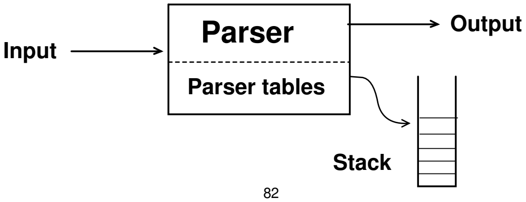
## LL(1) Parsing

- LL(1) is a Top Down parsing technique.
  An LL(1) parser is a *deterministic push down automata (DPDA)*
- Top down parsers are *predictive* parsers.
  Parse stack represents what the parser expects to see. As the parser encounters token that it expected to see, the parse stack gets modified to record this fact.
- LL(1) parser is controled by the incoming token and the top item in the parse stack. (LR parsers use more of the stack).
- If the LL(1) constructor algorithm succeeds, i.e. there are no conflicting table entries for some particular row,column then the language is LL(1) (later in semester).






## LL(1) Example

Grammar

S → d S A
  → b A c
A → d A
  → c

Input Tokens

d b c c d c ⊢

LL(1) Table

| | d | b | c | ⊢ |
|---|---|---|---|---|
| S | pop S push dSa | pop S push bAc | Error | Error |
| A | pop A push dA | Error | pop A push c | Error |
| d | pop d Next | Error | Error | Error |
| b | Error | Pop b Next | Error | Error |
| c | Error | Error | pop c Next | Error |
| ▽ | Error | Error | Error | Accept |

push xYz = push z push Y push x

Stack Snapshots

S ▽
d S A ▽
S A ▽ Next b
b A c A ▽
A c A ▽ Next c
c c A ▽
c A ▽ Next c
A ▽ Next d
d A ▽
A ▽ Next c
c ▽
▽ Next ⊢
Accept



## Reading Assignment

Fischer & LeBlanc

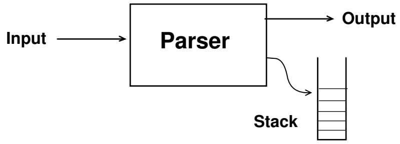
Chapter 6
Omit 6.7.1 6.7.3
Omit 6.8 6.10 6.12



## Shift Reduce Parsing

Bottom Up - Operator Precedence, LR(k), SLR(k), LALR(k)

- Parser Model




- Parser Actions

| | |
|---|---|
| Shift | Next input symbol is pushed onto the stack |
| Reduce | A sequence of symbols ( the *handle* ) starting at the top of the stack is reduced using a production rule to replace the symbols with one nonterminal symbol. |
| Accept | Successful end of parse. |
| Error | Call recovery routine to handle syntax error. |

- Choice of actions is based on the *contents* of the stack and the next $k$ input tokens.
- *Issues:* Location of the handle.
  Choice of replacement nonterminal symbol.

## Operator Precedence Parsing

- Simple form of shift-reduce parsing. Limited power but useful for expressions.
- Define three disjoint precedence relations $\lessdot$ , $\doteq$ , $\gtrdot$ between pairs of terminal symbols. Use these relations to locate handles.

| | | |
|---|---|---|
| a $\lessdot$ b | a yields precedence to b | marks left end of handle |
| a $\doteq$ b | a has the same precedence as b | marks interior of handle |
| a $\gtrdot$ b | a takes precedence over b | marks right end of handle |

- Example - precedence functions for a simple expression grammar

| Top of Stack | Input Token + | * | id | ( | ) | EOF |
|---|---|---|---|---|---|---|
| + | $\gtrdot$ | $\lessdot$ | $\lessdot$ | $\lessdot$ | $\gtrdot$ | $\gtrdot$ |
| * | $\gtrdot$ | $\gtrdot$ | $\lessdot$ | $\lessdot$ | $\gtrdot$ | $\gtrdot$ |
| id | $\gtrdot$ | $\gtrdot$ | | | $\gtrdot$ | $\gtrdot$ |
| ( | $\lessdot$ | $\lessdot$ | $\lessdot$ | $\lessdot$ | $\doteq$ | |
| ) | $\gtrdot$ | $\gtrdot$ | | | $\gtrdot$ | $\gtrdot$ |
| BOS | $\lessdot$ | $\lessdot$ | $\lessdot$ | $\lessdot$ | | |



## Operator Precedence Parsing

Let $a$ be the topmost terminal on the stack
Let $b$ be the current input symbol.

- if input is EOF and stack is empty
  ACCEPT
- if $a \lessdot b$ or $a \doteq b$
  SHIFT $b$ onto the stack
- if $a \gtrdot b$
  REDUCE by popping stack until top stack item is
  $\lessdot$ the last item popped.
- Otherwise
  ERROR

Note that this parsing algorithm makes little use of the stack in making decisions.



## LL(k) and LR(k) Parsing

- A parser which can determine the correct derivation (reduction) at each stage as it scans the input stream in one direction is called *deterministic.*
- A grammar is LL(k) if a parser can parse strings scanning from left to right using *left* derivations with k-symbol lookahead deterministically.
- A grammar is LR(k) if a parser can parse strings scanning from left to right using *right* reductions with k-symbol lookahead deterministically.
- LR(k) is the most general deterministic left-to-right language.
  Language properties:
  $\mathcal{L}$( LL(k) ) $\subset$ $\mathcal{L}$( LR(1) )
  $\mathcal{L}$( LR(k) ) $\subset$ $\mathcal{L}$( LR(1) )
  $\mathcal{L}$( LL(k) ) $\subset$ $\mathcal{L}$( LR(k) )
  There are LR(0) grammars which are not LL(k) for any k.



## First and Follow Sets

- Notation

| | |
|---|---|
| $A, B, S, \ldots$ | Nonterminal symbols |
| $a, b, c, \ldots$ | Terminal symbols |
| $\alpha, \beta, \gamma, \ldots$ | Strings of terminal and nonterminal symbols. |

- Define: $first(\ \alpha\ )$
  For a string $\alpha$ ( $\alpha \ \epsilon\ (\ V_n\ \cup\ V_t\ )^*$ ) the set $first(\ \alpha\ )$ is the set of *terminal* symbols that can occur at the beginning of strings derived from $\alpha$. Examples

| | | | |
|---|---|---|---|
| $first(\ a\ \beta\ )$ | = | $\{a\}$ | $a$ is a terminal symbol |
| $first(\ \lambda\ )$ | = | $\{\ \ \}$ | |
| $first(\ B\ \alpha\ )$ | = | $first(\ B\ )$ | B not nullable |
| | = | $first(\ B\ )\ \cup\ first(\ \alpha\ )$ | B nullable |

NOTE Fischer & LeBlanc include $\lambda$ in $first$ , these notes do not.

- Define: $follow(\ A\ )$
  For a nonterminal symbol $A$, the set $follow(\ A\ )$ is the set of *terminal symbols* that can immediately follow $A$ in any intermediate string derived from the goal symbol $S$
  $follow(\ A\ ) = \{\ w\ | S \rightarrow^*\ \alpha\ A\ \beta\ w\ \gamma\ \wedge\ \beta\ is\ nullable\}$

## LR(k) Parsing

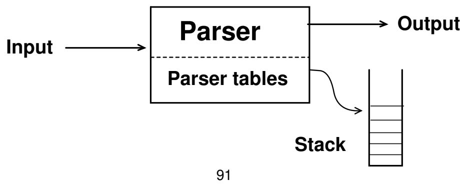
- LR(k) parsers are the most powerful class of deterministic bottom up parsers using k-symbol lookahead.
- If a grammar $G$ can be parsed by any deterministic parser with k-symbol lookahead, then it can be parsed by an LR(k) parser.
- A grammar $G$ is LR(k) if and only if the conditions:
    1. $S \rightarrow^* \alpha\, A\, w \rightarrow \alpha\, \beta\, w$
    2. $S \rightarrow^* \gamma\, B\, x \rightarrow \alpha\, \beta\, y$
    3. $first_k(w) = first_k(y)$

  imply that $\alpha\, A\, y = \gamma\, B\, x$
- SLR(k) and LALR(k) are practical versions of LR(k).
  Almost all practical compilers use k = 1.



## LR(1) Parsing

- LR(1) is a *bottom up* parse.
- Stack represents what the parser *has seen so far*.
- Parser is a *deterministic push down automata (DPDA)*
- Parser is controled by the incoming token (k = 1) and by the *contents* of the parser stack.
  *State numbers* (row indices) are used to identify possible parse stack contents.
- Parser uses a Stack/Reduce algorithm like operator precedence parsing.






## LR(1) Parsing

- The contents of the parser stack represents a string from which the past input can be derived.
- Inputs are stacked until the top elements in the stack (*the handle*) are a complete alternative (RHS) for some rule.
- When a handle is recognized, a reduction is performed and the handle on the stack is replaced by the nonterminal symbol (LHS) of the applicable rule.
- Initial parser stack is $\bigtriangledown$ and parsing continues until the stack contains $S\ \bigtriangledown$ and the next input is $\vdash$
- At each stage the the top elements in the stack represent the initial portion of *one or more* alternatives. The next input symbol narrows the number of number of possible alternatives. If the number of choices in narrowed to zero a syntax error has occurred.



## LR(1) Parsing

- Finally an input symbol is stacked that completes one or more alternatives.
  If there is more than one alternative the language is not LR(0).
- At this point the next input symbol (or $k$ symbols) must provide enough information to distinguish amoung the alternatives.
  If it doesn't the language isn't LR(1) (or LR(k).
- For LL(1) we had to know at the start of an alternative, given 1 input symbol which alternative to choose.
  For LR(1) we do not need to know which alternative to choose until we reach the end of a rule.
  *Then* the next input symbol must be sufficient to decide so that we can perform a reduction.
- Thus parsing decisions can be made *later* in an LR(1) parser than in LL(1).
  This is the reasion that $\mathcal{L}$( LL(k) ) $\subset$ $\mathcal{L}$( LR(1) )

## LR(1) Example

| Rule | Grammar |
|---|---|
| 1 | L → L , E |
| 2 | → E |
| 3 | E → a |
| 4 | → b |

Input Tokens

a , b ⊢

LR(1) Table

| Stack Confg | a | b | , | ⊢ | St# |
|---|---|---|---|---|---|
| ▽ | Shift a Next | Shift b Next | Error | Error | 0 |
| a ▽ | Error | Error | Reduce 3 | Reduce 3 | 1 |
| b ▽ | Error | Error | Reduce 4 | Reduce 4 | 2 |
| E ▽ | Error | Error | Reduce 2 | Reduce 2 | 3 |
| L ▽ | Error | Error | Shift , Next | Accept | 4 |
| , L ▽ | Shift a Next | Shift b Next | Error | Error | 5 |
| a , L ▽ | Error | Error | Reduce 3 | Reduce 3 | 6 |
| b , L ▽ | Error | Error | Reduce 4 | Reduce 4 | 7 |
| E , L ▽ | Error | Error | Reduce 1 | Reduce 1 | 8 |

Stack Snapshots

| Parse | State | Input | Action |
|---|---|---|---|
| ▽ | ▽ | a | Next |
| a ▽ | 1 ▽ | , | |
| E ▽ | 3 ▽ | , | |
| L ▽ | 4 ▽ | , | Next |
| , L ▽ | 5 4 ▽ | b | Next |
| b , L ▽ | 7 5 4 ▽ | ⊢ | |
| E , L ▽ | 8 5 4 ▽ | ⊢ | |
| L ▽ | 4 ▽ | ⊢ | Accept |



## LR(1) Parse Tables

- Some rows in table are the same as others, e.g. rows 1/6, 2/7. For smaller tables these rows can be merged and assigned multiple indices.
- If the grammar is right recursive the number of different parse stacks is infinite, but the number of different rows is finite (bounded by number of actions $\times$ number of columns) so we *must* merge the rows that are the same.
- Don't want the inefficiency of pattern matching the top elements in the stack against alternative stack configurations.
  We want parsing action to be determined by the single top element on the stack and a single input symbols (as in LL(1)).
- Assign a *state number* to each row in the table and stack the state number as a synonym for a complete stack configuration.
  The state number is labled *St#* in Slide 94.
- Define: *Shift* $i$
  Stack input symbol  Advance input  Go to State $i$



## LR(1) Example Revisited

Grammar

| | | | |
|---|---|---|---|
| 1 | L | → | L , E |
| 2 | | → | E |
| 3 | E | → | a |
| 4 | | → | b |

Condensed Parse Table

| Stack | | St# | a | b | , | ⊢ |
|---|---|---|---|---|---|---|
| | ▽ | 0 | Shift 1 | Shift 2 | | Accept |
| a ▽ | a , L ▽ | 1 | | | Reduce 3 | Reduce 3 |
| b ▽ | b , L ▽ | 2 | | | Reduce4 | Reduce 4 |
| | E ▽ | 3 | | | Reduce 2 | Reduce 2 |
| | L ▽ | 4 | | | Shift 5 | Reduce 0 |
| | , L ▽ | 5 | Shift 1 | Shift 2 | | |
| | E , L ▽ | 6 | | | Reduce 1 | Reduce 1 |

- After a Reduce, need a list of states to restart in, the *new state table*.
  With this table we don't need to represent the stack configurations directly in the parser tables.
- Top state on parse stack represents entire stack configuration



## Parsing Example Revisited

Example: parse of  a , b ⊢

| Old Stack | State Stack | Input | Action |
|---|---|---|---|
| ▽ | 0 | a | Shift 1 |
| a ▽ | 1 0 | , | Reduce 3 |
| E ▽ | 3 0 | , | Reduce 2 |
| L ▽ | 4 0 | , | Shift 5 |
| , L ▽ | 5 4 0 | b | Shift 2 |
| b , L ▽ | 2 5 4 0 | ⊢ | Reduce 4 |
| E , L ▽ | 6 5 4 0 | ⊢ | Reduce 1 |
| L ▽ | 4 0 | ⊢ | Reduce 0 |
| ▽ | 0 | ⊢ | Accept |

## Applying Reductions

- First $pop^l$ where $l$ is the length of the selected alternative.
  Need a table giving the length of each alternative.
  This popping uncovers some state. Note state 0 is never popped.
- Push a new state where new state is a function of top item in the state stack and the nonterminal symbol that is being *reduced to*
  State 0 on top represents signifies the empty stack.
  For example in previous slide:

| Reducing | Top state | Push state | From state | Rule(s) |
|---|---|---|---|---|
| E | 0 | 3 | 1,2 | 3,4 |
| E | 5 | 6 | 1,2 | 3,4 |
| L | 0 | 4 | 3 | 2 |
| L | 6 | 4 | 6 | 1 |



food for slide eater



food for slide eater



food for slide eater