CS445: Compiler Design

Compiler Block Diagram

Source code → scanner → tokens → parser → parse tree → semantic analyzer → annotated tree → optimizer → intermediate representation (IR) → code generation → target language

Lex, Flex, Yacc, and Bison

In yacc, what is the difference between a shift/reduce conflict and reduce/reduce conflict?
A reduce/reduce conflict occurs when we have a set of things that can be reduced to different nonterminals (for example x or y). We don't know which terminal to reduce to. In a shift/reduce, we can choose to reduce everything on the RHS to the LHS now, or we can shift something else onto the symbol stack and move on to a later reduction. Dangling else is a shift/reduce conflict.

Definition of Grammar

A given grammar generates the language (a programming language in our case). The language is a set of programs L = {P_0, P_1, …, P_n} that are generated by the grammar. A recognizer has to recognize each program as being a member of the language.

Ambiguity in Grammars

Given a grammar where number contains all integers:

exp → exp op exp | (exp) | number
op → + | - | *

Show that a legal derivation exists for the sentential form “(34 - 3) * 42” using the productions in the grammar. A sentential form is any valid member of the language generated by the grammar. Notice the grammar is ambiguous as we can substitute for either exp in the RHS exp → exp op exp.

The legal derivation:

exp → exp op exp
exp → exp op number
exp → exp * number
exp → (exp) * number
exp → (exp op exp) * number
exp → (exp op number) * number
exp → (exp - number) * number
exp → (number - number) * number
exp → (number - number) * number

The derivation shows “(34 - 3) * 42” is a legal member of the language generated by the grammar. In this example, we wrote the rightmost derivation as we always choose to substitute the RHS of exp. We call this an LALR(1) derivation.

Now, show the leftmost derivation for “(34 - 3) * 42.”

exp → exp op exp
exp → (exp) op exp
exp → (exp op exp) op exp
exp → (number op exp) op exp
exp → (number - exp) op exp
exp → (number - number) op exp
exp → (number - number) * exp
exp → (number - number) * number

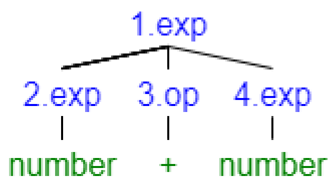
This leftmost derivation shows “(34 - 3) * 42” is a legal member of the language generated by the grammar.

We can write a leftmost derivation for “4 + 7” using the same grammar:

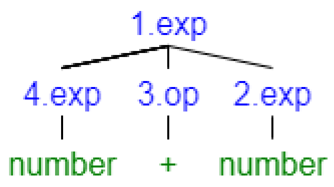
1. exp → exp op exp

2. exp → number op exp
3. exp → number + exp
4. exp → number + number

The corresponding parse tree for this derivation is:



We traverse the above tree in the order of the leftmost derivation, called preorder traversal. We can traverse the same tree in reverse postorder, as shown below. This corresponds to the rightmost derivation.



In compilers, we use abstract syntax trees (ASTs) to determine if the sentential form is a member of the grammar. Abstract syntax trees remove unnecessary terminals and improve efficiency. A parse tree has "everything" meaning all nonterminals that appear in the grammar must appear in the parse tree and the leaves in the tree must be terminals. In an AST, we remove the unnecessary nonterminals—those which don't provide additional information. This saves memory and decreases tree traversal time.

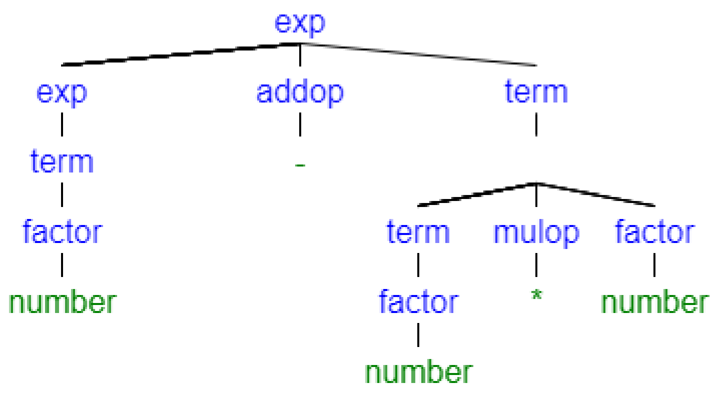
We want to codify precedence in our op symbols, following the PEMDAS concept from math. The original grammar was:

exp → exp op exp | (exp) | number
op → + | - | *
We can convert it to give multiplication precedence over addition/subtraction:
exp → exp addop exp | term
addop → + | -
term → term mulop term | factor
mulop → *
factor → (exp) | number

In this grammar, mulop is lower than addop and will appear lower in the parse tree. This is called a precedence cascade. We can further add associativity to the language. In this case, it is left associative because we write exp on the left in the first production. This grows the tree down to the right making it left recursive.

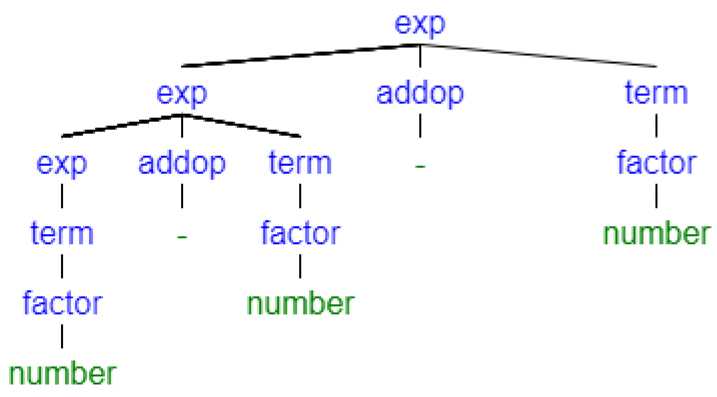
exp → exp addop term | term
addop → + | -
term → term mulop factor | factor
mulop → *
factor → (exp) | number

We can construct a parse tree for the sentential form “34 - 3 * 42.” Is it a valid sentential form?



Clearly, the sentential form is generated by the grammar. In order to resolve the second level of the tree (exp addop term), we must first go into the higher precedence (appearing lower in the grammar and parse tree) mulop.

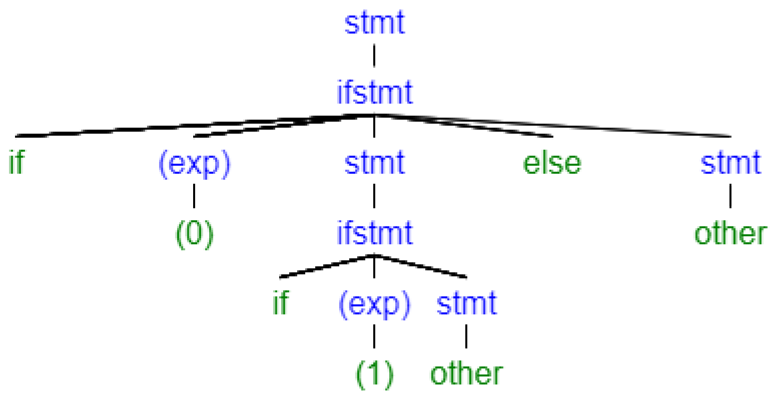
Now, do the same for the sentential form “34 - 3 - 42.”

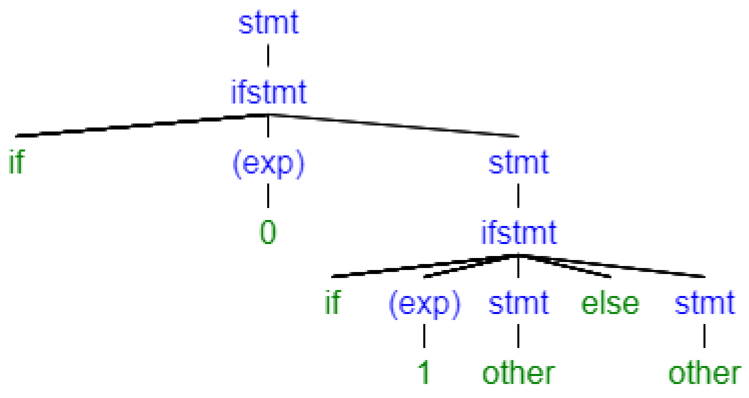


It is clear there is left associativity, meaning the tree “grows” to the left, meaning we need to resolve the left side before the right.

Here we have a grammar for simple if statements:
stmt → ifstmt | other
ifstmt → if (exp) stmt | if (exp) stmt else stmt
exp → 0 | 1

If we have the sentential form “if (0) if (1) other else other” it is ambiguous as we can derive two different parse trees that show the sentential form is a member of the language generated by the grammar. The two valid parse trees are:

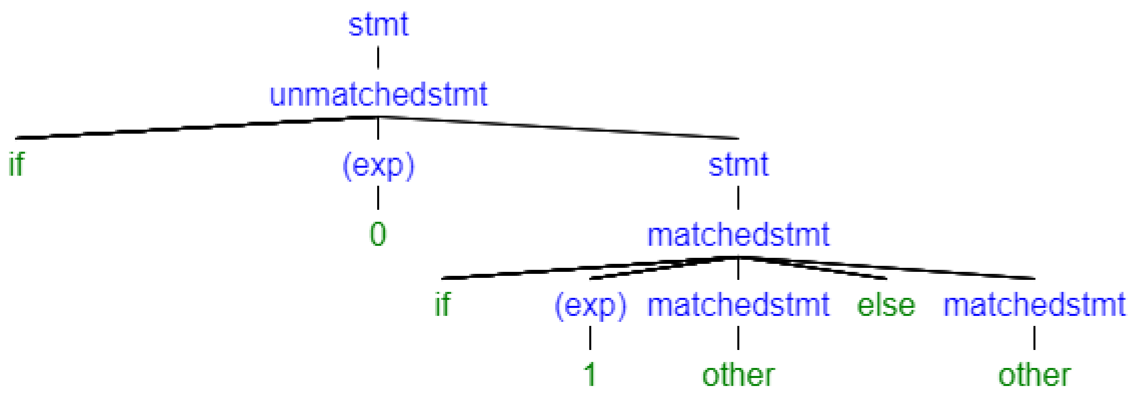




We want to enforce a rule where the else is matched to the closest unmatched if. This solves the dangling else ambiguity.

stmt → matchedstmt | unmatchedstmt
matchedstmt → if (exp) matchedstmt else matchedstmt | other
unmatchedstmt → if (exp) stmt | if (exp) matchedstmt else unmatchedstmt
exp → 0 | 1

We can prove “if (0) if (1) other else other” is a sentential form for the language generated by the new grammar.



There is no other valid parse tree that can be constructed for the sentential form. The grammar is not ambiguous. In programming languages, we often use {} to disambiguate the dangling else.

An unambiguous grammar for statements separated by semicolons is:
stmtseq → stmt ; stmtseq | stmt
stmt → s
The grammar is made ambiguous with the change:
stmtseq → stmtseq ; stmtseq | stmt
stmt → s

However, this is an unessential ambiguity as it does not have a meaningful impact on the language. The two grammars still generate the same things even if the second grammar is ambiguous.

Top-down Parsing

In top-down parsing, we traverse from the root of the tree downward. We can divide top-down parsers into two classes: predictive and backtracking. In predictive top-down parsers, we consider the current state and input to predict the construct being parsed at that time. In backtracking, we are able to backtrack to a saved state if errors are encountered. We save states in a stack. Top-down parsers have an implicit preorder traversal. We will study two top-down parsers in the predictive class: recursive descent and LL(1).

Note that the first letter in LL(1) indicates how the input is considered. In LL(1), it is left-to-right. The second letter indicates whether we are doing a leftmost or rightmost derivation (called reduction). In LL(1), we do leftmost derivation. The number 1 indicates how many tokens we consider at most before making our prediction.

Recursive Descent Parsers

Consider the below grammar.

exp → exp addop term | term
addop → + | -
term → term mulop factor | factor
mulop → *
factor → (exp) | number

In a recursive descent parser, we consider a production as being a rule (or programmatic function) that specifies how to “write the code” to recognize the left hand side.
For a factor, we may write the pseudocode:

```
procedure factor();
begin
    case token of
    ‘(‘:
        match(‘(‘);
        exp();
        match(‘)’);
    number:
        match(number);
    else error;
    end case;
end factor;
```

If it is a factor, the token must start with ‘(‘ as in (exp) or number. We write pseudocode for match as follows:

```
procedure match(expTok);
begin[
    if (token = expTok) then
        getToken();
    else
        error();
    endif
end match;
```

In match, we have an error if the next input is not the correct token. Otherwise, it is correct, and we consume the token.
If we were to write a procedure for exp, it would call term which would call factor which would call exp (recursive).

ifstmt → if (exp) stmt | if (exp) stmt else stmt

In the above simple grammar, both the productions have the same “prefix,” that is “if (exp) stmt.” If we see an “else,” we can assume we still need to parse the “if (exp) stmt else stmt” RHS. Otherwise, we assume we have successfully parsed to the first production. We can write pseudocode for this:

```
procedure ifstmt()
begin
        match(if);
        match(‘(‘);
        exp();
        match(‘)’);
        stmt();
        if (token = else) then
                match(else);
                stmt();
        endif;
end ifstmt;
```

In the previously discussed grammar,

exp → exp addop term | term
addop → + | -
term → term mulop factor | factor
mulop → *
factor → (exp) | number

we encounter an issue when creating a recursive descent parser. We must define an exp() function to parse the exp variable on the RHS of the first production. However, the LHS of the production also contains the exp variable. Therefore, exp is left recursive in the production. Partially converting the BNF form to EBNF allows us to solve this “chicken or the egg” problem.

The RHS of the first production in the new EBNF production represents a term followed by 0 or more addop coupled with a term.

exp → term {addop term}

We then write the pseudocode for exp().

```
procedure exp()
begin
        term();
        while (token = + or token = ‘-’) do
                match(token);
                term();
        end while
end exp;

procedure term()
begin
        factor();
        while (token = ‘*’) do
                match(token);
        end while
end term
```

How do we preserve the left associativity of + and -? We can rewrite exp().

```
procedure exp() : return integer
begin
        var tmp : integer;
        tmp := term();
        while(token = ‘t’ or token = ‘-’) do
                case token of
                        ‘+’:
                                match(‘+’);
                                tmp := tmp + term();
                        ‘-’:
                                match(‘-’);
                                tmp := tmp - term();
                end case;
        end while;
        return tmp;
end exp;
```

Let’s say we have a grammar with many finite terminals or variables.

A → a | b | … | z

We need to unambiguously determine which RHS our LHS is to be reduced to at a given point in parsing. This requires computations of “first sets.” We must examine all possible terminals that may match the string in question: First(a), First(b), …, First(z) where none are subsets of each other.

If these notes on first sets don’t make sense to you it’s because Wilder spent all of 3 minutes explaining the concept with the bulk of that being this quote:

"This requires computation of what's called the first sets of each token or of each nonterminal the first set of each nonterminal that we have to be able to be looking at to determine which right hand hand side which nonterminal thing it is that we're reducing the left hand to."

LL(1) Grammars and Their Parsers

Consider the grammar:

S → (S)S | ε

It is of the form Dyck as the parentheses are matched. Recursive descent LL(1) parsing requires a stack. We use the stack instead of recursive calls to procedures. In our stack the "$" character represents the bottom of the stack or end of input.

Is the sentential form "()" a member of the language generated by the grammar?
In LL(1) parsing, we either consume input or perform a reduction. When we consume input we call it a match like in the previously written pseudocode for a match() procedure.

Stack	Input	Action	Notes
$S	()$	S → (S)S	Reduction; input doesn't change
$S)S(	()$	match "("	We "look ahead" to the next symbol in input which is "(" and match it.
$S)S	)$	S → ε	Reduce the S to ε as we can't match ")"
$S)	)$	match ")"	
$S	$	S → ε	We still have S in the stack; we can reduce it to ε
$	$		We accept the sentential form through LL(1) parsing

Generalized, a top-down LL(1) parser looks like:

```
$startsymbol   inputstring$
.
. sequence of reductions and/or matches
.
$              $              string accepted
```

At any point when parsing, we can do one of two things:

1. Replace the nonterminal at the top of the symbol stack with a string (nonterminals or terminals) using the relevant production. This is called a reduce (also known as a generate).

2. Match a token on the top of the symbol stack with the next token of the input string.

Perform a leftmost reduction to prove the sentential form “()” is a member of the language generated by S → (S)S | ε.

S → (S)S	[S → (S)S]
→ ()S	[S → ε]
→ ()	[S → ε]

You can see the reductions in square brackets are the same as those in the “Action” column of the LL(1) table (ignoring match actions). This shows we performed a leftmost reduction in the LL(1) table.

We must make a choice when we have a nonterminal at the top of the parsing stack: we can choose any production. However, we must attempt a match if the top of the stack is a terminal. There is an error if we can’t do a match.

We can create a more formal lookup table for LL(1) parsing called the M[N, T] table. Note: M = machine, N = nonterminals, T = terminals. The table shows us which production to use for every nonterminal coupled with a terminal.
There are rules for the table:

- If A → α is a production in the grammar and there exists a derivation that goes from α through some sequence of derivations to aβ where a is a token (not a nonterminal), we add A → α to the table at entry M[A, a].
- If A → α is a production in the grammar and there exists derivations where α → ε and S$ → BAα (see below note) where S is the start symbol of the grammar and a is a token (or $), we add A → α to the table at entry M[A, a].

Note: he wrote this symbol here he wrote another symbol here . I think it’s a gamma but was nowhere else in the lecture, so I don’t know what it represents.

In the LL(1) lookup table (called a LUT), each row represents a nonterminal and each column a terminal. Each entry specifies the production that should be used to perform a reduction for the nonterminal in the M[N, T] column and terminal in the left side of the input string.

For the previous example for the grammar S → (S)S | ε and sentential form “(),” the table appears as follows:

M[N, T]	(	)	$
S	S → (S)S	S → ε	S → ε

The definition of an LL(1) grammar from the textbook:

A grammar is ::(1) if the associated parse table has at most one production in every entry in the matrix. If we have two or more, it is not LL(1) as it would be ambiguous and the parsing cannot deterministically predict the correct production to use and we cannot backtrack.

We can write a pseudocode procedure for LL(1) parsing.

```
push start symbol onto parsing stack
while top of stack != $ do
    if top of stack is a terminal a and next token in input = a then
        pop a off parsing stack (a match has occurred)
        advance input to next token
    else if top of stack is a nonterminal A and next input is a token a or $ and parsing table
    entry M[A, a] contains production A → $x_1$ $x_2$ … $x_n$ then
        Pop parsing stack for i := n down to 1 do
            push $x_i$ onto parsing stack
    else
        error
    endif

    if top of stack = $ and input = $
        accept the input
    else
        error
    endif
endwhile
```

Parsing Sentential Forms Using Parse Tables

Another example of constructing an LL(1) parse table:

stmt → ifstmt | other
ifstmt → if (exp) stmt elsepart
elsepart → else stmt | ε
exp → 0 | 1

M[N, T]	if	other	else	0	1	$
stmt	stmt → ifstmt	stmt → other	error	error	error	error
ifstmt	ifstmt → if (exp) stmt elsepart	error	error	error	error	error

elsepart	error	error	elsepart → else stmt elsepart → ε	error	error	elsepart → ε
exp	error	error	error	exp → 0	exp → 1	error

We have an ambiguity as there are two productions in an entry (the dangling else). So, we do not have a valid LL(1) parse tree.

An example of parsing the dangling else is shown in the sentential form “if (0) if (1) other else other.”

stack	input	action
$stmt	if (0) if (1) other else other$	stmt → ifstmt
$ifstmt	if (0) if (1) other else other$	ifstmt → if (exp) stmt elsepart
$elsepart stmt)exp(if	if (0) if (1) other else other$	match “if”
$elsepart stmt)exp(	(0) if (1) other else other$	match “(“
$elsepart stmt)exp	0) if (1) other else other$	exp → 0
$elsepart stmt)0	0) if (1) other else other$	match “0”
$elsepart stmt)	) if (1) other else other$	match “)“
$elsepart stmt	if (1) other else other$	stmt → ifstmt
$elsepart ifstmt	if (1) other else other$	ifstmt → if (exp) stmt elsepart
$elsepart elsepart stmt)exp(if	if (1) other else other$	match “if”
$elsepart elsepart stmt)exp(	(1) other else other$	match “(“
$elsepart elsepart stmt)exp	1) other else other$	exp → 1
$elsepart elsepart stmt)1	1) other else other$	match “1”
$elsepart elsepart stmt)	) other else other$	match “)”
$elsepart elsepart stmt	other else other$	stmt → other
$elsepart elsepart other	other else other$	match “other”

$elsepart elsepart	else other$	elsepart → else stmt (ambiguous; we could choose elsepart → ε)
$elsepart stmt else	else other$	match “else”
$elsepart stmt	other$	stmt → other
$elsepart other	other$	match “other”
$elsepart	$	elsepart → ε
$	$	accept

Clearly, “if (0) if (1) other else other” is a member of the language generated by the grammar.

Not all languages that we want to use work with LL(1) grammars. However, we can modify some grammars to work with LL(1) parsing. One method is left recursion removal.

Left Recursion Removal in Grammars

We want to remove left recursion to remove ambiguity and conform to LL(1) parsing requirements.

exp → exp addop term | term

The above grammar is left recursive meaning the parse tree will grow to the left for a given sentential form. It is also left associative and the left recursion is immediate. If we switched “exp” with “term” in the RHS it would be right recursive and right associative.

We can remove the left recursion and maintain left associativity to allow LL(1) parsing. This gets more complicated when left recursion is not immediate.

For example, the below grammar requires a substitution before left recursion is apparent. It has general immediate left recursion.

A → Bb | …
B → Aa | …

Consider the following grammar with simple immediate left recursion.

exp → exp addop term | term
It is of the form “A → Aα | β” and we can rewrite it as:
A → βA’

A’ → αA’ | ε

This form no longer has left recursion.

Here is what this looks like in the exp grammar:

exp → term exp’
exp’ → addop term exp’ | ε

Below is a grammar with general immediate left recursion.

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid B_1 \mid B_2 \mid \ldots \mid B_n$

We can reduce it to the below grammar.

$A \rightarrow B_1A' \mid B_2A' \mid \ldots \mid B_nA'$
$A' \rightarrow \alpha_1A' \mid \alpha_2A' \mid \ldots \mid \alpha_nA'$

We can do the same with the below grammar.

exp → exp + term | expterm | term
We rewrite it as follows.
exp → term exp’
exp’ → + term exp’ | - term exp’ | ε

This change does not modify the language, just the grammar. Left associativity is maintained. We can now use LL(1) parsing for the grammar.

Left Factoring in Grammars

Left factoring is required when we have two or more grammar choices that begin with the same prefix. Consider the general form of a grammar below.

A → αβ | αρ

There is a common prefix “α.” We need to correctly choose which production to use during parsing. We can fix this by factoring the grammar to:

A → αA’
A’ → β | ρ

Now, the prefix is only present in one production. We must capture all of the commonality on the left for this to work.

A more complex example of a grammar with this issue is:

stmtseq → stmt; stmtseq | stmt
stmt → s

Each production has the “s” prefix. We factor the grammar to remove the ambiguity.

stmtseq → stmt stmtseq’
stmtseq’ → ; stmtseq | ε

Let’s now attempt to factor the below grammar.

ifstmt → if (exp) stmt | if (exp) stmt else stmt

Here we have the prefix “if (exp) stmt.” In a recursive descent parser each production is a “rule” or procedure. We need to perform left factoring to ensure the correct choice is made during LL(1) parsing. The left factored version of the grammar is:

ifstmt → if (exp) stmt elsepart
elsepart → else stmt | ε

Below is another case where the grammar does not comply with the requirements of LL(1) parsing.

stmt → assignstmt | callstmt | other
assignstmt → identifier := exp
callstmt → identifier(explist)

Here, identifier is a common prefix for an assign and call statement. Before left factoring, we first bring assignstmt and callstmt into a single production.

stmt → identifier := exp | identifier(explist) | other

Then, we left factor.

stmt → identifier stmt’ | other
stmt’ → := exp | (explist)

The grammar now satisfies the requirements of LL(1) parsers.

Computing First and Follow Sets of Grammars

LL(1) parsing falls under the category of predictive parsing. When doing predictive parsing, we cannot backtrack. This means we cannot make mistakes when determining which production is used for parsing at a given time. Given at most one token in the input string we must determine which production to use for reduction or matching. Computing the first sets for a grammar is helpful when doing so. The First(x) function is what we use to create the parse table for LL(1). Generally, we perform left factoring and left recursion removal before calculating the first sets.

First(x) is computed as follows:

1. If x is a terminal or ε then First(x) = {x}.
2. If x is a nonterminal then for each production $x \rightarrow x_1\ x_2 \ldots x_n$ (any combination of terminals and nonterminals), First(x) contains First(x_1) - {ε}. If for some $i < n$ all sets First(x_1) … First(x_i) contains ε, then First(x) contains First(x_{i+1}) - {ε}. If all sets First(x_i) … First(x_n) contains ε then First(x) also contains ε.

Define F(x) for a sequence of terminals and nonterminals $\alpha = x_1\ x_2 \ldots x_n$ as:
First(α) contains First(x_1) - {ε}. For each i = 2 … n, if First(x_k) contains ε for all k = 1 … i - 1, then First(α) contains First(x_i) - {ε} and if for all i = 1 … n, First(x_i) contains ε, then First(α) contains ε.

We can write pseudocode for computing First(x).

```
for all nonterminals A do
    First(A) := {}
while any changes to any First(A) do
    for each production A → x_1 x_2 … x_n do
        k := 1
        continue := true
        while (continue = true) and k <= n do
            add First(x_k) - {ε} to First(A)
            if ε is not a member of First(A) then
                continue := true
            k := k + 1
            if (continue = true) then
                add {ε} to First(A)
```

Consider the below grammar.

exp → exp addop term | term
addop → + | -
term → term mulop factor | factor
mulop → *
factor → (exp) | number

When computing the first sets, we first break up the productions into individual lines:

exp → exp addop term
exp → term

addop → +
addop → -
term → term mulop factor
term → factor
mulop → *
factor → (exp)
factor → number

The goal is to have a set of possible RHS terminals and nonterminals for a given LHS. We have an error if there is an empty first set for the LHS provided.

The steps for computing the first sets for the above grammar are shown below. We begin with empty sets and continue to loop as long as there are changes to any first set for any nonterminal in the grammar. We cannot calculate the first set when there is recursion in the production or the first part of the RHS references an empty first set. When we reach the last column, no more changes to the first sets occur, and we are done computing the first sets.

First(exp) = {}	First(exp) = {}	First(exp) = {}	First(exp) = {(, number}
First(addop) = {}	First(addop) = {+, -}	First(addop) = {+, -}	First(addop) = {+, -}
First(term) = {}	First(term) = {}	First(term) = {(, number}	First(term) = {(, number}
First(mulop) = {}	First(mulop) = {*}	First(mulop) = {*}	First(mulop) = {*}
First(factor) = {}	First(factor) = {(, number}	First(factor) = {(, number}	First(factor) = {(, number}

A nullable nonterminal can be reduced to ε through a series of replacements. In the below grammar, "elsepart" is nullable.

stmt → ifstmt | other
Ifstmt → if (exp) stmt elsepart
elsepart → else stmt | ε
exp → 0 | 1

We can transfer the above grammar into individual lines:

stmt → ifstmt
stmt → other
Ifstmt → if (exp) stmt elsepart
elsepart → else stmt
elsepart → ε

exp → 0
exp → 1

First(stmt) = {}	First(stmt) = {other}	First(stmt) = {other, if}
First(ifstmt) = {}	First(ifstmt) = {if}	First(ifstmt) = {if}
First(elsepart) = {}	First(elsepart) = {else, ε}	First(elsepart) = {else, ε}
First(exp) = {}	First(exp) = {0, 1}	First(exp) = {0, 1}

Another grammar is:

stmtseq → stmt stmtseq’
stmtseq’ → ; stmtseq | ε
stmt → s

First, we write each production in its own line

stmtseq → stmt stmtseq’
stmtseq’ → ; stmtseq
stmtseq’ → ε
stmt → s

First(stmtseq) = {}	First(stmtseq) = {}	First(stmtseq) = {s}
First(stmtseq’) = {}	First(stmtseq’) = {;, ε}	First(stmtseq’) = {;, ε}
First(stmt) = {}	First(stmt) = {s}	First(stmt) = {s}

Given a nonterminal A, the set Follow(A) is composed of terminals (and possibly $) and is defined as follows:

1. If A is the start symbol $ ∈ Follow(A)
2. If there exists a production B → αAγ then First(γ) - {ε} ∈ Follow(A)
3. If there exists a production B → αAγ such that ε is in First(γ) then Follow(B) ∈ Follow(A)

Note: ε is never a member of a follow set, only first sets.

We can write pseudocode for computing the follow sets:

```
Follow(startsymbol) := {$}
for all nonterminals A != startsymbol do
    Follow(A) := {}
while any changes to any follow sets do
    for each production A → $x_1$ $x_2$ … $x_n$ do
        for each $x_i$ that is a nonterminal do
        add First($x_{i+1}$ $x_{i+2}$ … $x_n$) - {ε} to Follow($x_i$)
        if ε is in First($x_{i+1}$ $x_{i+2}$ … $x_n$) then
            add Follow(A) to Follow($x_i$)
```

Calculate the follow sets for the previously used example grammar and its first sets.

stmt → ifstmt
stmt → other
Ifstmt → if (exp) stmt elsepart
elsepart → else stmt
elsepart → ε
exp → 0
exp → 1

First(stmt) = {other, if}
First(ifstmt) = {if}
First(elsepart) = {else, ε}
First(exp) = {0, 1}

Follow(stmt) = {$, else}	Follow(stmt) = {$, else}
Follow(ifstmt) = {$}	Follow(ifstmt) = {$, else}
Follow(elsepart) = {$}	Follow(elsepart) = {$, else}
Follow(exp) = {)}	Follow(exp) = {)}

If the nonterminal is the last one in the RHS, use the Follow() of the LHS nonterminal.

Let’s compute the follow sets for another grammar we previously looked at.

stmtseq → stmt stmtseq’
stmtseq’ → ; stmtseq
stmtseq’ → ε
stmt → s

First(stmtseq) = {s}
First(stmtseq’) = {;, ε}
First(stmt) = {s}

Follow(stmtseq) = {$}	Follow(stmtseq) = {$}
Follow(stmtseq’) = {$}	Follow(stmtseq’) = {$}
Follow(stmt) = {;}	Follow(stmt) = {;, $}

Constructing Parse Tables for Grammars

Now that we can compute the first and follow sets, we can begin to construct parse tables. Use the instructions below to construct an LL(1) parse table for a given grammar.

For each nonterminal A and production A → α do:

1. For each token a in First(α) add A → α to entry M[A, a].
2. If ε is in First(α), for each element a of Follow(A) (it must be a token or $; not ε) add A → α to M[A, a].

A grammar in BNF is LL(1) if and only if:

1. For every production A → α_1 | α_2 | … | α_n, First(α_i) ∩ First(α_j) is empty for all i and j such that 1 <= i and j <= n and i != j.
2. For every nonterminal A such that First(A) contains ε, First(A) ∩ Follow(A) is empty.

Construct a parse table for the grammar.

exp → term exp'
exp' → addop term exp' | ε
addop → + | -
term → factor term'
term' → mulop factor term' | ε
mulop → *
factor → (exp) | number
	First(exp) = {(, number}
	First(exp') = {+, -, ε}
	First(addop) = {+, -}
	First(term) = {(, number}
	First(term') = {*, ε}
	First(mulop) = {*}
	First(factor) = {(, number}
		Follow(exp) = {$,)}
		Follow(exp') = {$,)}
		Follow(addop) = {(, number}
		Follow(term) = {$,), +, -}
		Follow(term') = {$,), +, -}
		Follow(mulop) = {(, number}
		Follow(factor) = {$,), +, -, *}

M[N, T]	(	number	)	+	-	*	$
exp	exp → term exp'	exp → term exp'	error	error	error	error	error
exp'	error	error	exp' → ε	exp' → addop term exp'	exp' → addop term exp'	error	exp' → ε
addop	error	error	error	addop → +	addop → -	error	error
term	term → factor term'	term → factor term'	error	error	error	error	error
term'	error	error	term' → ε	term' → ε	term' → ε	term' → mulop	term' → ε

						factor term’	
mulop	error	error	error	error	error	mulop → *	error
factor	factor → (exp)	factor → number	error	error	error	error	error

Bottom-up Parsers

LL(1) parsing is inherently top-down because the second L indicates a leftmost reduction. LR parsing is a bottom-up parser. We don’t necessarily need a number for the look ahead as in some cases we just need the parsing table and can consider the current state of the parsing stack without actually “looking ahead.” We will examine LR(1) where the input is considered left to right, we do rightmost reductions, and our look ahead is 1. Later we will look at SLR(1) or simplified LR(1) parsing. In this parsing method, we simplify the parsing table.

Generally, an LR(1) parse is of the following form:

Parse Stack	Input	Action
$	inputstring$	
.		
. Apply sequence of shift or reduce actions		
.		
$startsymbol	$	accept

We can perform one of two actions:

1. Shift a token from the inputstring onto the parse stack.
2. Reduce a string (of nonterminal or terminal) to a nonterminal A given a production A → α. In this action, we take α off the parse stack and put A in its place.

Bottom-up parsers are typically called shift/reduce parsers. Shift/reduce conflicts occur when it is ambiguous which action should be performed at a given point in parsing.

In bottom-up parsers, we must perform “augmentation of the grammar.” Consider the below grammar.

S → (S)S | ε

We must replace the original start rule. Augmentation is simply creating a new start production with the RHS being the LHS of the original start production.

S’ → S
S → (S)S | ε

Use LR(1) to determine if the sentential form “()” is a member of the grammar.

Stack	Input	Action
$	()$	shift “(“
$(	)$	reduce S → ε
$(S	)$	shift “)”
$(S)	$	reduce S → ε
$(S)S	$	reduce S → (S)S
$S	$	reduce S’ → S
$S’	$	accept

An example with another grammar and sentential form “n + n.” We always use the longest possible production when reducing. We must reduce if the input is “$” and there is no ε in the productions.

E’ → E
E → E+n | n

Stack	Input	Action
$	n+n$	shift “n“
$n	+n$	reduce E → n
$E	+n$	shift “+”
$E+	n$	shift “n”
$E+n	$	reduce E → E+n
$E	$	reduce E’ → E
$E’	$	accept

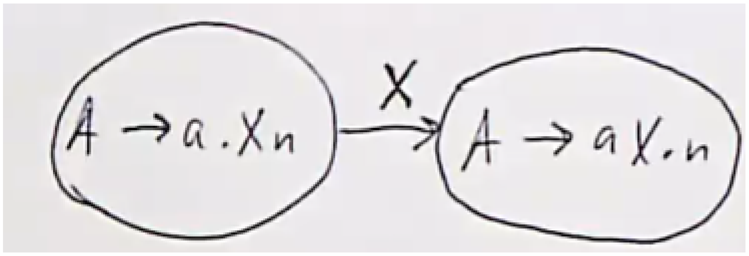
An example of creating a language to an NFA for LR(1) parsing:

S’ → S
S → (S)S | ε

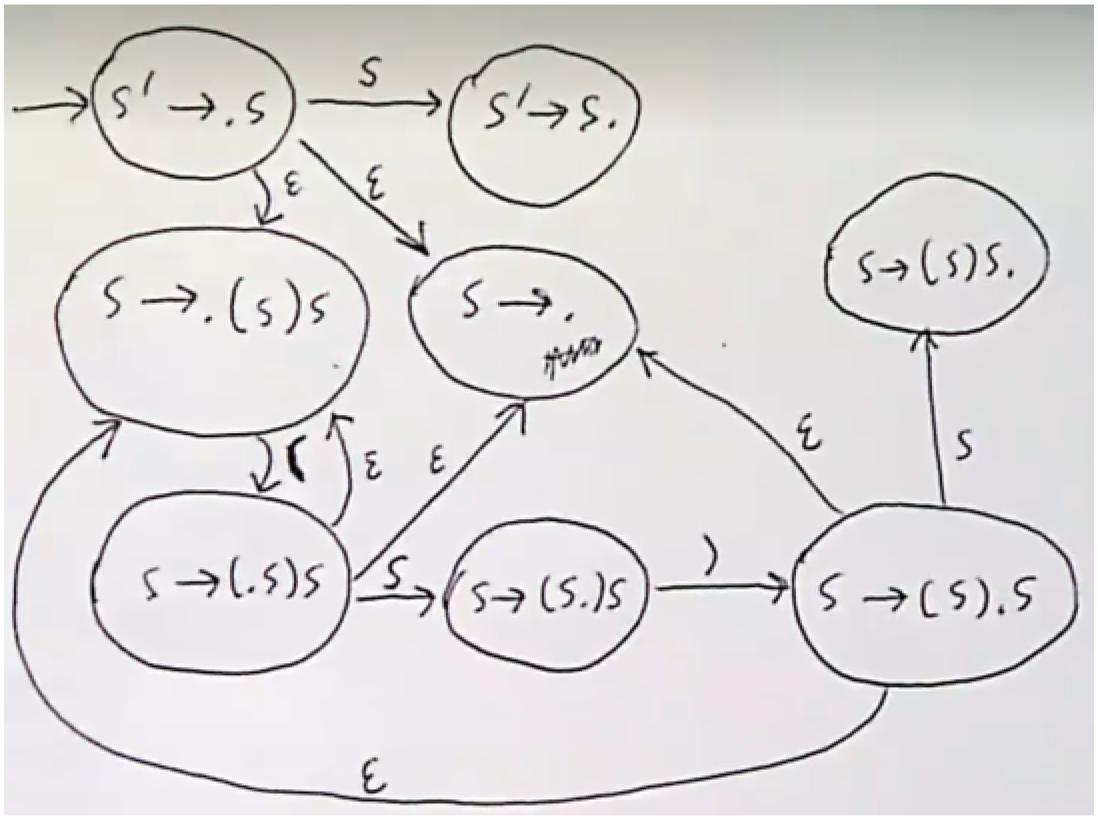
Reduce the above language to “LR items” that we use to determine the states in an NFA for the LR(1) parser. The “.” tracks your progress while reading the input string that is the RHS of a production. We write only the “.” for ε productions.

S’ → .S
S’ → S.
S → .(S)S
S → (.S)S
S → (S.)S
S → (S).S
S → (S)S.
S → .

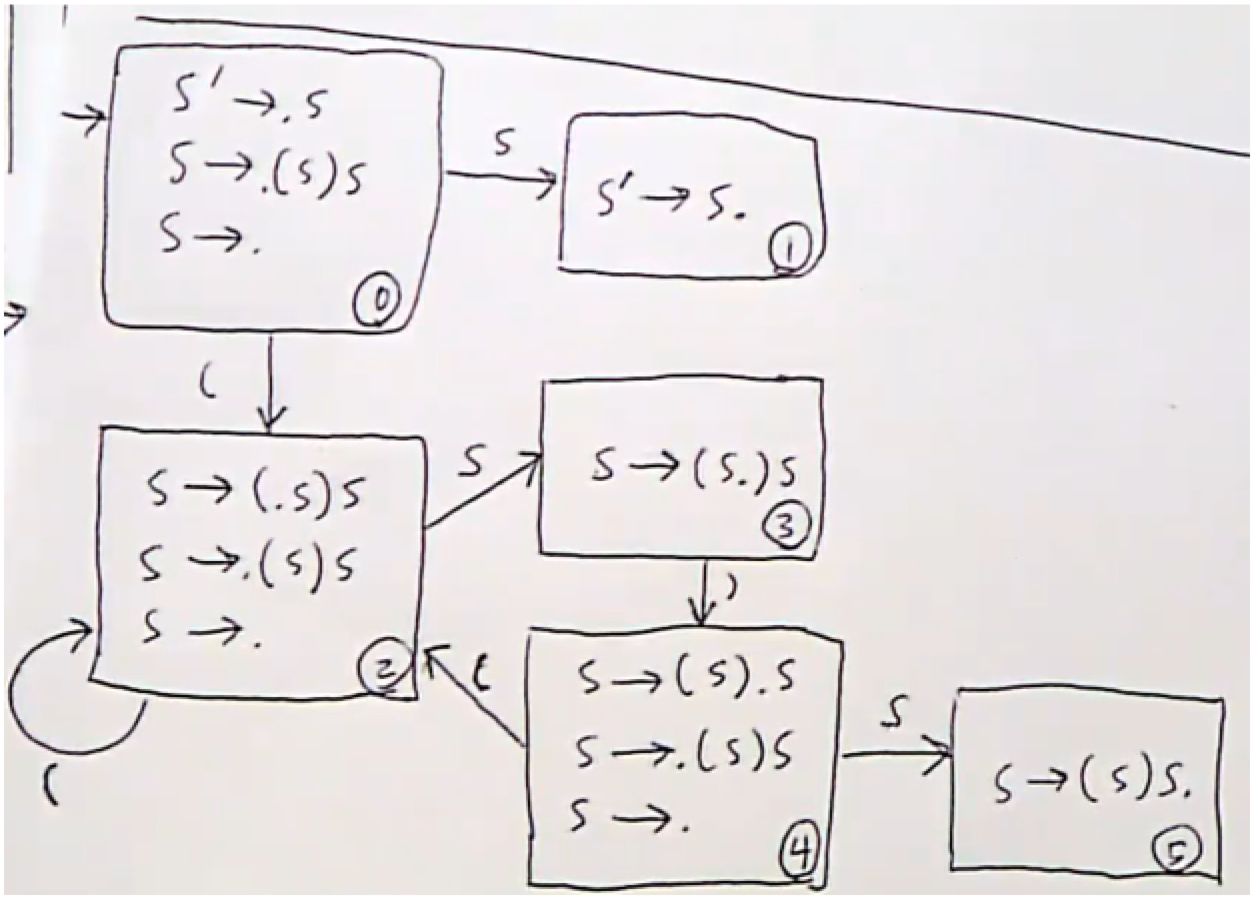
An NFA for an LR(1) grammar follows the below format. We let the parser decide the accept states and do not draw them as part of the NFA. We use ε transitions whenever a nonterminal can be reduced by a different production.



Write the NFA for the language.



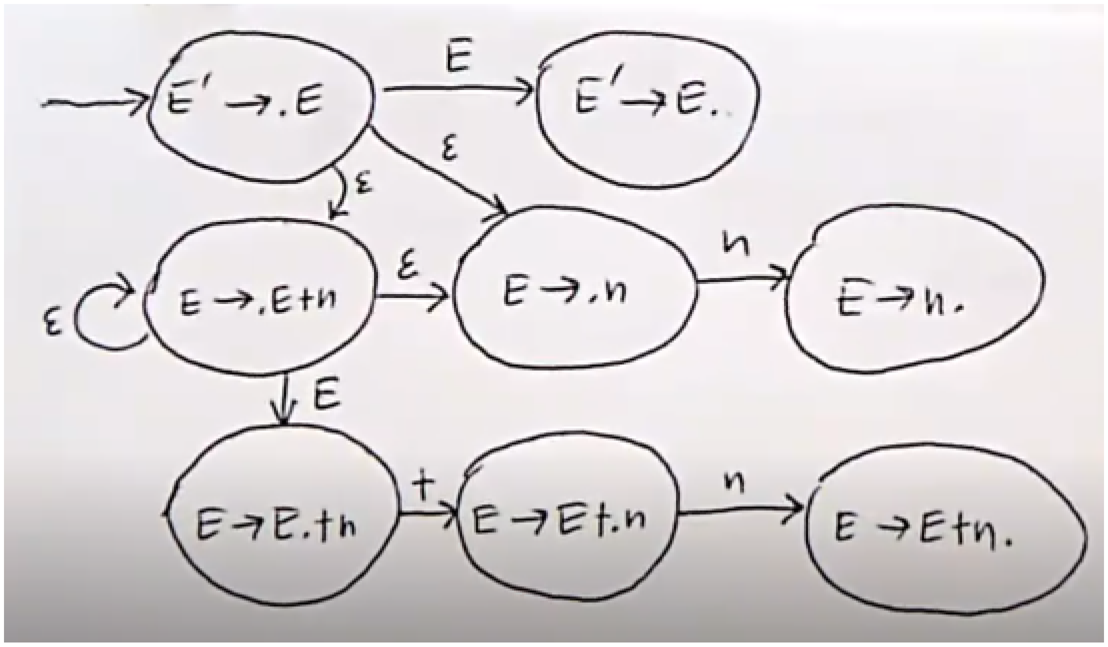
Convert it to a DFA. There are two types of items in each state: kernel items and closure items. Closure items are those productions that are reachable by a ε transition in the NFA. We often only reference kernel items when discussing the DFA.



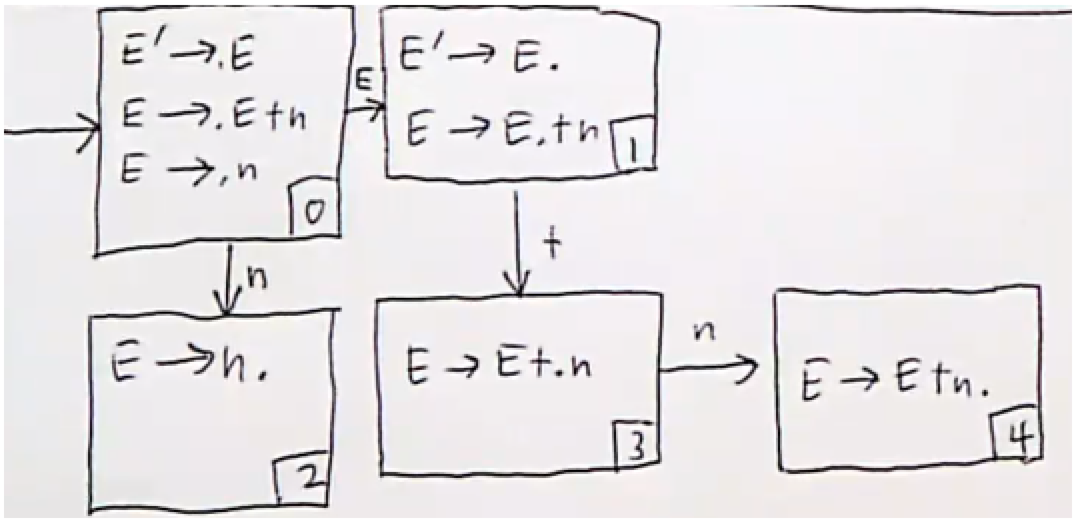
Write the NFA for the below grammar using LR(1) principles.

E’ → E
E → E+n | n

E’ → .E
E’ → E.
E → .E+n
E → E.+n
E → E+.n
E → E+n.
E → .n
E → n.



Convert to a DFA.



Now, create the LR(1) parsing table.The augmentation is always in state 0. We start the stack with “$0.” For the sentential form “nrestofstring,” the parse table looks as follows:

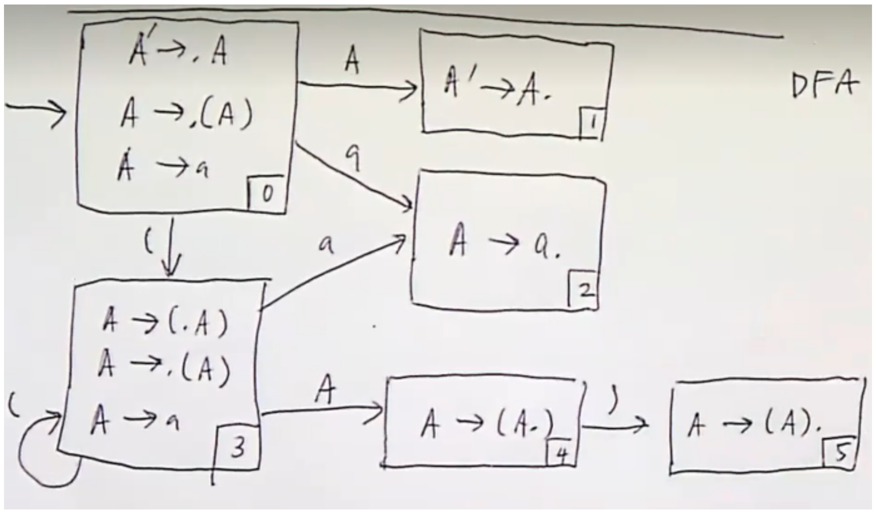
Stack	Input	Action
$0	$nrestofstring	shift “n”
$0n2	$restofstring	

In LR(0) parsing we do as follows:

1. If state S contains an item of the form A → α.xβ where x is a terminal, shift x onto the parse stack. If the terminal is indeed x, then the new state to be pushed is the state number containing A → αx.β. If it is not x, raise an error.
2. If state S contains any complete item (e.g. A → γ), then reduce by rule A → γ. If we reduce by the rule S’ → S, then enter the accept state, provided the input is empty. Error if the input is not empty.
   In all other cases, the new state of the parse is computed by:
    a. Remove string γ and all its states from the stack (backing up).
    b. Push A onto the stack.
    c. Push, as the new state, the number of the state corresponding to a production of the form B → αA.β.

Examine the simple grammar and its DFA.

A → (A) | a



Write the parse stack for the sentential form “((a)).” Note: we always want the top of the parse stack to contain the state number we are in.

Stack	Input	Action
$0	((a))$	shift “(“

$0(3	(a))$	shift “(“
$0(3(3	a))$	shift “a”
$0(3(3a2	))$	reduce A → a
$0(3(3A4	))$	shift “)”
$0(3(3A4)5	)$	reduce A → (A)
$0(3A4	)$	shift “)”
$0(3A4)5	$	reduce A → (A)
$0A1	$	reduce A’ → A
$0A’	$	accept

Write an LR(0) lookup table for the grammar. The “Input” column contains all terminals while the “Goto” column contains all nonterminals.

State	Action	Rule	Input			Goto
			(	a	)	A
0	shift		3	2	error	1
1	reduce	A’ → A				
2	reduce	A → a				
3	shift		3	2	error	4
4	shift		error	error	5	error
5	reduce	A → (A)				