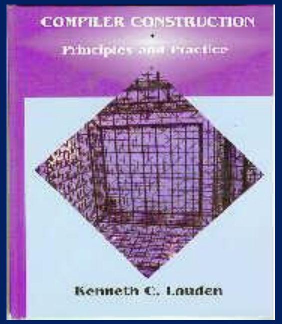
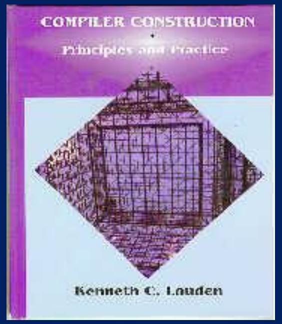


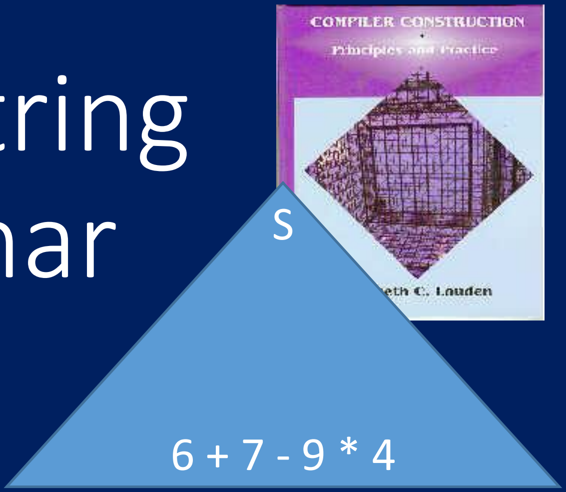
Chapter 4: Top Down Parsing

Outline



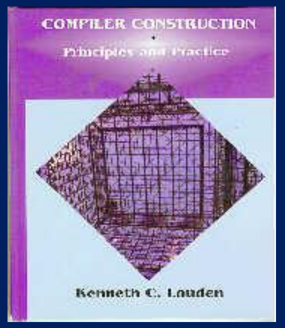
- Chapter 4 (top down) Vs. Chapter 5 (bottom up)
- Recursive descent parsers
- Predictive parsers (aka an LL(k) parser)

Remember: Given a grammar and a string in the language defined by the grammar



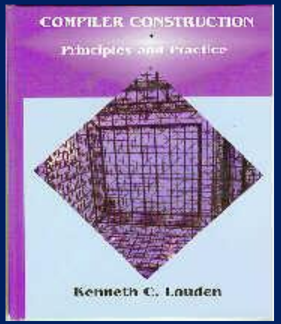
- The parsing problem is to connect the root node S with the tree leaves, the input
- There may be more than one way to derive the string leading to the same parse tree
  - it just depends on the order in which you apply the rules
  - and what parts of the string you choose to rewrite next
- All of the derivations are valid
- To simplify the problem and the algorithms, we often focus on one of
  - A leftmost derivation
  - A rightmost derivation

Top down vs. bottom up parsing



- Top-down parsers: starts constructing the parse tree at the top (root) and move down towards the leaves. - Chapter 4
  - Easy to implement by hand, but requires restricted grammars. E.g.:
    - Predictive parsers (e.g., LL(k))
- Bottom-up parsers: build nodes on the bottom of the parse tree first. – Chapter 5
  - Suitable for automatic parser generation, handles larger class of grammars. E.g.:
    - shift-reduce parser (or LR(k) parsers)
- Both are general techniques that can be made to work for all languages (but not all grammars!).

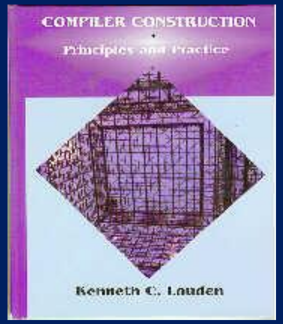
Two important parser classes: LL(k) and LR(k)



ANTLR (pronounced antler)
ANother Tool for Language Recognition,
uses LL(*). We will start with LL(1)

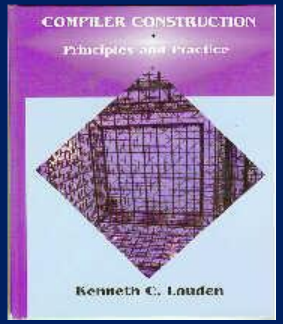
- The name LL(k) means:
  - L - Left-to-right scanning of the input
  - L - Constructing leftmost derivation
  - k – max number of input symbols needed to select parser action
- The name LR(k) means:
  - L - Left-to-right scanning of the input
  - R - Constructing rightmost derivation in reverse
  - k – max number of input symbols needed to select parser action
- Parsing an arbitrary CFG is $O(n^3)$.
  - If we constrain the grammar we can be $O(n)$.
- A LR(1) or LL(1) parser never need to “look ahead” more than one input token to know what parser production rule applies

Outline



- Chapter 4 (top down) Vs. Chapter 5 (bottom up)
- Recursive descent parsers
- Predictive parsers (aka an LL(k) parser)

Top Down Parsing Methods

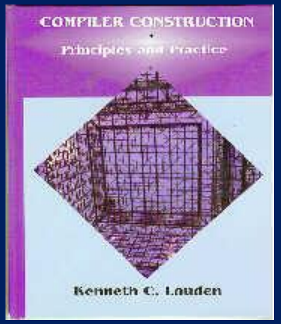


- Simplest method is a full-backup, recursive descent parser
- Often used for parsing simple languages
- Write recursive recognizers (subroutines) for each grammar rule
  - If rules succeeds perform some action (i.e., build a tree node, emit code, etc.)
  - If rule fails, return failure. Caller may try another choice or fail
  - On failure it "backs up"

For the grammar:

```
    <term> -> <factor> {(*|/)<factor>}*
void term() {  // Recursive descent parser written in C
   factor();    /* parse first factor*/
    while (next_token == ast_code ||  next_token == slash_code) {
        lexical();  /* get next token */
        factor();   /* parse next factor */
   }
}
```

Top Down Parsing Methods: Problems



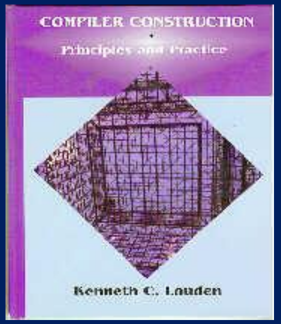
<S> ::= <A> | <B>

<A> ::= < < < <

<B> ::= < < < >

- Looking for < < < > (But we followed path A first).
- When going forward, the parser consumes tokens from the input, so what happens if we have to back up?
- Algorithms that use backup tend to be, in general, inefficient
  - There might be a large number of possibilities to try before finding the right one or giving up

Top Down Parsing Methods: Problems



<S> ::= <S> <

- The first step of <S> is to use production rule <S>.
  - This is like

```
void badFact(int n) {
    n * badFact(n-1);
    if (n==1) return 1;
}
```

Remember why you put the base case first?

- Grammar rules which are left-recursive lead to non-termination!
- Grammars can be indirectly left recursive:

  <S> ::= <A><B>…

  <A> ::= <S>

- We can manually or automatically rewrite a grammar removing left-recursion, making it ok for a top-down parser.

Eliminating Left Recursion

DrBC Note: Yes. We did change the associativity of the language.

- Consider left-recursive grammar

  S ::= S α

  S ::= β

- S generates strings

  β

  β α

  β α α …

- Rewrite using right-recursion

  S ::= β S’

  S’ ::= α S’ | ε

- Concretely

  T ::= T + id

  T ::= id

- T generates strings

  id

  id+id

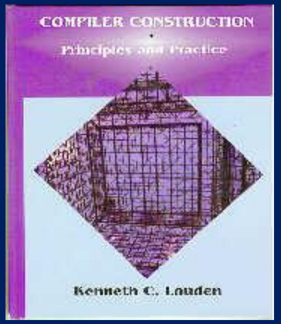
  id+id+id …

- Rewrite using right-recursion

  T -> id

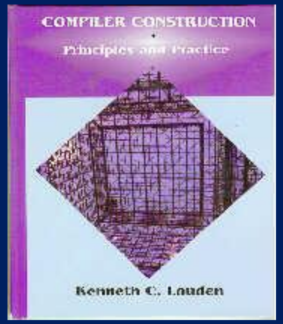
  T -> id T

Summary of Recursive Descent



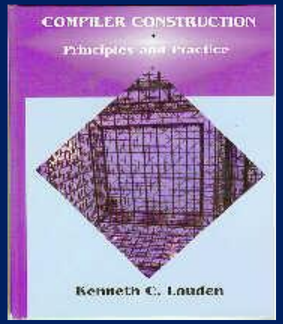
- Simple and general parsing strategy
  - Left-recursion must be eliminated first
  - … but that can be done automatically
- Unpopular because of backtracking
  - Thought to be too inefficient
- In practice, backtracking is eliminated by restricting the grammar, allowing us to successfully predict which rule to use
  - The gcc compiler now uses recursive descent

Outline



- Chapter 4 (top down) Vs. Chapter 5 (bottom up)
- Recursive descent parsers
- Predictive parsers (aka an LL(k) parser)

Predictive Parser



- A predictive parser uses information from the first terminal symbol of each expression to decide which production to use
- A predictive parser, aka an LL(k) parser, does a Left-to-right parse, a Leftmost-derivation, and k-symbol lookahead
  - Grammars where one can decide which rule to use by examining only the first token are LL(1)
  - LL(1) grammars are widely used in practice
  - The syntax of a Programing Language can usually be adjusted to enable it to be described with an LL(1) grammar

DrBC Note: if, then else begin print end ; and num will be referred to as handles.

Imagine a perfect world…

S ::= if E then S else S

S ::= begin S L

S ::= print E

An S expression starts either with an IF, BEGIN, or PRINT token.
If we only need to know the first token to know which production to use!

L ::= end

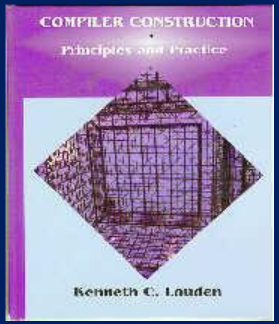
L ::= ; S L

There is no ambiguity with L either.
The first token is either end or ;
Either way we know what production to use.

E ::= num = num

E only has one choice! ☺

Predictive Parsing and Left Factoring (for the imperfect world)



E ::= T + E

E ::= T

For E, how could you predict which rule to use?

T ::= int

T ::= int * T

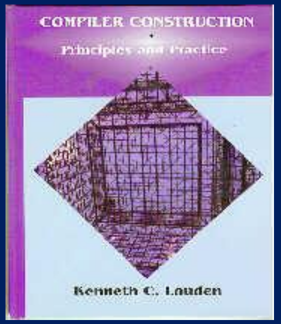
T ::= (E)

For T, two productions start with int

- A grammar must be left-factored before use for predictive parsing
- Left-factoring involves rewriting the rules so that, if a non-terminal has more than one rule, each begins with a terminal (handle).

Left-Factoring Example

Add new non-terminals X and Y to factor out common prefixes of rules



E ::= T + E

E ::= T

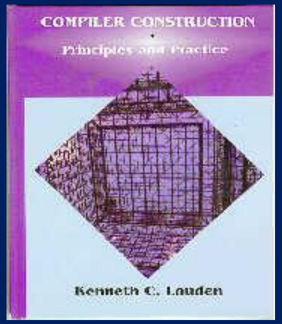
T ::= int

T ::= int * T

T ::= (E)

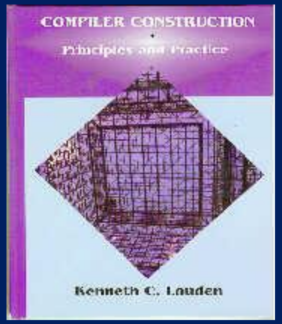
```
E ::= T X
  X ::= + E
  X ::= ε
T ::= ( E )
T ::= int Y
  Y ::= * T
  Y ::= ε
```

LL(1) Using Parsing Tables



- LL(1) means that for each non-terminal and token there is only one production
- Can be specified via 2D tables
  - One dimension for current non-terminal to expand
  - One dimension for next token
  - A table entry contains one production
- Method similar to recursive descent, except
  - For each non-terminal S
    - We look at the next token a
    - And chose the production shown at [S, a]
- We use a stack to keep track of pending non-terminals
- We reject when we encounter an error state
- We accept when we encounter end-of-input

LL(1) Parsing Table Example



E ::= T X

X ::= + E

T ::= (E)

T ::= int Y

Y ::= * T

Wouldn't it be nice to have a table like this...

Non-terminals

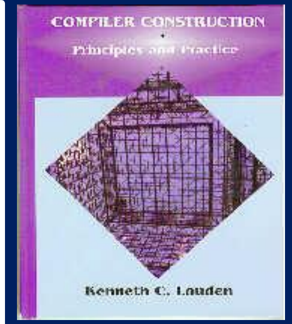
End of input symbol

Terminals

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

LL(1) Parsing Table Example

Stack	Input	Action
E $	int * int $	pop();push(T X)
T X $	int * int $	pop();push(int Y)
int Y X $	int * int $	pop();next++
Y X $	* int $	pop();push(* T)
* T X $	* int $	pop();next++
T X $	int $	pop();push(int Y)
int Y X $	int $	pop();next++;
Y X $	$	pop()
X $	$	pop()
$	$	ACCEPT! ← when $$



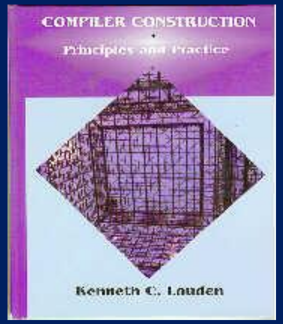
This table would not work if we ever had more than one option in the same cell.

E ::= T X
X ::= + E
X ::= ε
T ::= (E)
T ::= int Y
Y ::= * T
Y :: = ε

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

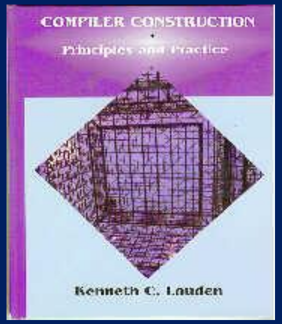
Empty cells are error conditions

Steps for automatically finding the LL Parse Table

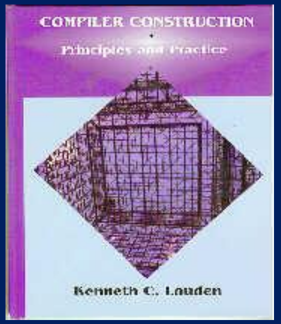


1. Remove the alternation "|" and list the terminals and nonterminals (with the start node::=$ added as production rule 0).
2. Compute first sets for nonterminals
3. Compute the follow sets (only needed if \epsilon is in grammar)
4. Compute the predict sets.
5. Create LL Parse Table

Steps for automatically finding the LL Parse Table



1. Remove the alternation “|” and list the terminals and nonterminals (with the start node::=$ added as production rule 0).
2. Compute first sets for nonterminals
3. Compute the follow sets (only needed if \epsilon is in grammar)
4. Compute the predict sets.
5. Create LL Parse Table



Remove the alternation "|"

E ::= T X

X ::= + E | ε

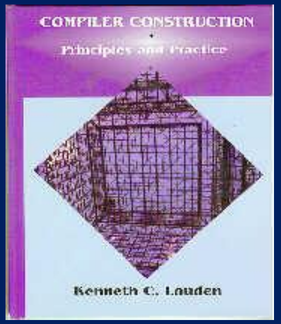
T ::= (E) | int Y

Y ::= * T | ε

→

0. S ::= E$
1. E ::= T X
2. X ::= + E
3. X ::= ε
4. T ::= (E)
5. T ::= int Y
6. Y ::= * T
7. Y ::= ε

Steps for automatically finding the LL Parse Table



1. Remove the alternation “|” and list the terminals and nonterminals (with the start node::=$ added as production rule 0).
2. Compute first sets for nonterminals
3. Compute the follow sets (only needed if \epsilon is in grammar)
4. Compute the predict sets.
5. Create LL Parse Table

Constructing Parsing Tables

DrBC Note: Essentially we are saying, if we have a production like this:

<addExp> ::= + <exp>

then the first is +

- LL(1) languages are those defined by a parsing table for the LL(1) algorithm
  - No table entry can be multiply defined
  - We want to generate parsing tables from CFG
- If A → α, where in the line of A can we place α?
  - In the column of t where t can start a string derived from α
    - α →* t β
    - We say that t ϵ First(α)
  - In the column of t if α is ε and t can follow an A
    - S →* β A t δ
    - We say t ϵ Follow(A)

If we have one like

<addExp> ::= <num>

Then then the first will be all the possible firsts of <num>

If we have one like

<addExp> ::= <num><val>

And <num> can be empty (ε)

Then the first will be all the possible firsts of <num> and of <val>

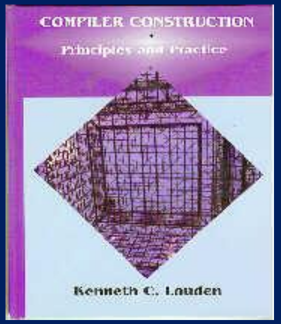
if we have one like

<addExp> ::= <num><val> …

And <num> and <val> etc. … can ALL be empty (ε)

Then the first will be all the possible firsts of <num> and of <val>, etc. PLUS the first of anything that can FOLLOW each of the terms.

Computing First Sets

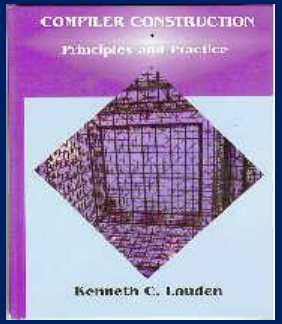


Definition: First(X) = { t | X →* tα} U {ε | X →* ε}

Algorithm sketch:

1. for all terminals t do First(t) ← { t }
2. for each production $X \rightarrow \varepsilon$ do First(X) ← { ε }
3. if $X \rightarrow A_1 \ldots A_n\ \alpha$ and $\varepsilon \in$ First(A_i), $1 \leq i \leq n$ do
   - add First(α) to First(X)
4. for each $X \rightarrow A_1 \ldots A_n$ s.t. $\varepsilon \in$ First(A_i), $1 \leq i \leq n$ do
   - add ε to First(X)
5. repeat steps 4 & 5 until no First set can be grown

First Sets. Example.



Recall the grammar

E ::= T X

X ::= + E

X ::= ε

T ::= (E)

T ::= int Y

Y ::= * T

Y ::= ε

Firsts of terminals

First(( ) = { (}

First( )) = {) }

First(int) = { int }

First(+) = { + }

First(*) = { * }

Firsts of non terminals

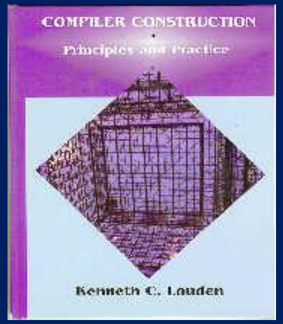
First(E) = {int, (} first(T)

First(X) = {+, ε }

First(T) = {int, (}

First(Y) = {*, ε }

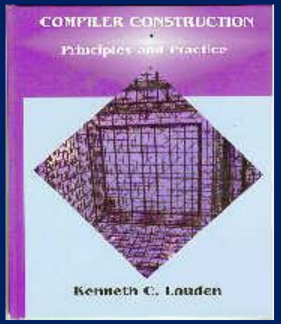
Steps for automatically finding the LL Parse Table



1. Remove the alternation “|” and list the terminals and nonterminals (with the start node::=$ added as production rule 0).
2. Compute first sets for nonterminals
3. Compute the follow sets (only needed if \epsilon is in grammar)
4. Compute the predict sets.
5. Create LL Parse Table

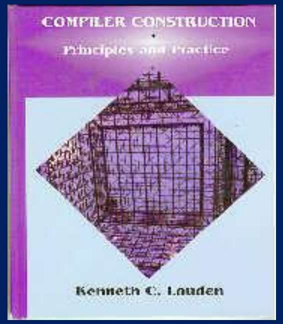
Computing Follow Sets

Definition: $\text{Follow}(X) = \{\, t \mid S \rightarrow^{*} \beta\, X\, t\, \delta \,\}$



- Intuition
  - If S is the start symbol then $\$ \in \text{Follow}(S)$
  - If $X \rightarrow A\ B$ then $\text{First}(B) \subseteq \text{Follow}(A)$ and
    $\text{Follow}(X) \subseteq \text{Follow}(B)$
  - Also if $B \rightarrow^{*} \varepsilon$ then $\text{Follow}(X) \subseteq \text{Follow}(A)$

Computing Follow Sets



Algorithm sketch:

1. Follow(S) $\leftarrow$ { \$ }
2. For each production $A \rightarrow \alpha\ X\ \beta$
   - add First(β) - {ε} to Follow(X)
3. For each $A \rightarrow \alpha\ X\ \beta$ where $\varepsilon \in$ First(β)
   - add Follow(A) to Follow(X)
4. repeat step(s) ___ until no Follow set grows

Compute Follow Sets

Recall
First(T) = {int, (}
First(E) = {int, (}
First(Y) = {*, ε }
First(+) = { + }

First(( ) = { (}
First( )) = {) }
First(int) = { int }
First(X) = {+, ε }
First(*) = { * }

Recall the grammar

0. S ::= E$
1. E ::= T X
2. X ::= + E
3. X ::= ε
4. T ::= (E)
5. T ::= int Y
6. Y ::= * T
7. Y ::= ε

Follow(E) = {Follow(X) ,), $}
Follow(T) = {First(X) , Follow(Y) , Follow(X) }
Follow(X) = { Follow(E) } - Remember X can be ε
Follow(+) = {First(E)}
Follow(( ) = { First(E)}
Follow( )) = {Follow(T) }
Follow(int) = { First(Y), Follow(Y) }
Follow(Y) = {Follow(T) } - Remember Y can be ε
Follow(*) = {First(T)}

Compute Follow Sets

Recall

First(T) = {int, (}
First(E) = {int, (}
First(Y) = {*, ε }
First(+) = { + }

First(( ) = { (}
First( )) = {) }
First(int) = { int }
First(X) = {+, ε }
First(*) = { * }

Follow(E) = {Follow(X) ,), $ }
Follow(T) = {First(X) + , Follow(Y) , Follow(X) }
Follow(X) = { Follow(E) }
Follow(+) = {First(E) int, (}
Follow(( ) = { First(E) int, (}
Follow( )) = {Follow(T) }
Follow(int) = { First(Y) *, Follow(Y) }
Follow(Y) = {Follow(T) }
Follow(*) = {First(T) int, (}

Compute Follow Sets

Recall

First(T) = {int, (}

First(E) = {int, (}

First(Y) = {*, ε }

First(+) = { + }

First(( ) = { (}

First( )) = {) }

First(int) = { int }

First(X) = {+, ε }

First(*) = { * }

Follow(E) = {Follow(X) ,), $ }

Follow(T) = { + , Follow(Y) , Follow(X) }

Follow(X) = { Follow(E)), $ }

Follow(+) = { int, (}

Follow(( ) = { int, (}

Follow( )) = {Follow(T) }

Follow(int) = { * , Follow(Y) }

Follow(Y) = {Follow(T) }

Follow(*) = { int, (}

Compute Follow Sets

Recall
First(T) = {int, (}
First(E) = {int, (}
First(Y) = {*, ε }
First(+) = { + }

First(( ) = { (}
First( )) = {) }
First(int) = { int }
First(X) = {+, ε }
First(*) = { * }

Follow(E) = {), $ }
Follow(T) = { + , Follow(Y) , Follow(X)), $ }
Follow(X) = {), $ }
Follow(+) = { int, (}
Follow(( ) = { int, (}
Follow( )) = {Follow(T) }
Follow(int) = { * , Follow(Y) }
Follow(Y) = {Follow(T) }
Follow(*) = { int, (}

Compute Follow Sets

Recall

First(T) = {int, (}

First(E) = {int, (}

First(Y) = {*, ε }

First(+) = { + }

First(( ) = { (}

First( )) = {) }

First(int) = { int }

First(X) = {+, ε }

First(*) = { * }

Follow(E) = {), $ }

Follow(T) = { + , Follow(Y) ,), $ }

Follow(X) = {), $ }

Follow(+) = { int, (}

Follow(( ) = { int, (}

Follow( )) = {Follow(T) }

Follow(int) = { * , Follow(Y) }

Follow(Y) = { Follow(T) +,), $ }

Follow(*) = { int, (}

Compute Follow Sets

Recall

First(T) = {int, (}

First(E) = {int, (}

First(Y) = {*, ε }

First(+) = { + }

First(( ) = { (}

First( )) = {) }

First(int) = { int }

First(X) = {+, ε }

First(*) = { * }

Follow(E) = {), $ }

Follow(T) = { + ,), $ }

Follow(X) = {), $ }

Follow(+) = { int, (}

Follow(( ) = { int, (}

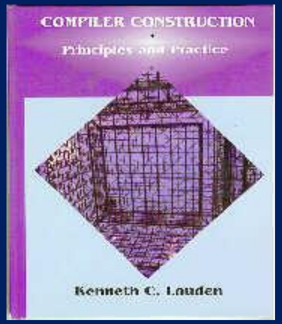
Follow( )) = {Follow(T) +,), $ }

Follow(int) = { * , Follow(Y) +,), $ }

Follow(Y) = { + ,) , $ }

Follow(*) = { int, (}

Compute Follow Sets



Follow(E) = {), $ }

Follow(T) = { +,), $ }

Follow(X) = {), $ }

Follow(+) = { int, (}

Follow(( ) = { int, (}

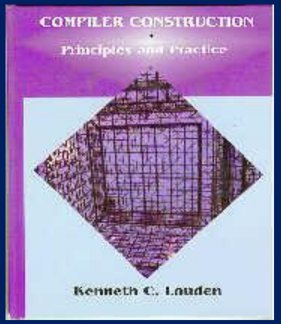
Follow( )) = { +,), $ }

Follow(int) = { *, +,), $ }

Follow(Y) = { +,), $ }

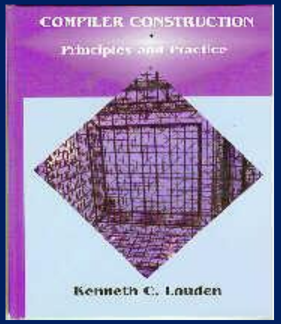
Follow(*) = { int, (}

Steps for automatically finding the LL Parse Table



1. Remove the alternation "|" and list the terminals and nonterminals (with the start node::=$ added as production rule 0).
2. Compute first sets for nonterminals
3. Compute the follow sets (only needed if \epsilon is in grammar)
4. Compute the predict sets.
5. Create LL Parse Table

Constructing LL(1) Parsing Tables



- Construct a parsing table T for CFG G
- For each production $A \rightarrow \alpha$ in G do:
  - For each terminal $t \in \text{First}(\alpha)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$, for each $t \in \text{Follow}(A)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(A)$ do
    - $T[A, \$] = \alpha$

Compute the predict sets

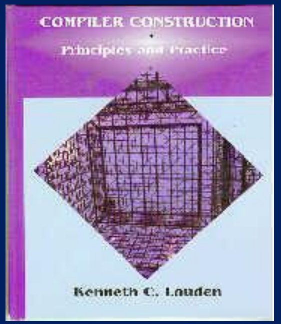
DrBC Note: I usually combine this with the next step, since the sets on the LEFT for an expression on the RIGHT must have empty intersections, and that is easier to see if you put them directly into the table.

First(E) = {int, (}
First(X) = {+, ε }
First(T) = {int, (}
First(Y) = {*, ε }

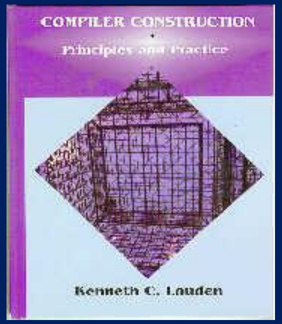
Follow(X) = {$,) }
Follow(Y) = {+,) , $}

Production	First set of the first symbol on the right hand side
1. E ::= T X	int, (
2. X ::= + E	+
3. X ::= ε	), $ - The follow of X
4. T ::= (E)	(
5. T ::= int Y	int
6. Y ::= * T	*
7. Y ::= ε	+,),$ - The follow of Y

Steps for automatically finding the LL Parse Table



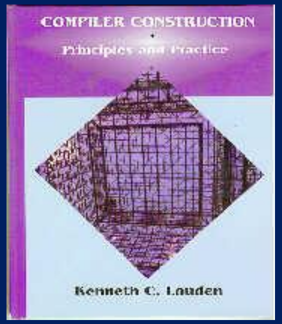
1. Remove the alternation "|" and list the terminals and nonterminals (with the start node::=$ added as production rule 0).
2. Compute first sets for nonterminals
3. Compute the follow sets (only needed if \epsilon is in grammar)
4. Compute the predict sets.
5. Create LL Parse Table



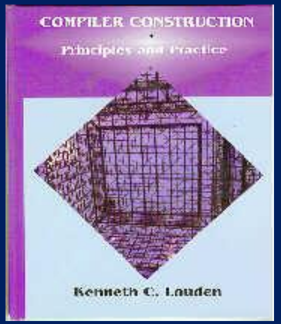
1.	E ::= T X	int, (
2.	X ::= + E	+
3.	X ::= ε	), $
4.	T ::= (E)	(
5.	T ::= int Y	int
6.	Y ::= * T	*
7.	Y ::= ε	+,),$

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

Steps for automatically finding the LL Parse Table



1. Remove the alternation "|" and list the terminals and nonterminals (with the start node::=$ added as production rule 0).
2. Compute first sets for nonterminals
3. Compute the follow sets (only needed if \epsilon is in grammar)
4. Compute the predict sets. – Dr. BC's Way.
5. Create LL Parse Table



E ::= T X

X ::= + E

T ::= (E)

T ::= int Y

Y ::= * T

First(E) = {int, (}

First(X) = {+, ε }

First(T) = {int, (}

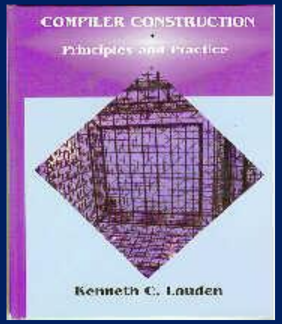
First(Y) = {*, ε }

Follow(X) = {$,) }

Follow(Y) = {+,) , $}

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

Notes on LL(1) Parsing Tables



- If any entry is multiply defined then G is not LL(1)
  - If G is ambiguous
  - If G is left recursive
  - If G is not left-factored
- Most programming language grammars are not LL(1)
- There are tools that build LL(1) tables