Programming Languages

Janyl Jumadinova

February 15-20, 2023

Scanning and Parsing

Scanner: translate source code to tokens (e.g., $< int >, +, < id >$)

- Report lexical errors like illegal characters and illegal symbols.

Scanning and Parsing

Scanner: translate source code to tokens (e.g., $< int >, +, < id >$)

- Report lexical errors like illegal characters and illegal symbols.

Parser: read token stream and reconstruct the derivation.

- Reports parsing errors – i.e., source that is not derivable from the grammar. E.g., mismatched parenthesis/braces, nonsensical statements ($x = 1 +;$)

What is Syntax Analysis aka Parsing?

- After lexical analysis (scanning), we have a series of tokens.
- In syntax analysis (or parsing), we want to interpret what those tokens mean.

What is Syntax Analysis aka Parsing?

- After lexical analysis (scanning), we have a series of tokens.
- In syntax analysis (or parsing), we want to interpret what those tokens mean.
- **Goal**: Recover the structure described by that series of tokens.
- **Goal**: Report errors if those tokens do not properly encode a structure.

Formal Languages

- An alphabet is a set $\sum$ of symbols that act as letters.
- A language over $\sum$ is a set of strings made from symbols in $\sum$.

Formal Languages

- An alphabet is a set $\sum$ of symbols that act as letters.
- A language over $\sum$ is a set of strings made from symbols in $\sum$.
- When scanning, our alphabet is ASCII or Unicode characters. We produced tokens.

Formal Languages

- An alphabet is a set $\sum$ of symbols that act as letters.
- A language over $\sum$ is a set of strings made from symbols in $\sum$.
- When scanning, our alphabet is ASCII or Unicode characters. We produced tokens.
- When parsing, our alphabet is the set of tokens produced by the scanner.

Grammar

Grammar consists of the following::

1. a set of terminals (same as an alphabet)
2. a set of non-terminal symbols, including a starting symbol
3. a set of rules

Grammar

Grammar consists of the following::

1. a set of terminals (same as an alphabet)
2. a set of non-terminal symbols, including a starting symbol
3. a set of rules

- Strings are derived from a grammar (e.g., S $\rightarrow$ aS $\rightarrow$ aaS $\rightarrow$ aabA $\rightarrow$ aab)
- At each step, a non-terminal is replaced by the sentential form on the right-hand side of a rule (a sentential form can contain non-terminals and/or terminals)
- Grammars generate languages

Sentential Form

- If $S \rightarrow *\alpha$, the string α is called a sentential form of the grammar.
- In the derivation $S \rightarrow \beta_1 \rightarrow \beta_2 \rightarrow ... \rightarrow \beta_n$, each of the β_i are sentential forms.
- A sentential form in a rightmost derivation is called a right-sentential form (similarly for leftmost and left-sentential).

Context-Free Grammar

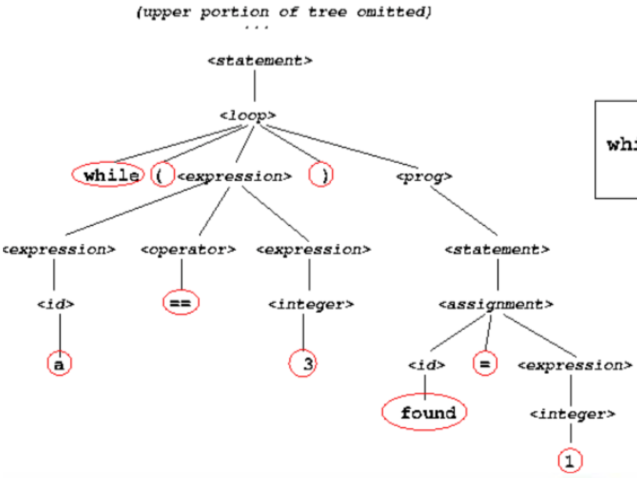
- A context-free grammar (or CFG) is a formalism for defining languages.
- A grammar is said to be context-free if every rule has a single non-terminal on the left-hand side
- This means you can apply the rule in any context.

Context-Free Grammar

Formally, a context-free grammar (as is the regular grammar) is a collection of four objects:

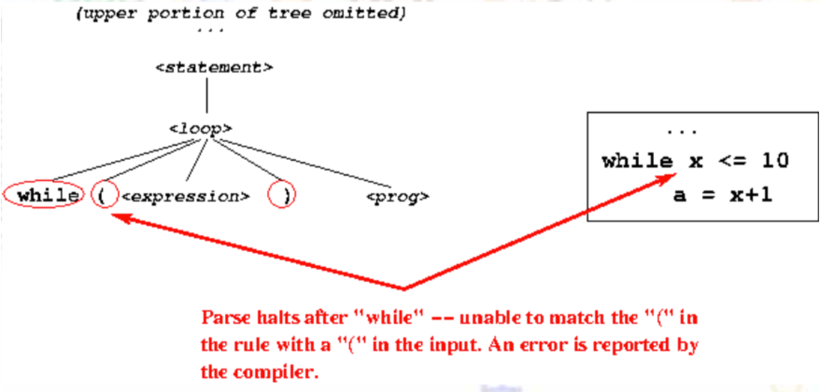
- A set of nonterminal symbols (or variables),
- A set of terminal symbols,
- A set of production rules saying how each nonterminal can be converted by a string of terminals and nonterminals, and
- A start symbol that begins the derivation.

Sample Parse Tree (portion)




```
...
while (a == 3)
    found = 1
```

Sample Parse Tree (failed)



Grammars for Java (version 8) and Python3

- Java: Overview of notation used:
  `https://docs.oracle.com/javase/specs/jls/se8/html/jls-2.html`
- Java: The full syntax grammar:
  `https://docs.oracle.com/javase/specs/jls/se8/html/jls-19.html`
- Python: The full grammar:
  `https://docs.python.org/3/reference/grammar.html`

Parsing Algorithms

LL Parsing (Left to right scan, Leftmost derivation)

- **Top-down**: start with grammar start symbol, work your way down until you get to terminals
- Generates a leftmost derivation (the leftmost derivation assuming unambiguous grammar)

Parsing Algorithms

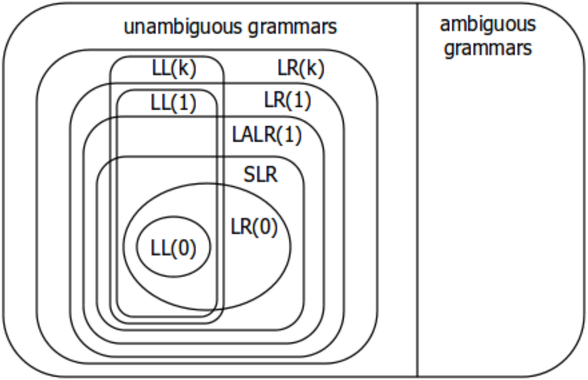
LL Parsing (Left to right scan, Leftmost derivation)

- **Top-down**: start with grammar start symbol, work your way down until you get to terminals
- Generates a leftmost derivation (the leftmost derivation assuming unambiguous grammar)

LR Parsing (Left to right scan, Rightmost derivation (reverse rightmost))

- **Bottom-up**: apply productions in reverse to convert the user's program to the start symbol
- Almost all practical programming languages have an LR(1) grammar

Language Hierarchies



Bottom-up Parsing Strategies

Beginning with the user's program, try to apply productions in reverse to convert the program back into the start symbol

Top-down Parsing Strategies

- Begin at root with a start symbol of the grammar
- Repeatedly pick a non-terminal and expand
- Success when expanded tree matches input

Top-down Parsing Strategies

- Begin at root with a start symbol of the grammar
- Repeatedly pick a non-terminal and expand
- Success when expanded tree matches input
- LL(k)

Top-down Parsing Strategies

- Begin at root with a start symbol of the grammar
- Repeatedly pick a non-terminal and expand
- Success when expanded tree matches input
- LL(k)

Beginning with the start symbol, try to guess the productions to apply to end up at the user's program.

Challenges in Top-down Parsing

- Top-down parsing begins with virtually no information.
- Begins with just the start symbol, which matches every program.
- How can we know which productions to apply?

Challenges in Top-down Parsing

- Top-down parsing begins with virtually no information.
- Begins with just the start symbol, which matches every program.
- How can we know which productions to apply?
  1. Guess and backtrack if we are wrong - **backtracking algorithms** (BFS/DFS).
  2. Based on remaining input, predict (without backtracking) which production to use - **predictive algorithms** (LL(1))

A Simple Predictive Parser: LL(1)

Top-down, predictive parsing:

- L: Left-to-right scan of the tokens
- L: Leftmost derivation
- (1): One token of lookahead

A Simple Predictive Parser: LL(1)

Top-down, predictive parsing:

- L: Left-to-right scan of the tokens
- L: Leftmost derivation
- (1): One token of lookahead
- Construct a leftmost derivation for the sequence of tokens.
- When expanding a non-terminal, we predict the production to use by looking at the next token of the input. The decision is forced.

LL(1) Algorithm: FYI

Suppose a grammar has start symbol `S` and LL(1) parsing table `T`. We want to parse string ω.

- Initialize a stack containing `S $`.
- Repeat until the stack is empty:
  - Let the next character of ω be `t`.
  - If the top of the stack is a terminal `r`:
    - If `r` and `t` don't match, report an error.
    - Otherwise consume the character `t` and pop `r` from the stack.
  - Otherwise, the top of the stack is a non-terminal `A`:
    - If `T[A,t]` is undefined, report an error.
    - Replace the top of the stack with `T[A,t]`.

Lox Interpreter Parser: Top Down

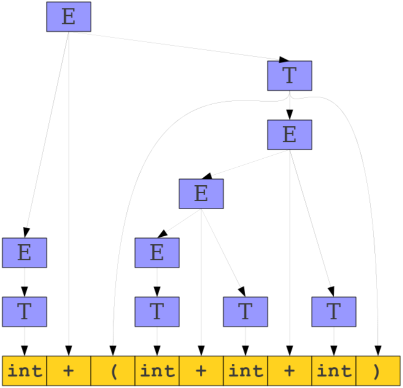
- Starts from the top or outermost grammar rule
- Goes down into the nested subexpressions
- Gets to the leaves of the syntax tree

Bottom Up Parsing

Idea: Apply productions in reverse to convert the user's program to the start symbol.

Bottom Up Parsing

$\mathbf{E} \rightarrow \mathbf{T}$
$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{T}$
$\mathbf{T} \rightarrow$ `int`
$\mathbf{T} \rightarrow (\mathbf{E})$



Bottom Up Parsing

$$E \rightarrow T$$
$$E \rightarrow E + T$$
$$T \rightarrow \texttt{int}$$
$$T \rightarrow (E)$$

$$
\begin{aligned}
& \texttt{int} + (\texttt{int} + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & T + (\texttt{int} + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & E + (\texttt{int} + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & E + (T + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & E + (E + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & E + (E + T + \texttt{int}) \\
\Rightarrow\ & E + (E + \texttt{int}) \\
\Rightarrow\ & E + (E + T) \\
\Rightarrow\ & E + (E) \\
\Rightarrow\ & E + T \\
\Rightarrow\ & E
\end{aligned}
$$

Bottom-Up Parsing

- The handle of the right-sentential form is a substring corresponding to the right-hand side of the production that produced it from the previous step in the rightmost derivation.
- Handle can also be represented as the production and the position of the last symbol being replaced.
- A left-to-right, bottom-up parse works by iteratively searching for a handle, then reducing the handle.

Shift/Reduce Algorithm

- One type of bottom-up parsers are called shift/reduce parsers.
- **Idea**: Split the input into two parts:
  - Left substring is our work area; all handles must be here.
  - Right substring is input we have not yet processed; consists purely of terminals.
- At each point, decide whether to:
  - Move a terminal across the split (**shift**)
  - Reduce a handle (**reduce**)

Shift/Reduce Algorithm

$$\mathbf{E} \rightarrow \mathbf{F}$$
$$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$$
$$\mathbf{F} \rightarrow \mathbf{F} * \mathbf{T}$$
$$\mathbf{F} \rightarrow \mathbf{T}$$
$$\mathbf{T} \rightarrow \texttt{int}$$
$$\mathbf{T} \rightarrow (\mathbf{E})$$

int	+	int	*	int	+	int

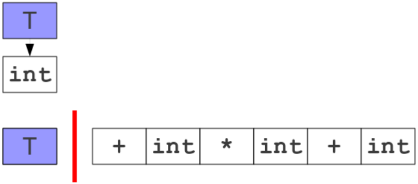
Shift/Reduce Algorithm

$\mathbf{E} \rightarrow \mathbf{F}$
$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$
$\mathbf{F} \rightarrow \mathbf{F} * \mathbf{T}$
$\mathbf{F} \rightarrow \mathbf{T}$
$\mathbf{T} \rightarrow \texttt{int}$
$\mathbf{T} \rightarrow (\mathbf{E})$

`int`	|	`+`	`int`	`*`	`int`	`+`	`int`

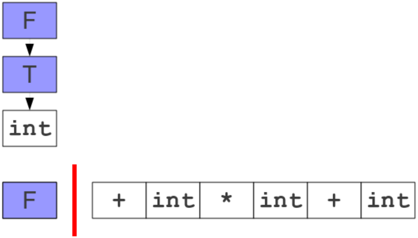
Shift/Reduce Algorithm

$E \rightarrow F$
$E \rightarrow E + F$
$F \rightarrow F * T$
$F \rightarrow T$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$



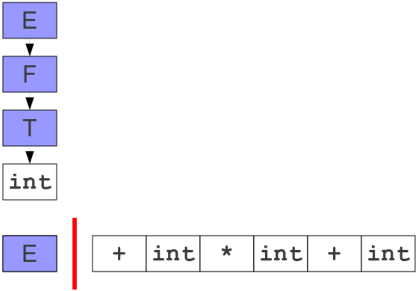
Shift/Reduce Algorithm

$\mathbf{E} \rightarrow \mathbf{F}$
$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$
$\mathbf{F} \rightarrow \mathbf{F} * \mathbf{T}$
$\mathbf{F} \rightarrow \mathbf{T}$
$\mathbf{T} \rightarrow \texttt{int}$
$\mathbf{T} \rightarrow (\mathbf{E})$



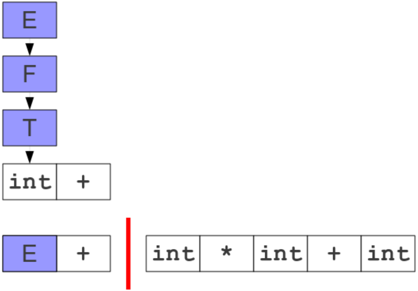
Shift/Reduce Algorithm

E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



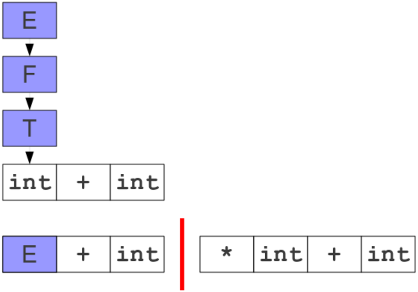
Shift/Reduce Algorithm

$\mathbf{E} \rightarrow \mathbf{F}$
$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$
$\mathbf{F} \rightarrow \mathbf{F} * \mathbf{T}$
$\mathbf{F} \rightarrow \mathbf{T}$
$\mathbf{T} \rightarrow \texttt{int}$
$\mathbf{T} \rightarrow \texttt{(E)}$



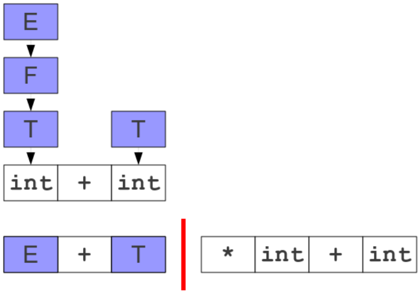
Shift/Reduce Algorithm

$\mathbf{E} \rightarrow \mathbf{F}$
$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$
$\mathbf{F} \rightarrow \mathbf{F} * \mathbf{T}$
$\mathbf{F} \rightarrow \mathbf{T}$
$\mathbf{T} \rightarrow \texttt{int}$
$\mathbf{T} \rightarrow (\mathbf{E})$



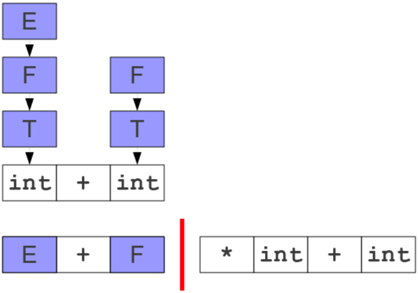
Shift/Reduce Algorithm

$\mathbf{E} \rightarrow \mathbf{F}$
$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$
$\mathbf{F} \rightarrow \mathbf{F} * \mathbf{T}$
$\mathbf{F} \rightarrow \mathbf{T}$
$\mathbf{T} \rightarrow \texttt{int}$
$\mathbf{T} \rightarrow \texttt{(E)}$



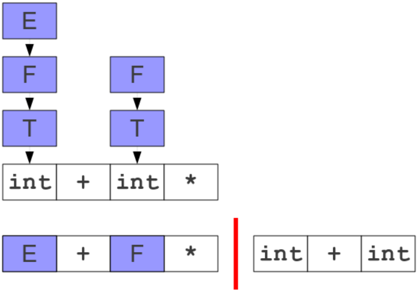
Shift/Reduce Algorithm

E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



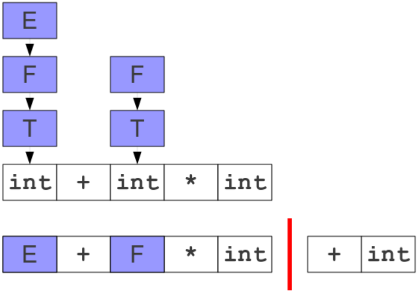
Shift/Reduce Algorithm

$E \rightarrow F$
$E \rightarrow E + F$
$F \rightarrow F * T$
$F \rightarrow T$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$



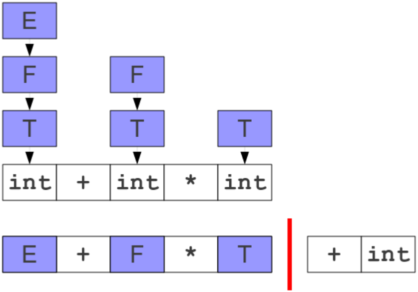
Shift/Reduce Algorithm

$\mathbf{E} \rightarrow \mathbf{F}$
$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$
$\mathbf{F} \rightarrow \mathbf{F} * \mathbf{T}$
$\mathbf{F} \rightarrow \mathbf{T}$
$\mathbf{T} \rightarrow$ `int`
$\mathbf{T} \rightarrow$ `(`$\mathbf{E}$`)`



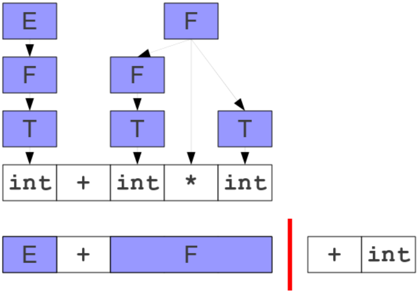
Shift/Reduce Algorithm

E → F
E → E + F
F → F * T
F → T
T → int
T → (E)



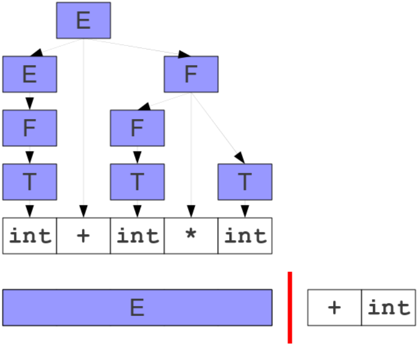
Shift/Reduce Algorithm

$$\begin{aligned}
\mathbf{E} &\rightarrow \mathbf{F} \\
\mathbf{E} &\rightarrow \mathbf{E} + \mathbf{F} \\
\mathbf{F} &\rightarrow \mathbf{F} * \mathbf{T} \\
\mathbf{F} &\rightarrow \mathbf{T} \\
\mathbf{T} &\rightarrow \mathtt{int} \\
\mathbf{T} &\rightarrow (\mathbf{E})
\end{aligned}$$



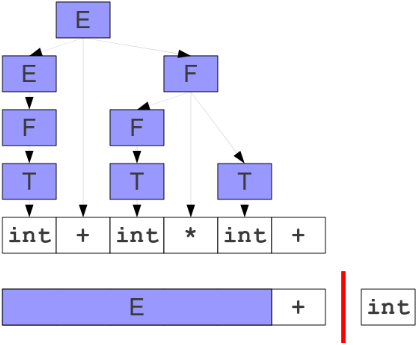
Shift/Reduce Algorithm

$\mathbf{E} \rightarrow \mathbf{F}$
$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$
$\mathbf{F} \rightarrow \mathbf{F} * \mathbf{T}$
$\mathbf{F} \rightarrow \mathbf{T}$
$\mathbf{T} \rightarrow$ `int`
$\mathbf{T} \rightarrow$ `(`$\mathbf{E}$`)`



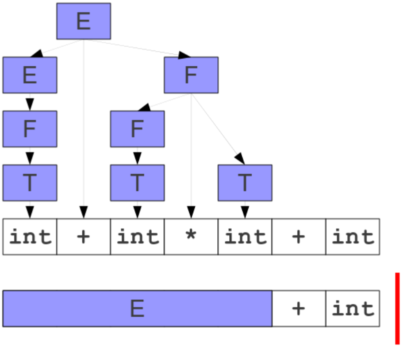
Shift/Reduce Algorithm

$\mathbf{E} \rightarrow \mathbf{F}$
$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$
$\mathbf{F} \rightarrow \mathbf{F} * \mathbf{T}$
$\mathbf{F} \rightarrow \mathbf{T}$
$\mathbf{T} \rightarrow$ `int`
$\mathbf{T} \rightarrow$ (E)



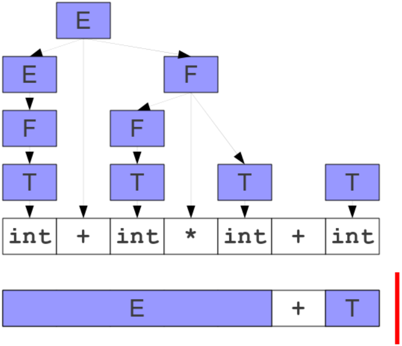
Shift/Reduce Algorithm

$\mathbf{E} \rightarrow \mathbf{F}$
$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$
$\mathbf{F} \rightarrow \mathbf{F} * \mathbf{T}$
$\mathbf{F} \rightarrow \mathbf{T}$
$\mathbf{T} \rightarrow$ `int`
$\mathbf{T} \rightarrow$ $(\mathbf{E})$



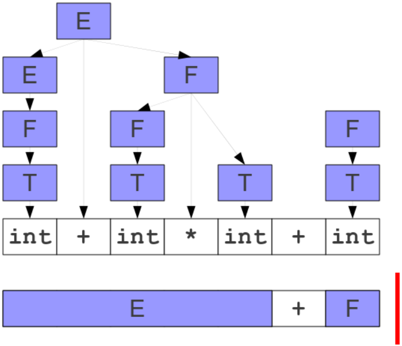
Shift/Reduce Algorithm

$\mathbf{E} \to \mathbf{F}$
$\mathbf{E} \to \mathbf{E} + \mathbf{F}$
$\mathbf{F} \to \mathbf{F} * \mathbf{T}$
$\mathbf{F} \to \mathbf{T}$
$\mathbf{T} \to \texttt{int}$
$\mathbf{T} \to \texttt{(}\mathbf{E}\texttt{)}$



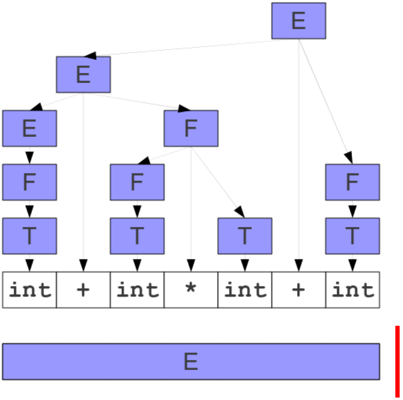
Shift/Reduce Algorithm

$\mathbf{E} \rightarrow \mathbf{F}$
$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$
$\mathbf{F} \rightarrow \mathbf{F} * \mathbf{T}$
$\mathbf{F} \rightarrow \mathbf{T}$
$\mathbf{T} \rightarrow \texttt{int}$
$\mathbf{T} \rightarrow (\mathbf{E})$



Shift/Reduce Algorithm

$\mathbf{E} \rightarrow \mathbf{F}$
$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{F}$
$\mathbf{F} \rightarrow \mathbf{F} * \mathbf{T}$
$\mathbf{F} \rightarrow \mathbf{T}$
$\mathbf{T} \rightarrow$ `int`
$\mathbf{T} \rightarrow$ `(E)`



Activity 3.2: Shift/Reduce Algorithm

Trace through the shift-reduce algorithm given the input:
`https://forms.gle/fvSRJFqpnTeBskKx6`

Observations

- Since reductions are always at the right side of the left area, we never need to shift from the left to the right.
- No need to "uncover" something to do a reduction.
- Consequently, shift/reduce parsing means
  Shift: Move a terminal from the right to the left area.
  Reduce: Replace some number of symbols at the right side of the left area.

Observations

- All activity in a shift/reduce parser is at the far right end of the left area.
- **Idea**: Represent the left area as a stack.

Observations

- All activity in a shift/reduce parser is at the far right end of the left area.
- **Idea**: Represent the left area as a stack.
- Shift: Push the next terminal onto the stack.
- Reduce: Pop some number of symbols from the stack, then push the appropriate non-terminal.

Reduce when we know we have a handle.

Finding Handles

- Where do we look for handles?
  - **At the top of the stack**.

Finding Handles

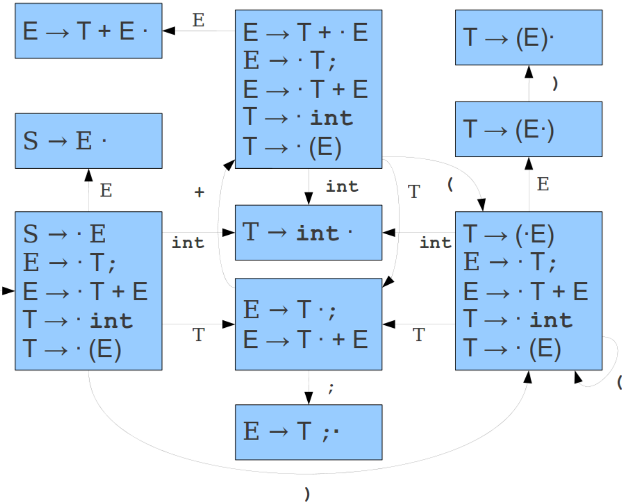
- Where do we look for handles?
  - **At the top of the stack**.
- How do we search for possible handles?

Finding Handles

- Where do we look for handles?
  - **At the top of the stack**.
- How do we search for possible handles?
  - **Build a handle-finding automaton.**
- How do we recognize handles?
  - Once we have found a candidate handle, how do we check that it really is the handle?

A Deterministic Automaton

$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow$ int
$T \rightarrow$ (E)



Handle Recognition

- Our automaton will tell us all places where a handle might be.
- However, if the automaton says that there might be a handle at a given point, we need a way to confirm this.

Handle Recognition

- Our automaton will tell us all places where a handle might be.
- However, if the automaton says that there might be a handle at a given point, we need a way to confirm this.
- We can use predictive bottom-up parsing: *Have a deterministic procedure for guessing where handles are.*
- There are many predictive algorithms, each of which recognize different grammars.

LR(1)

- Bottom-up predictive parsing with:
  L:Left-to-right scan
  R:Rightmost derivation
  (1): One token lookahead
- Tries to intelligently find handles by using a lookahead token at each step.

LR(1)

- Guess which series of productions we are reversing.
- Use this information to maintain information about what lookahead to expect.
- When deciding whether to shift or reduce, use lookahead to disambiguate.

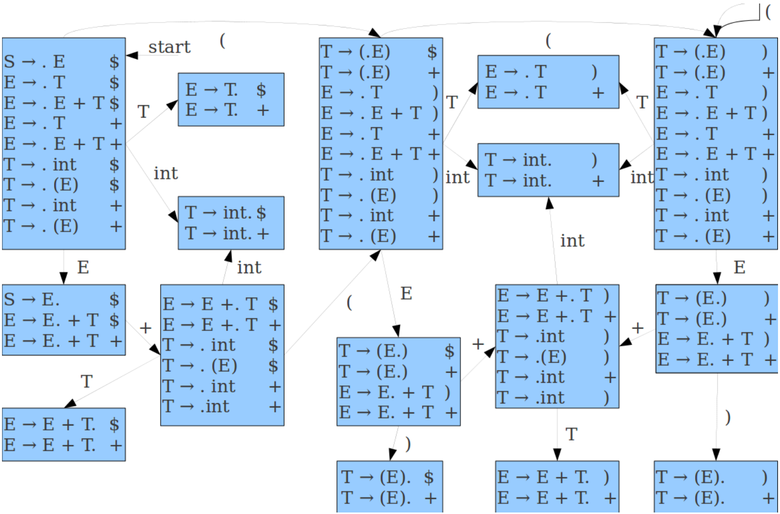
LR(1)

- How do we know what lookahead to expect at each state?
- Observation:
  - There are only finitely many productions we can be in at any point.
  - There are only finitely many positions we can be in each production.
  - There are only **finitely many lookahead sets** at each point.

LR(1)

- How do we know what lookahead to expect at each state?
- Observation:
  - There are only finitely many productions we can be in at any point.
  - There are only finitely many positions we can be in each production.
  - There are only **finitely many lookahead sets** at each point.
  - Construct an automaton to track lookaheads!

A Deterministic LR(1) Automata



Representing LR(1) Automata

- LR(1) parsers are usually represented via two tables: an action table and a goto table.
- The action table maps each state to an action:
  - shift, which shifts the next terminal, and
  - reduce $A \rightarrow \omega$, which performs reduction.

Representing LR(1) Automata

- LR(1) parsers are usually represented via two tables: an action table and a goto table.
- The action table maps each state to an action:
  - shift, which shifts the next terminal, and
  - reduce $A \to \omega$, which performs reduction.
- Any state of the form $A \to \omega\cdot$ does that reduction; everything else shifts.
- The goto table maps state/symbol pairs to a next state.
- This is just the transition table for the automaton.

LR Parsing table

(1) E → E + T
(2) E → T
(3) T → (E)
(4) T → id

State	Action					Goto	
	id	+	(	)	$	E	T
0	S4		S3			S1	S2
1		S5			accept		
2	R2	R2	R2	R2	R2		
3	S4		S3			S6	S2
4	R4	R4	R4	R4	R4		
5	S4		S3				S8
6		S5		S7			
7	R3	R3	R3	R3	R3		
8	R1	R1	R1	R1	R1		

LR Parsing table

S → E
E → T | E + T
T → id | (E)

Input: (id + id)

Parse Stack	Remaining Input	Action
	(id + id) $	Shift (
(	id + id) $	Shift id
(id	+ id) $	Reduce T → id
(T	+ id) $	Reduce E → T
(E	+ id) $	Shift +
(E+	id) $	Shift id
(E+id	) $	Reduce T → id
(E+T	) $	Reduce E → E+T; (Ignore: E→T)
(E	) $	Shift)
(E)	$	Reduce T → (E)
T	$	Reduce E → T
E	$	Reduce S → E
S	$	Accept

LR Parser

State on TOS	Action						Goto	
	id	+	(	)	$		E	T
0	S4		S3				S1	S2
1		S5			accept			
2	R2	R2	R2	R2	R2			
3	S4		S3				S6	S2
4	R4	R4	R4	R4	R4			
5	S4		S3					S8
6		S5		S7				
7	R3	R3	R3	R3	R3			
8	R1	R1	R1	R1	R1			

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow (E)$
(4) $T \rightarrow id$

State stack	Remaining Input	Parser action
S0	id + (id)$	Shift S4 onto state stack, move ahead in input
S0S4	+ (id)$	Reduce 4) T → id, pop state stack, goto S2, input unchanged
S0S2	+ (id)$	Reduce 2) E → T, goto S1
S0S1	+ (id)$	Shift S5
S0S1S5	(id)$	Shift S3
S0S1S5S3	id)$	Shift S4 (saw another id)
S0S1S5S3S4	)$	Reduce 4) T → id, goto S2
S0S1S5S3S2	)$	Reduce 2) E → T, goto S6
S0S1S5S3S6	)$	Shift S7
S0S1S5S3S6S7	$	Reduce 3) T → (E), goto S8
S0S1S5S8	$	Reduce 1) E → E + T, goto S1 *

LR(1) Algorithm

- Begin with an empty stack and the input set to $\omega\$$, where ω is the string to parse. Set **state** to the initial state.

LR(1) Algorithm

- Begin with an empty stack and the input set to $\omega\$$, where ω is the string to parse. Set **state** to the initial state.
- Repeat the following:
  - Let the next symbol of input be `t`.
  - If **action[state,t]** is **shift**, then shift the input and set **state=goto[state,t].**
  - If **action[state,t]** is reduce $A \rightarrow \omega$:
    - Pop $|\omega|$ symbols off the stack; replace them with A.
    - Let the state atop the stack be top-state.
    - Set **state=goto[top-state,A]**
  - If **action[state,t]** is accept, then the parse is done.
  - If **action[state,t]** is error, report an error.

LALR(1)

Look Ahead LR Parser

- LR(1) produces numerous states.
- LALR(1): merge similar states:
  - same core (LR(0))
  - only differences in lookaheads