td>b := c + (d := 5 + 6, d) $</td>
<td>r $ S \rightarrow$ id := $E$</td>
</tr>
<tr>
<td>5</td>
<td>1 2 $S$</td>
<td>b := c + (d := 5 + 6, d) $</td>
<td>shift 3</td>
</tr>
<tr>
<td>6</td>
<td>1 2 $S$ 3 ;</td>
<td>b := c + (d := 5 + 6, d) $</td>
<td>shift 4</td>
</tr>
<tr>
<td>7</td>
<td>1 2 $S$ 3 ; 4 id</td>
<td>:= c + (d := 5 + 6, d) $</td>
<td>shift 6</td>
</tr>
<tr>
<td>8</td>
<td>1 2 $S$ 3 ; 4 id 6 :=</td>
<td>c + (d := 5 + 6, d) $</td>
<td>shift 20</td>
</tr>
<tr>
<td>9</td>
<td>1 2 $S$ 3 ; 4 id 6 := 20 id</td>
<td>$(d := 5 + 6, d) $</td>
<td>r $ E \rightarrow$ id</td>
</tr>
</tbody>
</table>
Reduce using production $E \to \text{id}$.

<table>
<thead>
<tr>
<th>step</th>
<th>stack $\mapsto$</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1 4 id 6:=</td>
<td>7; \ b:=c+(d:=5+6,d)$</td>
<td>shift 10</td>
</tr>
<tr>
<td>3</td>
<td>1 4 id 6:= 10 num</td>
<td>; \ b:=c+(d:=5+6,d)$</td>
<td>r \ E \to \text{num}</td>
</tr>
<tr>
<td>4</td>
<td>1 4 id 6:= 11 E</td>
<td>; \ b:=c+(d:=5+6,d)$</td>
<td>r \ S \to \text{id} := \ E</td>
</tr>
<tr>
<td>5</td>
<td>1 2 S</td>
<td>; \ b:=c+(d:=5+6,d)$</td>
<td>shift 3</td>
</tr>
<tr>
<td>6</td>
<td>1 2 S 3;</td>
<td>b:=c+(d:=5+6,d)$</td>
<td>shift 4</td>
</tr>
<tr>
<td>7</td>
<td>1 2 S 3; 4 id</td>
<td>:=c+(d:=5+6,d)$</td>
<td>shift 6</td>
</tr>
<tr>
<td>8</td>
<td>1 2 S 3; 4 id 6:=</td>
<td>c+(d:=5+6,d)$</td>
<td>shift 20</td>
</tr>
<tr>
<td>9</td>
<td>1 2 S 3; 4 id 6:= 20 id</td>
<td>+(d:=5+6,d)$</td>
<td>r \ E \to \text{id}</td>
</tr>
<tr>
<td>10</td>
<td>1 2 S 3; 4 id 6:= 11 E</td>
<td>+(d:=5+6,d)$</td>
<td></td>
</tr>
</tbody>
</table>
Lookup action.

<table>
<thead>
<tr>
<th>step</th>
<th>stack $\mapsto$</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1 4_{id} 6 := 10_{num} ; b := c+(d := 5+6, d)$</td>
<td></td>
<td>r $E \rightarrow \text{num}$</td>
</tr>
<tr>
<td>4</td>
<td>1 4_{id} 6 := 11_E ; b := c+(d := 5+6, d)$</td>
<td></td>
<td>r $S \rightarrow \text{id} := E$</td>
</tr>
<tr>
<td>5</td>
<td>1 2_S ; b := c+(d := 5+6, d)$</td>
<td></td>
<td>shift 3</td>
</tr>
<tr>
<td>6</td>
<td>1 2_S 3 ; b := c+(d := 5+6, d)$</td>
<td></td>
<td>shift 4</td>
</tr>
<tr>
<td>7</td>
<td>1 2_S 3 ; 4_{id} := c+(d := 5+6, d)$</td>
<td></td>
<td>shift 6</td>
</tr>
<tr>
<td>8</td>
<td>1 2_S 3 ; 4_{id} 6 := c+(d := 5+6, d)$</td>
<td></td>
<td>shift 20</td>
</tr>
<tr>
<td>9</td>
<td>1 2_S 3 ; 4_{id} 6 := 20_{id} + (d := 5+6, d)$</td>
<td></td>
<td>r $E \rightarrow \text{id}$</td>
</tr>
<tr>
<td>10</td>
<td>1 2_S 3 ; 4_{id} 6 := 11_E + (d := 5+6, d)$</td>
<td></td>
<td>shift 16</td>
</tr>
</tbody>
</table>
Shift the input token and push the new state into the stack.

<table>
<thead>
<tr>
<th>step</th>
<th>stack $\mapsto$ input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>$1 4\text{id }6:= 10\text{num }b:=c+(d:=5+6,d)$</td>
<td>$r\ E \rightarrow \text{num}$</td>
</tr>
<tr>
<td>4</td>
<td>$1 4\text{id }6:= 11_E \ b:=c+(d:=5+6,d)$</td>
<td>$r\ S \rightarrow \text{id }:= E$</td>
</tr>
<tr>
<td>5</td>
<td>$1 2_S \ b:=c+(d:=5+6,d)$</td>
<td>shift 3</td>
</tr>
<tr>
<td>6</td>
<td>$1 2_S 3; \ b:=c+(d:=5+6,d)$</td>
<td>shift 4</td>
</tr>
<tr>
<td>7</td>
<td>$1 2_S 3; 4\text{id }:=c+(d:=5+6,d)$</td>
<td>shift 6</td>
</tr>
<tr>
<td>8</td>
<td>$1 2_S 3; 4\text{id }6:= c+(d:=5+6,d)$</td>
<td>shift 20</td>
</tr>
<tr>
<td>9</td>
<td>$1 2_S 3; 4\text{id }6:= 20\text{id }+(d:=5+6,d)$</td>
<td>$r\ E \rightarrow \text{id}$</td>
</tr>
<tr>
<td>10</td>
<td>$1 2_S 3; 4\text{id }6:= 11_E + (d:=5+6,d)$</td>
<td>shift 16</td>
</tr>
<tr>
<td>11</td>
<td>$1 2_S 3; 4\text{id }6:= 11_E 16+(d:=5+6,d)$</td>
<td></td>
</tr>
</tbody>
</table>
Lookup action.

<table>
<thead>
<tr>
<th>step</th>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1 4 id 6 := 11E ; b := c + (d := 5 + 6, d) $</td>
<td>r $ S \rightarrow \text{id} := E$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1 2S ; b := c + (d := 5 + 6, d) $</td>
<td>shift 3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1 2S 3 ; b := c + (d := 5 + 6, d) $</td>
<td>shift 4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1 2S 3 ; 4 id := c + (d := 5 + 6, d) $</td>
<td>shift 6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1 2S 3 ; 4 id 6 := c + (d := 5 + 6, d) $</td>
<td>shift 20</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1 2S 3 ; 4 id 6 := 20 id ; (d := 5 + 6, d) $</td>
<td>r $ E \rightarrow \text{id}$</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1 2S 3 ; 4 id 6 := 11E ; (d := 5 + 6, d) $</td>
<td>shift 16</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1 2S 3 ; 4 id 6 := 11E 16 + (d := 5 + 6, d) $</td>
<td>shift 8</td>
<td></td>
</tr>
</tbody>
</table>
Shift the input token and push the new state into the stack.

<table>
<thead>
<tr>
<th>step</th>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>$1 4 \text{id} 6 := 11E$</td>
<td>; $b := c + (d := 5 + 6, d)$</td>
<td>$r \ S \rightarrow \text{id} := E$</td>
</tr>
<tr>
<td>5</td>
<td>$1 2S$</td>
<td>; $b := c + (d := 5 + 6, d)$</td>
<td>shift 3</td>
</tr>
<tr>
<td>6</td>
<td>$1 2S 3 ;$</td>
<td>$b := c + (d := 5 + 6, d)$</td>
<td>shift 4</td>
</tr>
<tr>
<td>7</td>
<td>$1 2S 3 ; 4 \text{id}$</td>
<td>:= $c + (d := 5 + 6, d)$</td>
<td>shift 6</td>
</tr>
<tr>
<td>8</td>
<td>$1 2S 3 ; 4 \text{id} 6 :=$</td>
<td>$c + (d := 5 + 6, d)$</td>
<td>shift 20</td>
</tr>
<tr>
<td>9</td>
<td>$1 2S 3 ; 4 \text{id} 6 := 20 \text{id}$</td>
<td>$(d := 5 + 6, d)$</td>
<td>$r \ E \rightarrow \text{id}$</td>
</tr>
<tr>
<td>10</td>
<td>$1 2S 3 ; 4 \text{id} 6 := 11E$</td>
<td>$(d := 5 + 6, d)$</td>
<td>shift 16</td>
</tr>
<tr>
<td>11</td>
<td>$1 2S 3 ; 4 \text{id} 6 := 11E 16 +$</td>
<td>$(d := 5 + 6, d)$</td>
<td>shift 8</td>
</tr>
<tr>
<td>12</td>
<td>$1 2S 3 ; 4 \text{id} 6 := 11E 16 + 8( \text{d := 5 + 6, d})$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lookup action.

<table>
<thead>
<tr>
<th>step</th>
<th>stack (\mapsto)</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1 (2_s)</td>
<td>(b:=c+(d:=5+6,d))$</td>
<td>shift 3</td>
</tr>
<tr>
<td>6</td>
<td>1 (2_s) 3;</td>
<td>(b:=c+(d:=5+6,d))$</td>
<td>shift 4</td>
</tr>
<tr>
<td>7</td>
<td>1 (2_s) 3; 4_{id}</td>
<td>(:=c+(d:=5+6,d))$</td>
<td>shift 6</td>
</tr>
<tr>
<td>8</td>
<td>1 (2_s) 3; 4_{id} 6:=</td>
<td>(c+(d:=5+6,d))$</td>
<td>shift 20</td>
</tr>
<tr>
<td>9</td>
<td>1 (2_s) 3; 4_{id} 6:= 20_{id}</td>
<td>((d:=5+6,d))$</td>
<td>r (E \to \text{id})</td>
</tr>
<tr>
<td>10</td>
<td>1 (2_s) 3; 4_{id} 6:= 11_{E}</td>
<td>((d:=5+6,d))$</td>
<td>shift 16</td>
</tr>
<tr>
<td>11</td>
<td>1 (2_s) 3; 4_{id} 6:= 11_{E} 16+</td>
<td>((d:=5+6,d))$</td>
<td>shift 8</td>
</tr>
<tr>
<td>12</td>
<td>1 (2_s) 3; 4_{id} 6:= 11_{E} 16+ 8( (\text{d}:=5+6,d))$</td>
<td>shift 4</td>
<td></td>
</tr>
</tbody>
</table>
Shift the input token and push the new state into the stack.

<table>
<thead>
<tr>
<th>step</th>
<th>stack →</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1 2_S</td>
<td>; b:=c+(d:=5+6,d)$</td>
<td>shift 3</td>
</tr>
<tr>
<td>6</td>
<td>1 2_S 3; b:=c+(d:=5+6,d)$</td>
<td>shift 4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1 2_S 3; 4_id :=c+(d:=5+6,d)$</td>
<td>shift 6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1 2_S 3; 4_id 6:= c+(d:=5+6,d)$</td>
<td>shift 20</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1 2_S 3; 4_id 6:= 20_id +(d:=5+6,d)$</td>
<td>r E → id</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1 2_S 3; 4_id 6:= 11_E +(d:=5+6,d)$</td>
<td>shift 16</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1 2_S 3; 4_id 6:= 11_E 16 + (d:=5+6,d)$</td>
<td>shift 8</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1 2_S 3; 4_id 6:= 11_E 16 + 8 (d:=5+6,d)$</td>
<td>shift 4</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1 2_S 3; 4_id 6:= 11_E 16 + 8 (4_id :=5+6,d)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>step</td>
<td>stack</td>
<td>input</td>
<td>action</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>6</td>
<td>1 2S 3;</td>
<td>b:=c+(d:=5+6,d)$</td>
<td>shift 4</td>
</tr>
<tr>
<td>7</td>
<td>1 2S 3; 4 id</td>
<td>:=c+(d:=5+6,d)$</td>
<td>shift 6</td>
</tr>
<tr>
<td>8</td>
<td>1 2S 3; 4 id 6:=</td>
<td>c+(d:=5+6,d)$</td>
<td>shift 20</td>
</tr>
<tr>
<td>9</td>
<td>1 2S 3; 4 id 6:= 20 id</td>
<td>+(d:=5+6,d)$</td>
<td>r $E \rightarrow$ id</td>
</tr>
<tr>
<td>10</td>
<td>1 2S 3; 4 id 6:= 11E</td>
<td>+(d:=5+6,d)$</td>
<td>shift 16</td>
</tr>
<tr>
<td>11</td>
<td>1 2S 3; 4 id 6:= 11E 16+</td>
<td>(d:=5+6,d)$</td>
<td>shift 8</td>
</tr>
<tr>
<td>12</td>
<td>1 2S 3; 4 id 6:= 11E 16+ 8(</td>
<td>d:=5+6,d)$</td>
<td>shift 4</td>
</tr>
<tr>
<td>13</td>
<td>1 2S 3; 4 id 6:= 11E 16+ 8(4 id :=r+6,d)$</td>
<td></td>
<td>shift 6</td>
</tr>
</tbody>
</table>
Shift the input token and push the new state into the stack.

<table>
<thead>
<tr>
<th>step</th>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1 2S 3;</td>
<td>b:=c+(d:=5+6, d)$</td>
<td>shift 4</td>
</tr>
<tr>
<td>7</td>
<td>1 2S 3; 4_id</td>
<td>:=c+(d:=5+6, d)$</td>
<td>shift 6</td>
</tr>
<tr>
<td>8</td>
<td>1 2S 3; 4_id 6:=</td>
<td>c+(d:=5+6, d)$</td>
<td>shift 20</td>
</tr>
<tr>
<td>9</td>
<td>1 2S 3; 4_id 6:= 20_id</td>
<td>+(d:=5+6, d)$</td>
<td>r E → id</td>
</tr>
<tr>
<td>10</td>
<td>1 2S 3; 4_id 6:= 11_E</td>
<td>+(d:=5+6, d)$</td>
<td>shift 16</td>
</tr>
<tr>
<td>11</td>
<td>1 2S 3; 4_id 6:= 11_E 16+</td>
<td>(d:=5+6, d)$</td>
<td>shift 8</td>
</tr>
<tr>
<td>12</td>
<td>1 2S 3; 4_id 6:= 11_E 16+8(</td>
<td>d:=5+6, d)$</td>
<td>shift 4</td>
</tr>
<tr>
<td>13</td>
<td>1 2S 3; 4_id 6:= 11_E 16+8( 4_id</td>
<td>:=5+6, d)$</td>
<td>shift 6</td>
</tr>
<tr>
<td>14</td>
<td>1 2S 3; 4_id 6:= 11_E 16+8( 4_id 6:=</td>
<td>r+6, d)$</td>
<td></td>
</tr>
</tbody>
</table>
Lookup action in parsing table.

<table>
<thead>
<tr>
<th>step</th>
<th>stack →</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1 2S 3; 4\text{id} :=c+(d:=5+6,d)$</td>
<td>shift 6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1 2S 3; 4\text{id} 6:= c+(d:=5+6,d)$</td>
<td>shift 20</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1 2S 3; 4\text{id} 6:= 20\text{id} + (d:=5+6,d)$</td>
<td>r \text{E} → \text{id}</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1 2S 3; 4\text{id} 6:= 11E + (d:=5+6,d)$</td>
<td>shift 16</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1 2S 3; 4\text{id} 6:= 11E 16+ (d:=5+6,d)$</td>
<td>shift 8</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1 2S 3; 4\text{id} 6:= 11E 16+ 8(d:=5+6,d)$</td>
<td>shift 4</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1 2S 3; 4\text{id} 6:= 11E 16+ 8(4\text{id} :=5+6,d)$</td>
<td>shift 6</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1 2S 3; 4\text{id} 6:= 11E 16+ 8(4\text{id} 6:= 5+6,d)$</td>
<td>shift 10</td>
<td></td>
</tr>
</tbody>
</table>
Shift the input token and push the new state into the stack.

<table>
<thead>
<tr>
<th>step</th>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1 2S 3; 4_id :=c+(d:=5+6,d)$</td>
<td>shift 6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1 2S 3; 4_id 6:= c+(d:=5+6,d)$</td>
<td>shift 20</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1 2S 3; 4_id 6:= 20_id + (d:=5+6,d)$</td>
<td>r $E \rightarrow \text{id}$</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1 2S 3; 4_id 6:= 11_E + (d:=5+6,d)$</td>
<td>shift 16</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1 2S 3; 4_id 6:= 11_E 16 + (d:=5+6,d)$</td>
<td>shift 8</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1 2S 3; 4_id 6:= 11_E 16 + 8( d:=5+6,d)$</td>
<td>shift 4</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1 2S 3; 4_id 6:= 11_E 16 + 8( 4_id :=5+6,d)$</td>
<td>shift 6</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1 2S 3; 4_id 6:= 11_E 16 + 8( 4_id 6:= 5+6,d)$</td>
<td>shift 10</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1 2S 3; 4_id 6:= 11_E 16 + 8( 4_id 6:= 10_num +6,d)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Lookup action in parsing table.**

<table>
<thead>
<tr>
<th>step</th>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1 2s 3; 4_id 6:= c+(d:=5+6,d)$</td>
<td>shift 20</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1 2s 3; 4_id 6:= 20id + (d:=5+6,d)$</td>
<td>r E → id</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1 2s 3; 4_id 6:= 11E + (d:=5+6,d)$</td>
<td>shift 16</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1 2s 3; 4_id 6:= 11E 16 + (d:=5+6,d)$</td>
<td>shift 8</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1 2s 3; 4_id 6:= 11E 16 + 8(d:=5+6,d)$</td>
<td>shift 4</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1 2s 3; 4_id 6:= 11E 16 + 8(4_id :=5+6,d)$</td>
<td>shift 6</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1 2s 3; 4_id 6:= 11E 16 + 8(4_id 6:= 5+6,d)$</td>
<td>shift 10</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1 2s 3; 4_id 6:= 11E 16 + 8(4_id 6:= 10num +6,d)$</td>
<td>r E → num</td>
<td></td>
</tr>
</tbody>
</table>

LR parsing of input
Reduce $E \rightarrow \text{num}$ and $\text{goto}[6, E] = 11$.

<table>
<thead>
<tr>
<th>step</th>
<th>stack $\mapsto$</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1 2S 3; 4_id 6 := c+(d:=5+6,d)$</td>
<td>shift 20</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1 2S 3; 4_id 6 := 20id</td>
<td>r $E \rightarrow \text{id}$</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1 2S 3; 4_id 6 := 11E</td>
<td>shift 16</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1 2S 3; 4_id 6 := 11E 16 + (d:=5+6,d)$</td>
<td>shift 8</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1 2S 3; 4_id 6 := 11E 16 + 8(d:=5+6,d)$</td>
<td>shift 4</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1 2S 3; 4_id 6 := 11E 16 + 8(4_id :=5+6,d)$</td>
<td>shift 6</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1 2S 3; 4_id 6 := 11E 16 + 8(4_id 6 :=5+6,d)$</td>
<td>shift 10</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1 2S 3; 4_id 6 := 11E 16 + 8(4_id 6 :=10num +6,d)$</td>
<td>r $E \rightarrow \text{num}$</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1 2S 3; 4_id 6 := 11E 16 + 8(4_id 6 :=11E +6,d)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
And more ...
<table>
<thead>
<tr>
<th>step</th>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>YY</td>
<td>$1 \ 2_s$</td>
<td>$$ $</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 3: Parsing

3.3. LR Parsing

LR(0) Parsing
LR(0) grammars are those that can be parsed looking only at the stack, making shift/reduce decisions without any lookahead. Though this class of grammars is too weak to be very useful, the algorithm for constructing LR(0) parsing tables is a good introduction to the LR(1) parser construction algorithm.

Apple, 2nd, Section 3.3, page 58.
Grammar 3.20

An example grammar for use in LR(0) parser generation; Appel, 2nd, Grammar 3.20, page 58.

0 \[ S' \rightarrow S \$

1 \[ S \rightarrow ( L )

2 \[ S \rightarrow \text{id}

3 \[ L \rightarrow S

4 \[ L \rightarrow L , S
LR(0) items

An LR(0) item is a production with a \( \bullet \) in it.

\[
A \rightarrow X_1 \cdots X_j \bullet X_{j+1} \cdots X_n
\]

The \( \bullet \) represents the current positions of the parser.

A state of the parser is a set of items.
Example

1:

\[ S' \rightarrow \bullet S \$ \]
\[ S \rightarrow \bullet (L) \]
\[ S \rightarrow \bullet \text{id} \]
Consider the NFA where $\delta(i, X)$ is “move the $\bullet$ over.”
Example

What happens if we shift an id in state 1?

2: \[ S \rightarrow \text{id} \bullet \]
Example

What happens if we shift an ( in state 1?

3:

\begin{align*}
S & \rightarrow ( \bullet L ) \\
L & \rightarrow \bullet L , S \\
L & \rightarrow \bullet S \\
S & \rightarrow \bullet ( L ) \\
S & \rightarrow \bullet \text{id}
\end{align*}

What happens after a reduction of an \( S \)-production in state 1?

4:

\begin{align*}
S' & \rightarrow S \bullet \$ 
\end{align*}
CLOSE

Algorithm to compute CLOSE[$I$] for set of items $I$.

CLOSE[$I$] := $I$
repeat
    for each item $A \rightarrow \alpha \bullet X \beta$ in CLOSE[$I$]
        for each production $X \rightarrow \gamma$ in the grammar
            add $X \rightarrow \bullet \gamma$ to $CLOSE[I]$
        end for
    end for
until no changes
GOTO is the transition function of a DFA. Algorithm to compute GOTO\[I, X\] for set of items I and terminals and nonterminals X.

\[
J := \emptyset \\
\text{for each item } A \rightarrow \alpha \bullet X \beta \text{ in } I \\
\quad \text{add } A \rightarrow \alpha X \bullet \beta \text{ to } J \\
\text{end for} \\
\text{GOTO}[I, X] := \text{CLOSE}[J]
\]
LR(0) parser

Algorithm to compute a graph with sets of items as nodes and directed edges labeled by the terminal and nonterminal symbols from the augmented grammar.

Begin by adding CLOSE[\{S' \rightarrow \bullet S\}] to the graph P representing the initial state of the parser.

repeat
    for each vertex \( I \in P \) and each grammar symbol \( X \)
        let \( J \) be the item set \( \text{GOTO}[I, X] \)
        add \( J \) to the vertices of \( P \)
        let \( e \) be the edge from \( I \) to \( J \) labeled \( X \)
        add \( e \) to the edges of \( P \)
    end for
until no changes
LR(0) Action Table

For each set of items $I$.

1. If $A \rightarrow \alpha \bullet t\beta$ is in $I$ for some terminal symbol $t$, then add the directive “shift $S$” to $\text{ACTION}[I, t]$, where $S$ is the item set $\text{GOTO}[I, t]$.

2. If $A \rightarrow \alpha \bullet$ is in $I$, then add the directive “reduce $A \rightarrow \alpha$” to $\text{ACTION}[I, t]$ for every terminal symbol $t$.

3. If $S \rightarrow \alpha \bullet \$ is in $I$, then add the directive “accept” to $\text{ACTION}[I, \$]$. 
Example LR(0) Parser Generation

Appel, 2nd, Grammar 3.20, page 58 [page 279].

Figure 3.21, page 61: DFA of sets of LR(0) items for Grammar 3.20.

Table 3.22, page 61: LR(0) parsing table for Grammar 3.20 [page 293].
Grammar 3.20

An example grammar for use in LR(0) parser generation; Appel, 2nd, Grammar 3.20, page 58.

0 $S' \rightarrow S\$
1 $S \rightarrow ( L )$
2 $S \rightarrow \text{id}$
3 $L \rightarrow S$
4 $L \rightarrow L, S$
\[
S' \rightarrow \bullet S \$
\]
\[
S \rightarrow \bullet ( L )
\]
\[
S \rightarrow \bullet \text{id}
\]
LR parser automaton
LR parser automaton

Figure 3.21 — 3 of 13
LR parser automaton
LR parser automaton

Figure 3.21 — 5 of 13
LR parser automaton

Figure 3.21 — 8 of 13
LR parser automaton
LR parser automaton
LR parser automaton
LR(0) Parsing Table for Grammar 3.20

Appel, 2nd, Table 3.22, page 61.

<table>
<thead>
<tr>
<th>state</th>
<th>action</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>id ,</td>
<td>$</td>
</tr>
<tr>
<td>1</td>
<td>s3 s2</td>
<td>g4</td>
</tr>
<tr>
<td>2</td>
<td>r2 r2 r2 r2 r2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s3 s2</td>
<td>g7 g5</td>
</tr>
<tr>
<td>4</td>
<td>acc</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s6 s8</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r1 r1 r1 r1 r1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r3 r3 r3 r3 r3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s3 s2</td>
<td>g9</td>
</tr>
<tr>
<td>9</td>
<td>r4 r4 r4 r4 r4</td>
<td></td>
</tr>
</tbody>
</table>
Loudin, Example 5.12

An example for LR(0) and SLR(1) parsing. Louden, Example 5.12, page 213.

\[
\begin{align*}
0 & \quad S' \rightarrow S \, \$ \\
1 & \quad S \rightarrow I \\
2 & \quad S \rightarrow o \\
3 & \quad I \rightarrow \, i \, S \\
4 & \quad I \rightarrow \, i \, S \, e \, S
\end{align*}
\]
\[
S' \rightarrow \bullet S \, \$ \\
S \rightarrow \bullet I \\
S \rightarrow \bullet o \\
I \rightarrow \bullet i S \\
I \rightarrow \bullet i S e S
\]
LR parser automaton
LR parser automaton

S' → •S$
S → •I
S → •o
I → •iS
I → •iSeS

S' → S •$
1

S → I •
2

S → o •
3
LR parser automaton

Louden, Figure 5.6 — 5 of 14
LR parser automaton
LR parser automaton
LR parser automaton 306 Louden, Figure 5.6 — 12 of 14
LR parser automaton
LR(0) parsing table for the grammar Louden, Example 5.12, page 213.

<table>
<thead>
<tr>
<th>state</th>
<th>action</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>i</td>
<td>e</td>
</tr>
<tr>
<td>0</td>
<td>s4</td>
<td>s3</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r1</td>
<td>r1</td>
</tr>
<tr>
<td>3</td>
<td>r2</td>
<td>r2</td>
</tr>
<tr>
<td>4</td>
<td>s4</td>
<td>s3</td>
</tr>
<tr>
<td>5</td>
<td>r3</td>
<td>s6/r3</td>
</tr>
<tr>
<td>6</td>
<td>s4</td>
<td>s3</td>
</tr>
<tr>
<td>7</td>
<td>r4</td>
<td>r4</td>
</tr>
</tbody>
</table>
SLR parsing table for Louden, Example 5.12, page 213 (still a conflict).

<table>
<thead>
<tr>
<th>state</th>
<th>action</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>i</td>
<td>e</td>
</tr>
<tr>
<td>0</td>
<td>s4</td>
<td>s3</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r1</td>
<td>r1</td>
</tr>
<tr>
<td>3</td>
<td>r2</td>
<td>r2</td>
</tr>
<tr>
<td>4</td>
<td>s4</td>
<td>s3</td>
</tr>
<tr>
<td>5</td>
<td>s6/r3</td>
<td>r3</td>
</tr>
<tr>
<td>6</td>
<td>s4</td>
<td>s3</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>r4</td>
</tr>
</tbody>
</table>

\[ \text{FLW}[S] = \text{FLW}[I] = \{e, $\} \]

Favoring a shift over a reduce in state 5 captures the rule that the “else” goes with the closest “if.”
Chapter 3: Parsing

3.3. LR Parsing

SLR Parsing
SLR stands for Simple LR. The idea is to reduce a production $A \rightarrow \alpha$ if lookahead is in $\text{FLW}[A]$.

(SLR($k$) is possible using $\text{FLW}_k[A]$ in place of $\text{FLW}[A] = \text{FLW}_1[A]$. SLR=SLR(1).)
SLR Action Table

For each set of items $I$.

1. If $A \rightarrow \alpha \cdot t\beta$ is in $I$ for some terminal symbol $t$, then add the directive “shift $S$” to $\text{ACTION}[I, t]$, where $S$ is the item set $\text{GOTO}[I, t]$.

2. If $A \rightarrow \alpha \cdot$ is in $I$, then add the directive “reduce $A \rightarrow \alpha$” to $\text{ACTION}[I, t]$ for every terminal $t$ in the $\text{FLW}[A]$.

3. If $S \rightarrow \alpha \cdot \$$ is in $I$, then add the directive “accept” to $\text{ACTION}[I, \$$].
Grammar 3.23

An SLR, but not LR(0) grammar. Apple, 2nd, Grammar 3.23, page 62.

\[
\begin{align*}
0 & \quad S' \rightarrow E \, $ \\
1 & \quad E \rightarrow T + E \\
2 & \quad E \rightarrow T \\
3 & \quad T \rightarrow \text{id}
\end{align*}
\]
Example LR(0) Parser Generation

Appel, 2nd, Figure 3.24, page 62.

1:

\[
S \rightarrow \cdot E \cdot$
\]
\[
E \rightarrow \cdot T + E
\]
\[
E \rightarrow \cdot T
\]
\[
T \rightarrow \cdot \text{id}
\]

2:

\[
S \rightarrow E \cdot \$
\]

3:

\[
E \rightarrow T \cdot + E
\]
\[
E \rightarrow T \cdot
\]
4:

\[ E \rightarrow T + \bullet E \]

\[ E \rightarrow \bullet T + \bullet E \]
LR(0) parsing table for Grammar 3.23.

<table>
<thead>
<tr>
<th>state</th>
<th>action</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id + $</td>
<td>$E T</td>
</tr>
<tr>
<td>1</td>
<td>s5</td>
<td>g2 g3</td>
</tr>
<tr>
<td>2</td>
<td>acc</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>r2 s4, r2</td>
<td>r2</td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td>g6 g3</td>
</tr>
<tr>
<td>5</td>
<td>r3 r3 r3</td>
<td>r3</td>
</tr>
<tr>
<td>6</td>
<td>r1 r1 r1</td>
<td>r1</td>
</tr>
</tbody>
</table>
LR(0) parser generation requires that we reduce on *all* terminals in states 3, 5, and 6.

Using $\text{FLW}[E] = \{\$\}$ (state 3), using $\text{FLW}[T] = \{+, \$\}$ (state 5), and using $\text{FLW}[E] = \{\$\}$ (state 6), we limit the reduce actions to cases that can possibly occur. Using FLW in this way produces an SLR parser.

Now the shift in state 3 (on $+$) does not conflict with a reduce action.
SLR Parsing Table for Grammar 3.23

Appel, 2nd, Figure 3.25, page 63: SLR parsing table for Grammar 3.23.

<table>
<thead>
<tr>
<th>state</th>
<th>action</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>s5</td>
<td>g2 g3</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>acc</td>
</tr>
<tr>
<td>3</td>
<td>s4 r2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td>g6 g3</td>
</tr>
<tr>
<td>5</td>
<td>r3 r3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>r1</td>
</tr>
</tbody>
</table>
Another Example of SLR(1) Parsing

Loudin, Example 5.12, page 213–215. Note the grammar is ambiguous.

Loudin, Figure 5.6, page 214: DFA of sets of LR(0) items for Examle 5.12

Loudin, Table 5.9, page 214: SLR(1) Parsing Table for Example 5.12
Chapter 3: Parsing

3.3. LR Parsing

LR(1) Parsing
LR(1) items

An LR(1) item is a production with a • in it and one terminal symbol of lookahead.

$$A \rightarrow X_1 \cdots X_j \bullet X_{j+1} \cdots X_n, \ z$$

The • represents the current position of the parser: α is on the top of the stack, and at the head of the input is a string derivable from βz.

A state of the parser is a set of items.
Example LR(1) Item Set

\[
\begin{align*}
S' & \rightarrow \bullet S \, \$ \quad ? \\
S & \rightarrow \bullet V = E \, \$ \\
S & \rightarrow \bullet E \quad \$ \\
E & \rightarrow \bullet V \quad \$ \\
V & \rightarrow \bullet \text{id} \quad \$ \\
V & \rightarrow \bullet \text{id} \quad = \\
V & \rightarrow \bullet \ast E \quad \$ \\
V & \rightarrow \bullet \ast E \quad =
\end{align*}
\]

A more compact notation:
\[ S' \rightarrow \bullet S \$ \]
\[ S \rightarrow \bullet V = E \$ \]
\[ S \rightarrow \bullet E \$ \]
\[ E \rightarrow \bullet V \$ \]
\[ V \rightarrow \bullet \text{id} \equiv,\$ \]
\[ V \rightarrow \bullet \ast E \equiv,\$ \]
CLOSE

Algorithm to compute $\text{CLOSE}[I]$ for set of items $I$.

$\text{CLOSE}[I] := I$

repeat
  for each item $A \rightarrow \alpha \bullet X\beta, z$ in $\text{CLOSE}[I]$
    for each production $X \rightarrow \gamma$ in the grammar
      for any $w \in \text{FIRST}[\beta z]$
        add $X \rightarrow \bullet \gamma, w$ to $\text{CLOSE}[I]$
      end for
    end for
  end for
until no changes
Algorithm to compute $\text{GOTO}[I, X]$ for set of items $I$ and terminals and nonterminals $X$. (Essentially the same as for the case of $LR(0)$, just different items.)

\[ J := \emptyset \]

for each item $A \rightarrow \alpha \cdot X \beta, z$ in $I$

\[ \text{add } A \rightarrow \alpha X \cdot \beta, z \text{ to } J \]

end

\[ \text{GOTO}[I, X] := \text{CLOSE}[J] \]
LR(1) parser

Same algorithm as in the case of $LR(0)$, just different GOTO function.

Algorithm to compute a graph with sets of items as nodes and directed edges labeled by the terminal and nonterminal symbols from the augmented grammar.

Begin by adding CLOSE[$\{S' \rightarrow \bullet S\$, $?\}$] to the graph $P$ representing the initial state of the parser.

repeat
    for each vertex $I \in P$ and each grammar symbol $X$
        let $J$ be the item set $\text{GOTO}[I, X]$
        add $J$ to the vertices of $P$
        let $e$ be the edge from $I$ to $J$ labeled $X$
        add $e$ to the edges of $P$
    end for
until no changes
LR(1) Action Table

For each set of items $I$.

1. If $A \rightarrow \alpha \bullet t\beta$, $z$ is in $I$ for some terminal symbol $t$, then add the directive “shift $S$” to $\text{ACTION}[I, t]$, where $S$ is the set $\text{GOTO}[I, t]$. (Lookahead $z$ is ignored.)

2. If $A \rightarrow \alpha \bullet$, $z$ is in $I$, then add the directive “reduce $A \rightarrow \alpha$” to $\text{ACTION}[I, z]$.

3. If $S \rightarrow \alpha \bullet \$, $?$ is in $I$, then add “accept” to $\text{ACTION}[I, \$]$. 
Example LR(1) Parser Generation


Figure 3.27, page 65: LR(1) states for Grammar 3.26.

Table 3.28, page 65, LR(1) parsing table for Grammar 3.26.
Grammar 3.26


0 \[ S' \rightarrow S \$ \]

1 \[ S \rightarrow V = E \]

2 \[ S \rightarrow E \]

3 \[ E \rightarrow V \]

4 \[ V \rightarrow \text{id} \]

5 \[ V \rightarrow \ast E \]
Example LR(1) Item Set Construction

Create the first state of the parser automaton. Begin with \( \{ S' \rightarrow \bullet S\$, $\}$, now compute closure:

1: \[ S' \rightarrow \bullet S$ \]
Example LR(1) Item Set Construction

Create the first state of the parser automaton. • before the nonterminal $S$.

1:

\[
\begin{align*}
S' & \rightarrow \bullet S \, \$ \quad ? \\
S & \rightarrow \bullet V = E \quad ? \\
S & \rightarrow \bullet E \quad ?
\end{align*}
\]
Example LR(1) Item Set Construction

Create the first state of the parser automaton. Compute FIRST[$\beta\alpha$] where $\beta = $, and $\alpha =$. So, FIRST[$\alpha $] = {$$}.

1:

$$S' \rightarrow \bullet S \$$
$$S \rightarrow \bullet V = E \$$
$$S \rightarrow \bullet E \$$
Example LR(1) Item Set Construction

Create the first state of the parser automaton. $\bullet$ before the nonterminal $V$.

1:

\[
\begin{align*}
S' & \rightarrow \bullet S \, \$ \, ? \\
S & \rightarrow \bullet V = E \, \$ \\
S & \rightarrow \bullet E \, \$ \\
V & \rightarrow \bullet \text{id} \, ? \\
V & \rightarrow \bullet \ast E \, ?
\end{align*}
\]
Example LR(1) Item Set Construction

Create the first state of the parser automaton. Compute FIRST[βz] where β is =E, and z is $. So, FIRST [=E $] is {=}.

1:

\[
\begin{align*}
S' & \rightarrow \bullet S \$ \quad ? \\
S & \rightarrow \bullet V = E \$ \\
S & \rightarrow \bullet E \quad $ \\
V & \rightarrow \bullet \text{id} \quad = \\
V & \rightarrow \bullet \ast E \quad = 
\end{align*}
\]
Example LR(1) Item Set Construction

Create the first state of the parser automaton. • before the nonterminal $E$.

1:

\[
\begin{align*}
S' & \rightarrow \bullet S \$ \ ? \\
S & \rightarrow \bullet V = E \$ \\
S & \rightarrow \bullet E \$ \\
E & \rightarrow \bullet V \ ? \\
V & \rightarrow \bullet \text{id} \ = \\
Va & \rightarrow \bullet * E \ =
\end{align*}
\]
Example LR(1) Item Set Construction

Create the first state of the parser automaton.

Compute \( \text{FIRST}[\beta z] \) where \( \beta = \epsilon \), and \( z = \$ \). So, \( \text{FIRST}[$$] = \{\$\} \).

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td></td>
</tr>
<tr>
<td>( S' \rightarrow \bullet S $ )</td>
<td>?</td>
</tr>
<tr>
<td>( S \rightarrow \bullet V = E $ )</td>
<td></td>
</tr>
<tr>
<td>( S \rightarrow \bullet E $ )</td>
<td></td>
</tr>
<tr>
<td>( E \rightarrow \bullet V $ )</td>
<td></td>
</tr>
<tr>
<td>( V \rightarrow \bullet \text{id} )</td>
<td>=</td>
</tr>
<tr>
<td>( V \rightarrow \bullet^* E )</td>
<td>=</td>
</tr>
</tbody>
</table>
Example LR(1) Item Set Construction

Now we have another $\bullet$ before the nonterminal $V$. As in LR(0) parsing no new productions will be added, but items with new lookaheads are possible.

Compute $\text{FIRST}[\beta z]$ where $\beta = \epsilon$, and $z =$ $. So, $\text{FIRST}[\$] = \{\$\}$. This lookahead must be added to the $V \rightarrow \bullet \cdots$ rules.

No further changes occur, so the construction is complete.

\[
\begin{array}{l}
S' \rightarrow \bullet S\$ \\
S \rightarrow \bullet V = E \$
\end{array}
\]

1:

\[
\begin{array}{l}
S' \rightarrow \bullet S\$ \\
S \rightarrow \bullet V = E \$
\end{array}
\]

\[
\begin{array}{l}
S \rightarrow \bullet E \\
E \rightarrow \bullet V \\
V \rightarrow \bullet \text{id} \quad =, \$
\end{array}
\]

\[
\begin{array}{l}
V \rightarrow \bullet^* E \quad =, \$
\end{array}
\]
LR(1) parser automaton

<table>
<thead>
<tr>
<th>Production</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S' \rightarrow \bullet S\ $</td>
<td>?</td>
</tr>
<tr>
<td>$S \rightarrow \bullet V = E\ $</td>
<td>$\ $</td>
</tr>
<tr>
<td>$S \rightarrow \bullet E\ $</td>
<td>$\ $</td>
</tr>
<tr>
<td>$E \rightarrow \bullet V\ $</td>
<td>$\ $</td>
</tr>
<tr>
<td>$V \rightarrow \bullet \text{id}\ $</td>
<td>$\ $</td>
</tr>
<tr>
<td>$V \rightarrow \bullet \ast E\ $</td>
<td>$\ $</td>
</tr>
</tbody>
</table>

Figure 3.27 — 1 of 14
LR(1) parser automaton
LR(1) parser automaton

Figure 3.27 — 3 of 14
LR(1) parser automaton

Figure 3.27 — 4 of 14
\[ S' \rightarrow S \cdot \$ \ ? \]

\[ S \rightarrow \cdot V = E \$ \]

\[ E \rightarrow \cdot V \$ \]

\[ S \rightarrow \cdot E \$ \]

\[ V \rightarrow \cdot \text{id} =, \$ \]

\[ V \rightarrow \cdot \star E =, \$ \]

\[ S \rightarrow \star V = \cdot E \$ \]

\[ E \rightarrow \cdot V \$ \]

\[ V \rightarrow \cdot \text{id} \$ \]

\[ V \rightarrow \cdot \star E \$ \]
LR(1) parser automaton
LR(1) parser automaton

Figure 3.27 — 7 of 14
LR(1) parser automaton
LR(1) parser automaton

Figure 3.27 — 9 of 14
LR(1) parser automaton

Figure 3.27 — 11 of 14
LR(1) parser automaton

Figure 3.27 — 12 of 14
LR(1) parser automaton
LR(1) parser automaton
Chapter 3: Parsing

3.3. LR Parsing

LALR(1) Parsing
LR(1) parsing tables can be very large, with many states. A smaller table can be made by merging any two states whose items are identical except for lookahead sets. The resulting parser is called an LALR(1) parser, for Look-Ahead LR(1).

For some grammars, the LALR(1) table contains reduce-reduce conflicts where the LR(1) table has none, but in practice the difference matters little. What does matter is that the LALR(1) parsing table requires less memory to represent than the LR(1) table, since there can be many fewer states.

Similar LR(0) Cores

\[\begin{align*}
V &\rightarrow \ast \bullet E =, \$ \\
E &\rightarrow \bullet V =, \$
\end{align*}\]

\[\begin{align*}
V &\rightarrow \bullet \text{id} =, \$
V &\rightarrow \bullet \ast E =, \$
\end{align*}\]

\[\begin{align*}
V &\rightarrow \ast \bullet E =, \$
E &\rightarrow \bullet V =, \$
\end{align*}\]

\[\begin{align*}
V &\rightarrow \bullet \text{id} =, \$
V &\rightarrow \bullet \ast E =, \$
\end{align*}\]
Simple LALR Parser Generation

Construct $C_{lr} = \{I_0, \ldots, I_n\}$, the collection of sets of LR(1) items.

Replace each subset of $C$ whose elements all have the same LR(0) core by their union $J_i$.

Let $C_{lalr} = \{J_0, \ldots, J_m\}$ be the resulting sets of LR(1) items.

Compute the parsing actions as before (for LR(1) parsing) The goto table will coalesce appropriately. But reduce/reduce conflicts may emerge.

Combining LR(1) states to form the DFA of LALR(1) items solves the problem of large parsing tables, but it still requires the entire DFA of LR(1) items to be computed. In fact, it is possible to compute the entire DFA of LALR(1) items directly from the DFA of LR(0) items through a process of propagating lookaheads.

Loudin, Section 5.4, page 225.
Efficient LALR Parser Generation

See ASU.
Chapter 3: Parsing

3.3. LR Parsing

LR Parsing of Ambiguous Grammars
LR Parsing of Ambiguous Grammars

Theoretically, it is impossible to parse an ambiguous grammar. But practically, it is possible to resolve some shift/reduce conflicts by favoring one or the other. This may be a convenient way of specifying an unambiguous grammar with requiring revisions in the input grammar.
Appel, Table 3.33, page 73, LR(1) parsing table for Grammar 3.5.

<table>
<thead>
<tr>
<th>state</th>
<th>id</th>
<th>num</th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>(</th>
<th>)</th>
<th>$</th>
<th>goto</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s2</td>
<td>s3</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td></td>
<td></td>
<td></td>
<td>g5</td>
</tr>
<tr>
<td>3</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s2</td>
<td>s3</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g7</td>
</tr>
<tr>
<td>5</td>
<td>s8</td>
<td>s10</td>
<td>s12</td>
<td>s14</td>
<td>acc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r7</td>
<td>r7</td>
<td>r7</td>
<td>r7</td>
<td>r7</td>
<td>r7</td>
<td>r7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>s8</td>
<td>s10</td>
<td>s12</td>
<td>s14</td>
<td>s6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s2</td>
<td>s3</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g9</td>
</tr>
<tr>
<td>9</td>
<td>s8/r5</td>
<td>s10/r5</td>
<td>s12/r5</td>
<td>s14/r5</td>
<td>r5</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>s2</td>
<td>s3</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g11</td>
</tr>
<tr>
<td>11</td>
<td>s8/r6</td>
<td>s10/r6</td>
<td>s12/r6</td>
<td>s14/r6</td>
<td>r6</td>
<td>r6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>s2</td>
<td>s3</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g13</td>
</tr>
<tr>
<td>13</td>
<td>s8/r3</td>
<td>s10/r3</td>
<td>s12/r3</td>
<td>s14/r3</td>
<td>r3</td>
<td>r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>s2</td>
<td>s3</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g15</td>
</tr>
<tr>
<td>15</td>
<td>s8/r4</td>
<td>s10/r4</td>
<td>s12/r4</td>
<td>s14/r4</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conflict Resolution

The classic shift/reduce conflict occurs when the LR(1) item $E \to E \odot \alpha, \ast$ indicating “shift” on $\odot$ is in the same item set as $E \to E \oplus E \bullet$, $\odot$ indicating “reduce” with production $E \to E \oplus E$ on $\odot$

$$
E \to E \odot E \bullet \odot \\
E \to E \bullet \odot \alpha \ast
$$

What is the correct parse tree for the sentential form $E \oplus E \odot \alpha$?

```
E
  / \  \\
E   E
  / \  \\
E   E
```

shift on $\oplus$

```
E
  /  \\
E   \odot
```

reduce on $\oplus$

```
E
  /  \\
E   \odot
```

alpha
Conflict Resolution

Higher precedence binds more tightly.

Compare the precedence of the lookahead token $\otimes$ with the precedence of the rule $E \oplus E$, which is determined, by default, to be the precedence of the last token.

- $\text{token } \otimes > \text{rule } \oplus$ shift
- equal, right associative shift
- equal, non-associative error
- equal, left associative reduce
- $\text{token } \otimes < \text{rule } \oplus$ reduce
Assuming “normal” precedence rules:

precence left +, - low
precence left *, / high

Appel, Table 3.34, page 75, conflicts of Table 3.33 resolved

<table>
<thead>
<tr>
<th>state</th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r5</td>
<td>r5</td>
<td>s12</td>
<td>s14</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>r6</td>
<td>r6</td>
<td>s12</td>
<td>s14</td>
</tr>
<tr>
<td>12</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
</tr>
</tbody>
</table>
Chapter 3: Parsing

3.4. Using Parser Generators
Chapter 3: Parsing

3.4. Using Parser Generators

Java CUP

CUP was written by Scott Hudson while at Georgia Institute of Technology. Version 0.10j of July 1999 is available at

http://www.cs.princeton.edu/~appel/modern/java/CUP/
Chapter 3: Parsing

3.4. Using Parser Generators

Conflicts
Chapter 3: Parsing

3.4. Using Parser Generators

Precedence Directives
Chapter 3: Parsing

3.5. Error Recovery
Review

- Regular expressions to NFA
- NFA to DFA
- Nullable, first, and follow
- LL(1) parsing
- LR(0), SLR, LR(1), LALR(1) parsing
More Generally

• Definition of formal language, regular expression
• Recursive descent parsers
• Scanners versus recognizers
• Definition of grammars, parse trees, ambiguity
• Hierarch of formal languages
Overview of Assignment 5

- Study AST for MiniJava
- Add to JavaCC parser semantic actions to create AST
- Understand visitor pattern
- Design symbol table
- Code visitor to create symbol table
- Code visitor to perform semantic checking.
Overview of Assignment 6

- Review the visitor pattern
- Review AST for MiniJava
- Study the give IR code
- Fix the design symbol table
- Detect missing semantics errors
- Study chapter 6
- Advise: use a \texttt{sparc} package, but ignore the books abstract classes
- Code visitor to translate to IR code
- Use or don’t use “generic IR trees”