Parsing Programming Language Grammars

Context Free Grammars

A context-free grammar G is defined as a 4-tuple $(\mathrm{N}, \Sigma, P, S)$ where N is the alphabet of **non-terminal** symbols used in the grammar, Σ is the set of **terminal** symbols used in the grammar and which make up the strings of the language accepted by the grammar, P is the set of **productions** or grammar rules, and $S \in N$ is the unique **start symbol** of the grammar. The productions in P are rules of the form $A \rightarrow \alpha$ where $A \in N$ and α is a string of symbols from $N \cup \Sigma$.[1]

So, for instance, a context-free grammar for a very small fragment of English might be given by:

$$
\begin{array}{rcll}
N & = & \{ & S, NP, VP, N, TV, IV \}^{2} \\
\Sigma & = & \{ & john, jane, loves, swims \} \\
P & = & \{ & S \rightarrow NP\ VP, \\
 & & & NP \rightarrow N, \\
 & & & VP \rightarrow IV, \\
 & & & VP \rightarrow TV\ NP, \\
 & & & N \rightarrow john, \\
 & & & N \rightarrow jane, \\
 & & & TV \rightarrow loves, \\
 & & & IV \rightarrow swims \} \\
S & = & & S
\end{array}
$$

As we will need to talk about the set $N \cup \Sigma$ frequently, we will give it the name V (for no particular reason). Borrowing the Kleene-star notation from regular expressions, the set of all strings of non-terminals and terminals is written V^*. Similarly, Σ^* is the set of all possible strings of terminal symbols (of which the grammar generates some subset). We will use the Greek letters $\alpha, \beta, \gamma, \ldots$ to denote strings from V^* and $u, v, w, \ldots$ to denote strings from Σ^*. Uppercase letters, $A, B, C, \ldots$ will be used to denote individual non-terminals from the set N, and lowercase letters, $a, b, c, \ldots$ will be used to denote individual terminals from the set Σ

Given a a string $\alpha A \beta \in V^*$ and a production $A \rightarrow \gamma \in P$, we say that $\alpha A \beta$ **derives** $\alpha \gamma \beta$ in a single step, written $\alpha A \beta \Rightarrow \alpha \gamma \beta$. We extend this definition by saying that the string

[1] These notes are derived in large part from notes and papers distributed by Dr. Jean Gallier in his graduate compiler construction course taught at the University of Pennsylvania during the mid-1980's.

[2] Sentences, Noun Phrases, Verb Phrases, Nouns, Transitive Verbs, Intransitive Verbs

α derives β in one or more steps (written $\alpha \stackrel{+}{\Rightarrow} \beta$) if there exists a value $n \geq 1$, and strings $\alpha_0, \alpha_1, \ldots, \alpha_n \in V^*$ such that $\alpha_0 = \alpha$ and $\alpha_n = \beta$ and $\alpha = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \cdots \Rightarrow \alpha_{n-1} \Rightarrow \alpha_n = \beta$. We say α derives β (written $\alpha \stackrel{*}{\Rightarrow} \beta$) if either $\alpha = \beta$ or $\alpha \stackrel{+}{\Rightarrow} \beta$.

From among all derivations we distinguish two useful classes: A derivation $\alpha \stackrel{+}{\Rightarrow} \beta$ is a **leftmost derivation** if at each step in the derivation the next production is applied to the leftmost non-terminal in the string at that point. Similarly, a derivation is a **rightmost derivation** if at each step in the derivation the next production is applied to the rightmost non-terminal in the string at that point. We denote these sorts of derivations with the arrows $\stackrel{+}{\underset{lm}{\Rightarrow}}$, $\stackrel{*}{\underset{lm}{\Rightarrow}}$, $\stackrel{+}{\underset{rm}{\Rightarrow}}$, and $\stackrel{*}{\underset{rm}{\Rightarrow}}$.

So, for instance, the derivation:

$$\begin{aligned} S &\Rightarrow NP\ VP \Rightarrow N\ VP \Rightarrow john\ VP \Rightarrow john\ TV\ NP \\ &\Rightarrow john\ loves\ NP \Rightarrow john\ loves\ N \Rightarrow john\ loves\ jane \end{aligned}$$

is a leftmost derivation, while:

$$\begin{aligned} S &\Rightarrow NP\ VP \Rightarrow NP\ TV\ NP \Rightarrow NP\ TV\ N \Rightarrow NP\ TV\ jane \\ &\Rightarrow NP\ loves\ jane \Rightarrow N\ loves\ jane \Rightarrow john\ loves\ jane \end{aligned}$$

is a rightmost derivation of the same string.

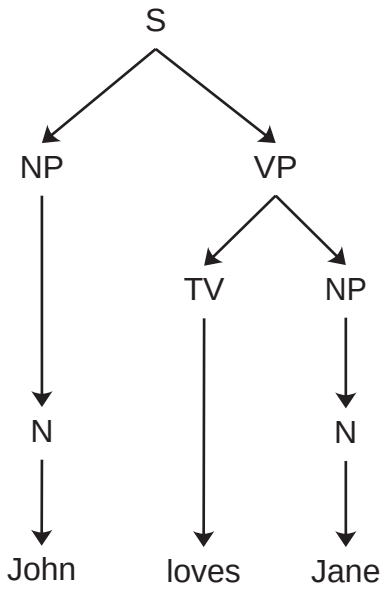
Strings w of terminals that can be derived from the start symbol S of the grammar (that is, the strings in the language generated by the grammar) are called **sentences**. Strings of terminals and non-terminals that can be derived from S (that is, the strings that occur along the way in the derivation of sentences) are called **sentential forms**.

Given any derivation $A \stackrel{+}{\Rightarrow} \alpha$ there is a uniquely determined corresponding **parse tree** (or **derivation tree**). This tree has its root labeled by A and its leaves labeled by the symbols in α in order. Internal nodes are labeled by non-terminals and represent steps of the derivation. The children of a node are labeled with the symbols from the right hand side of the production rule applied at that step in the derivation.

Note that a given parse tree can result from several different derivations, but a given derivation corresponds to only a single parse tree.

We will generally be interested only in parse trees that have their roots labeled by S, the start symbol of the grammar under consideration, as the leaves of these trees form sentences (or sentential forms) in the language generated by the grammar.

Both the derivations above build the following parse tree, though they build it in different orders:



Generation and Recognition of Strings

As we have just seen, a context-free grammar can be used to **generate** strings in the language of the grammar by simply starting with the start symbol and applying a series of derivation steps until one arrives at a string of terminal symbols.

More importantly for our purposes, the grammar can also be used to **recognize** whether a given string of terminal symbols belongs to the language generated by the grammar. The process involves analyzing the string to see whether it is possible to build a legal derivation tree for it using the productions in the grammar. This process is called **parsing** the string, and the machine or program that performs the analysis is called a **parser**.

There are two basic strategies for parsing, based on whether the parser attempts to build a derivation tree for the input string starting from the top at the root, or from the bottom at the leaves.

A **top-down parser** begins at the root with the start symbol and applies one of the rules for expanding that non-terminal. It then picks one of the resulting leaves (usually the leftmost) and proceeds recursively. At some point it will find that the leftmost leaf is a terminal symbol. At that point it compares the leaf with the first symbol in the input stream. If they match it continues with the next leaf, and so on. If it gets to a leaf that is still a non-terminal it goes back to the recursive expansion process. If at some point it finds a terminal symbol in a leaf that does not match the corresponding symbol in the input stream it goes back to the last place it made a choice for how to expand a non-terminal symbol and picks a different rule if one exists.

This process continues either until the entire input string has been matched, in which case it is **accepted**, or until the parser has run out of places to backtrack to to pick a different rule, in which case the input string is **rejected**.

While this procedure sounds enormously costly and non-deterministic, special versions

of it have been invented for certain classes of grammars that are surprisingly well behaved.

A **Bottom-up parser** begins at the leaves and scans the input (usually left-to-right). Occasionally, as it moves along, the parser recognizes a string of contiguous symbols as matching the right hand side of some production in the grammar. When this happens, the production is applied to yield a parent node for those symbols. At all times the parser considers the upper edge of the parse tree as the current input string that it is evaluating.

At present, due to developments in parser construction during the last several decades, bottom-up parsers are generally favored for programming languages.

The SLR(1) Parser Construction Algorithm

The SLR(1) parser construction algorithm builds bottom-up parsers for a certain class of context-free grammars. At this point I will not describe the properties that a grammar must have for the algorithm to work. Suffice to say that most programming language grammars are sufficiently restricted as to belong to the the class, or can easily be massaged so that they do.

The parser that the algorithm builds is capable of processing its input (and building a parse tree for it bottom-up) in a single left-to-right pass, using no lookahead. If the input string is accepted, the parse tree constructed corresponds to one that would be built in a rightmost derivation of the input string.

In order to understand the SLR(1) algorithm, we will first need to make a few technical definitions.

Viable Prefixes and Characteristic Languages

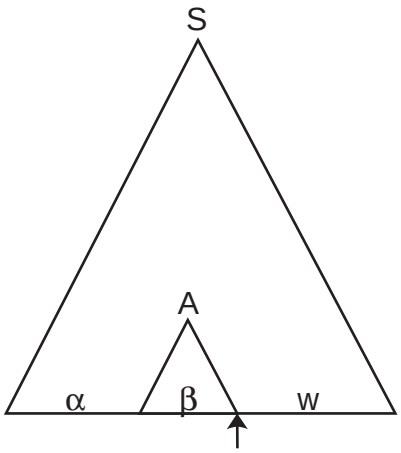
Consider the set of strings of the form $\alpha\beta \in V^*$ such that $S \overset{+}{\underset{rm}{\Rightarrow}} \alpha A w$ and $\alpha A w \Rightarrow \alpha\beta w$ in a single step using the production $A \rightarrow \beta$ (where $\alpha, \beta \in V^*$ are strings of terminals and non-terminals, $A \in N$ is a non-terminal, and $w \in \Sigma^*$ is a string of terminals).

Such strings are called **viable prefixes** of the language generated by the grammar, the set of all such strings is called the **characteristic language** of the grammar. So, for instance, given the grammar

$$G = (\{E\}, \{a, b\}, \{E \rightarrow aEb, E \rightarrow ab\}, E)$$

the characteristic language of G consists of the set of all strings of the form $a^n b$, and $a^n E b$, where $n \geq 1$.

Why are these strings interesting? Well, the sentential forms $\alpha A w$ and $\alpha\beta w$ are exactly the ones that occur along the way in the construction of a rightmost derivation using a left-to-right pass through the input. Consider the following diagram of a derivation tree for the sentential forms $\alpha A w$ and $\alpha\beta w$:



Since the parser is building the derivation from the bottom up while scanning left-to-right, the trick will be to recognize when the material to the left of the input pointer is of the form $\alpha\beta$ and that the parser may therefore apply (in reverse) the production $A \rightarrow \beta$ to yield αA. The string w is just the as yet unanalyzed portion of the input string to the right of the current input pointer.

So, being able to identify the viable prefixes of the language enables us to make a guess as to when it is time to apply a production.

The LR(1) and SLR(1) parser construction algorithms are dependent on the following fundamental (and fortuitous) result:

> For any context free grammar G, the characteristic language of G, consisting of all the viable prefixes of the language generated by G, is regular.

This means that the characteristic strings can be recognized using the relatively simple machinery of a finite automaton.

The parser construction algorithm therefore begins by constructing the **characteristic automaton** of the grammar, which is a finite automaton that recognizes the characteristic language of the grammar.

Before we continue, it should be mentioned that part of the way that an SLR(1) parser manages to do its work without any lookahead is by delaying its decisions so that reductions alway apply to the portions of the input string to the left of the current input pointer. That is, we reduce $\alpha\beta$ to αA only once the input pointer is pointing to the first symbol in w, the string of terminals following β. Therefore, it is necessary to augment the grammar for which we wish to build a parser with a new production $S' \rightarrow S\$$, where S is the original start state of the grammar, S' is a new start state, and \$ is a new explicit end-of-input marker. This insures that the parser will have a token to point at while it makes final decisions about whether the input string can be reduced to an S. Thus, to build a parser for the grammar G given above, we would consider the augmented grammar:

$$G' = (\{S', E\}, \{a, b\}, \{S' \rightarrow E\$, E \rightarrow aEb, E \rightarrow ab\}, S')$$

1) Constructing the Characteristic NFA

The characteristic Non-deterministic Finite Automaton for the augmented grammar $G' = (N', \Sigma', P', S')$ is an automaton whose input symbols are drawn from the set $V' = N' \cup \Sigma'$. The states of the automaton each correspond to a single **item** or **marked production** derived from P' as follows:

> If the rule $A \rightarrow \alpha$ is in P' and α is a string of n symbols from V', then this rule generates $n + 1$ distinct marked productions. Each is identical to the original rule, except that a single instance of the new character "." is inserted somewhere in the right hand side.

So, for instance, the rule $E \rightarrow aEb$ produces the four marked productions

$$E \rightarrow .aEb \qquad E \rightarrow a.Eb \qquad E \rightarrow aE.b \qquad E \rightarrow aEb.$$

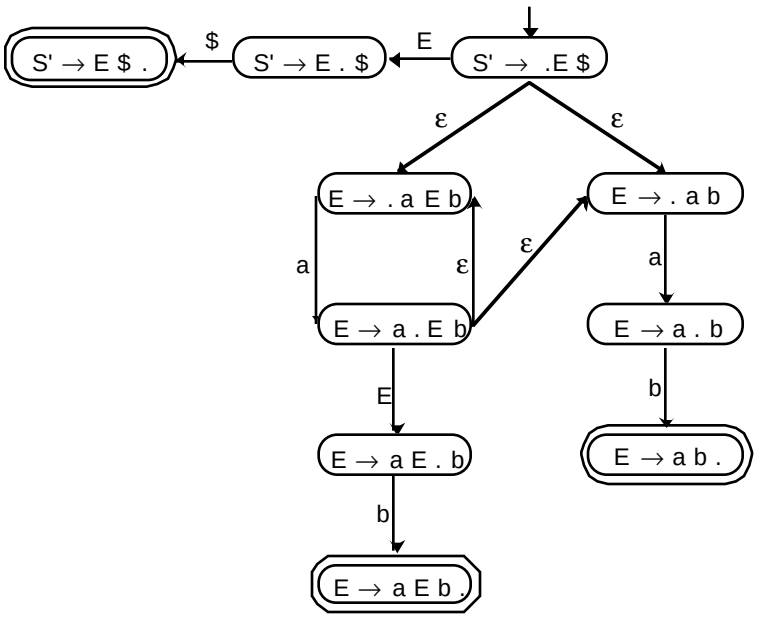
Thus the characteristic NFA for the simple grammar G' above has 10 states.

The initial state of the characteristic NFA is the state corresponding to the item $S' \rightarrow .S\$$. The final states are those corresponding to items of the form $A \rightarrow \alpha.$ with the dot in the rightmost position.
The transitions out of a state are determined as follows:

1. If the state corresponds to the item $A \rightarrow \alpha.a\beta$, where $a \in \Sigma'$, there is a single transition, labeled by a, out of the state and leading to the state corresponding to the item $A \rightarrow \alpha a.\beta$

2. If the state corresponds to the item $A \rightarrow \alpha.A\beta$, where $A \in N'$, there is a transition, labeled by A, out of the state and leading to the state corresponding to the item $A \rightarrow \alpha A.\beta$. In addition, for each rule of the form $A \rightarrow \gamma$ in P', there is an ϵ-transition to the state corresponding to the item $A \rightarrow .\gamma$

Thus the simple grammar G' above has the following characteristic non-deterministic finite automaton:



2) Constructing the Characteristic DFA

Having constructed the characteristic non-deterministic finite automaton for the grammar, the next step is to convert it into a deterministic finite automaton. This is accomplished by using the standard algorithm for building a DFA equivalent to a given NFA. In order to state that algorithm, we must first make the following definition:

> Given a state s in an NFA, the **ϵ-closure** of s is the set of states satisfying the following conditions:
>
> 1. $s \in \epsilon$-closure(s)
> 2. If the state t is in the ϵ-closure(s) and there is an ϵ-transition from t to another state u, then u is also in the ϵ-closure(s).
>
> That is, the ϵ-closure of s is all the states that can be reached from s using only ϵ-transitions.

By extension, we also define the ϵ-closure of a set of states S as

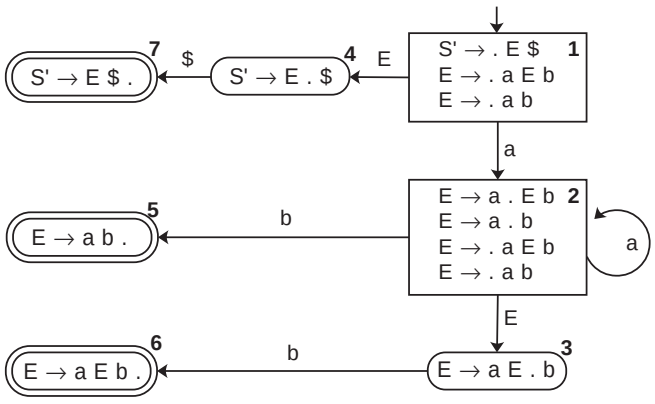
$$\epsilon\text{-closure}(S) = \bigcup_{s \in S} \epsilon\text{-closure}(s)$$

In the following algorithm we will denote states in the original NFA by subscripting them with an N and those in the DFA under construction by subscripting them with a D. Recall that each state in the DFA corresponds to a set of states from the NFA. The algorithm proceeds as follows:

1. If S_{0N} is the start state of the NFA, let S_{0D}, the start state of the DFA, be $S_{0D} = \epsilon$-closure(S_{0N}). Initially, the state S_{0D} is *unmarked*.

2. Select an unmarked state S_{iD} in the DFA under construction and mark it. If there are no unmarked states, go to step 5.
3. For each symbol a in the input symbol set of the NFA:
   (a) Let $T = \{t_N |$there is a state $s_N \in S_{iD}$ and a transition in the NFA from s_N to t_N on input $a\}$.
   (b) Let $S_{jD} = \epsilon$-closure(T).
   (c) If S_{jD} is not already a state in the DFA, add it unmarked.
   (d) Add a transition from S_{iD} to S_{jD} on input a.
4. Return to step 2.
5. The final (accepting) states of the DFA are exactly those which contain a final state of the NFA.

Running this algorithm to convert the characteristic NFA of the simple grammar G' above yields the following DFA:



It should be easy to see that this DFA accepts the characteristic language of G which we said was the set of all strings of the form $a^n b$, and $a^n E b$, where $n \geq 1$.

3) Constructing the SLR(1) Parse Table

With the construction of the characteristic finite automaton completed, the last step consists of examining the transitions of the automaton and extracting from them the definition of the SLR(1) **parse table** for the grammar. The parse table is simply a description of what to do at each step of the parse given the state the parser is currently in and the terminal symbol that the input pointer is currently pointing to.

The states of the parser correspond to the states of the characteristic DFA. Therefore, the parse table has one row for each state of the characteristic DFA, except for the state labeled by $S' \rightarrow S\$.$ which is extraneous. It is divided horizontally into two parts. The **action table** has one column for each terminal symbol in Σ' (that is, *including* the special terminal symbol \$. The **goto table** has one column for each non-terminal symbol in N (that is, *not including* the special non-terminal S').

The First and Follow Sets of a Non-terminal

Before we can fill in the table we will first need to define, and show how to compute, the following two sets:

- If $A \in N$ then First(A)= $\{a | a \in \Sigma', A \stackrel{+}{\Rightarrow} a\alpha\}$. That is, it is the set of all terminals that can occur as the first symbol in a sentential form derived from A.
- If $A \in N$ then Follow(A)= $\{a | a \in \Sigma', B \stackrel{+}{\Rightarrow} \alpha A a \beta, B \in N'\}$. That is, it is the set of all terminals that can occur immediately after A in a sentential form.

The First sets of all the non-terminals are computed as follows:

1. For each $A \in N$, let InitFirst(A)= $\{a | a \in \Sigma', A \rightarrow a\alpha \in P'\}$
2. Construct a directed precedence graph for the non-terminals of the grammar such that there is an edge from A to B if and only if there is a rule of the form $A \rightarrow B\alpha$ in the augmented grammar (that is, if B can occur as the first item in an expansion of A, which means that anything that can occur first in a B can occur first in an A).
3. For each non-terminal A, if R is the set of non-terminals reachable from A in the precedence graph, then:

$$\text{First}(A) = \text{InitFirst}(A) \cup \bigcup_{B \in R} \text{InitFirst}(B)$$

For the simple grammar G, there is only one non-terminal E, so First(E) = InitFirst(E) = $\{a\}$.

The Follow sets of all the non-terminals are computed as follows:

1. For each $A \in N$, let InitFollow(A)= $\{a | a \in \Sigma', B \rightarrow \alpha A a \beta \in P'\} \cup \{a | a \in \Sigma', B \rightarrow \alpha A C \beta \in P', a \in \text{First}(C)\}$

2. Construct a directed precedence graph for the non-terminals of the grammar such that there is an edge from A to B if and only if there is a rule of the form $B \rightarrow \alpha A$ in the augmented grammar (that is, if A can occur as the last item in an expansion of B, which means that anything that can follow a B can follow an A).

3. For each non-terminal A, if R is the set of non-terminals reachable from A in the precedence graph, then:

$$\text{Follow}(A) = \text{InitFollow}(A) \cup \bigcup_{B \in R} \text{InitFollow}(B)$$

As above, for the simple grammar G, there is only one non-terminal E, so Follow(E) = InitFollow(E) = $\{b, \$\}$. Note we are only building First and Follow sets for the non-terminals in core grammar, but the production rules used are those in the augmented grammar, which is why \$ is in Follow(E).

Filling in the Parse Table

The cells of table are filled in according to the following rules. The first 3 rules explain how to fill in the action table, the fourth rule explains how to fill in the goto table.

1. If state I of the characteristic DFA contains the item $A \rightarrow \alpha . a \beta$ and there is a transition from state I to state J on input a, then action(I,a)=Shift(J). This is generally abbreviated SJ.

2. If state I contains an item of the form $A \rightarrow \alpha .$, then for every terminal $a \in$Follow(A), action(I,a)=Reduce($A \rightarrow \alpha$). This is generally abbreviated RJ where J is the number of the production rule being used for the reduction.

3. If state I contains the item $S' \rightarrow S.\$$, then action(I,\$)=accept.

4. If state I contains the item $B \rightarrow \alpha . A \beta$ and there is a transition from state I to state J on input A, then goto(I,A)= J

Using these rules, we see that the SLR(1) parse table for the simple grammar G' is given by:

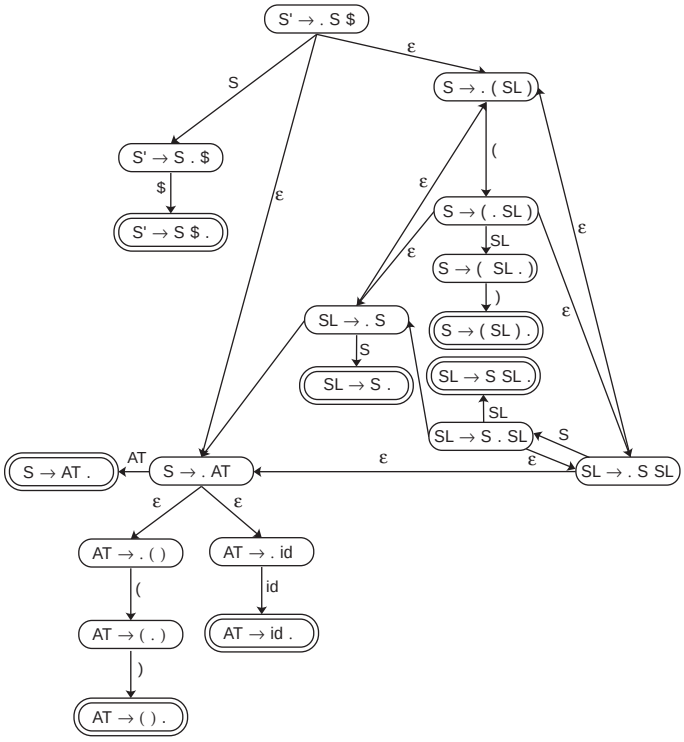
State	Action			Goto
	a	b	$	E
1	Shift(2)			4
2	Shift(2)	Shift(5)		3
3		Shift(6)		
4			accept	
5		Reduce($E \rightarrow ab$)	Reduce($E \rightarrow ab$)	
6		Reduce($E \rightarrow aEb$)	Reduce($E \rightarrow aEb$)	

A Bigger Example

As a larger example, consider the following simplified version of the s-expression grammar used for the course projects:

$$
\begin{array}{rcl}
N & = & \{ \quad S, SL, AT \ \}^{3} \\
\Sigma & = & \{ \quad (,), id \ \}^{4} \\
P & = & \{ \quad 1 : S \rightarrow AT, \\
 & & \quad\ \ 2 : S \rightarrow (\ SL\ ), \\
 & & \quad\ \ 3 : SL \rightarrow S, \\
 & & \quad\ \ 4 : SL \rightarrow S\ SL, \\
 & & \quad\ \ 5 : AT \rightarrow (\ ), \\
 & & \quad\ \ 6 : AT \rightarrow id \ \} \\
S & = & \quad S
\end{array}
$$

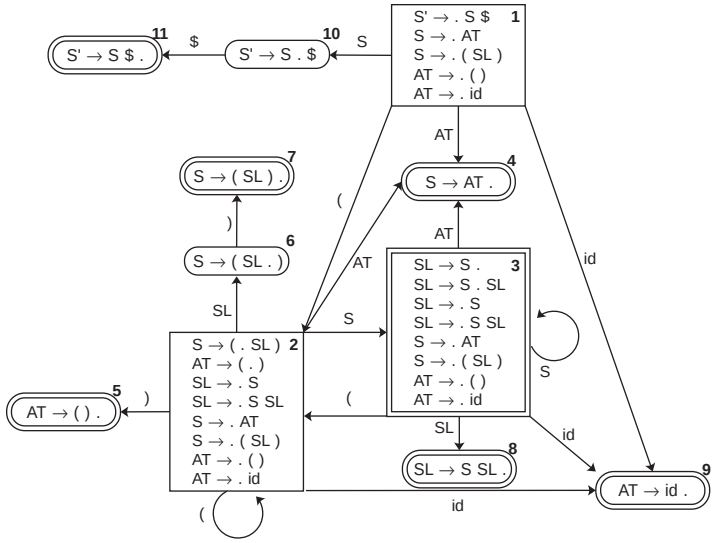
The characteristic non-deterministic finite automaton for the augmented version of this grammar is given by:



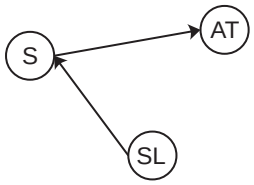
After running the algorithm to convert this to a deterministic finite automaton, we arrive at the following machine:

[3] S-Expressions, S-Expression Lists, Atoms

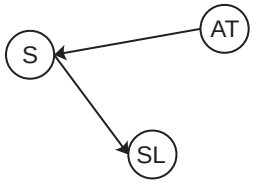
[4] Here we assume that the lexer is tokenizing the input and that the identity of individual identifiers is merged into the token *id*.



In order to use this machine to build the parse table for this grammar, we must first compute the first and follow sets for the grammar. The precedence graph for First is:



While the precedence graph for Follow is:



That these two graphs are inverses of one another is a coincedence.

Now, the values for InitFirst are as follows:

$$\begin{aligned} \text{InitFirst}(S) &= \{(\} \\ \text{InitFirst}(SL) &= \{\} \\ \text{InitFirst}(AT) &= \{(, id\} \end{aligned}$$

Then, using the precedence graph method, we find that:

$$\text{First}(S) = \text{InitFirst}(S) \cup \text{InitFirst}(AT) = \{(\} \cup \{(, id\} = \{(, id\}$$

$$\begin{aligned}
\text{First}(SL) &= \text{InitFirst}(SL) \cup \text{InitFirst}(S) \cup \text{InitFirst}(AT) \\
&= \{\} \cup \{(\} \cup \{(, id\} = \{(, id\} \\
\text{First}(AT) &= \text{InitFirst}(AT) = \{(, id\}
\end{aligned}$$

Next, we compute InitFollow, using the grammar and the values of First:

$$\begin{aligned}
\text{InitFollow}(S) &= \{\$\} \cup \text{First}(SL) = \{\$\} \cup \{(, id\} = \{\$, (, id\} \\
\text{InitFollow}(SL) &= \{)\} \\
\text{InitFollow}(AT) &= \{\}
\end{aligned}$$

Finally, using the precedence graph, we get the following values for Follow:

$$\begin{aligned}
\text{Follow}(S) &= \text{InitFollow}(S) \cup \text{InitFollow}(SL) = \{\$, (, id\} \cup \{)\} = \{\$, (,), id\} \\
\text{Follow}(SL) &= \text{InitFollow}(SL) = \{)\} \\
\text{Follow}(AT) &= \text{InitFollow}(AT) \cup \text{InitFollow}(S) \cup \text{InitFollow}(SL) \\
&= \{\} \cup \{)\} \cup \{\$, (, id\} = \{\$, (,), id\}
\end{aligned}$$

All this data then leads to the following parse table:

State	Action				Goto		
	(	)	id	$	S	SL	AT
1	S2		S9		10		4
2	S2	S5	S9		3	6	4
3	S2	R3	S9		3	8	4
4	R1	R1	R1	R1			
5	R5	R5	R5	R5			
6		S7					
7	R2	R2	R2	R2			
8		R4					
9	R6	R6	R6	R6			
10				acc			

Using the SLR(1) Parse Table

The algorithm for making use of an LR(1) or SLR(1) parse table to process an input string is called the LR Shift/Reduce Algorithm. The algorithm makes use of a stack which holds parser state numbers, and proceeds as follows:

1. Initialize the stack and push state 1 onto it.
2. Position the input pointer so that it points to the first token in the input stream.
3. Let I be the state on the top of the stack and let a be the terminal symbol currently pointed to by the input pointer. Lookup the entry for action(I,a) in the parse table.
   (a) If the action is Shift(J) then:
      i. Push state J onto the stack,
      ii. Advance the input pointer to the next token, and
      iii. Return to step 3.
   (b) If the action is Reduce($A \rightarrow \alpha$) (or Reduce(r), where grammar rule r is $A \rightarrow \alpha$) where A is some non-terminal symbol and α is some string of terminals and non-terminals n symbols long, then:
      i. Remove the top n symbols from the stack,
      ii. Let J be the state now on the top of the stack. Lookup goto(J,A) and push that state onto the stack, and
      iii. Return to step 3 (without advancing the input pointer).
   (c) If the action is **accept** then accept.
   (d) Otherwise, reject.

An Example Grammar and The Corresponding LR-1 Parse Table

A simplified version of the s-expression grammar used for the course projects:

$$
\begin{array}{rcll}
N & = & \{ & S, SL, AT \ \} \\
\Sigma & = & \{ & (,), id \ \} \\
P & = & \{ & 1 : S \rightarrow AT, \\
 & & & 2 : S \rightarrow (\ SL\ ), \\
 & & & 3 : SL \rightarrow S, \\
 & & & 4 : SL \rightarrow S\ SL, \\
 & & & 5 : AT \rightarrow (\ ), \\
 & & & 6 : AT \rightarrow id \ \} \\
S & = & & S
\end{array}
$$

State	Action				Goto		
	(	)	id	$	S	SL	AT
1	S2		S9		10		4
2	S2	S5	S9		3	6	4
3	S2	R3	S9		3	8	4
4	R1	R1	R1	R1			
5	R5	R5	R5	R5			
6		S7					
7	R2	R2	R2	R2			
8		R4					
9	R6	R6	R6	R6			
10				acc			