

Bottom-Up Parsing Algorithms

Lecture Notes by
Profs. Alex Aiken and George Necula
(UCB)

CS780(Prasad)

L134ALG

1

Outline

- More about handles
- **Viable prefixes**: A building block for recognizing handles
- Computing **viable prefixes**
- Simple LR parsing (SLR)

CS780(Prasad)

L134ALG

2

Review

- $\alpha\beta$ is a **handle** of $\alpha\beta\omega$ in a rightmost derivation if
$$S \Rightarrow^* \alpha X \omega \rightarrow \alpha\beta\omega$$
- Handles always appear at the top of the stack
 - To the right of the rightmost non-terminal.
 - Thus shift and reduce moves are sufficient.
- To implement shift-reduce parsing, we must detect handles.
- But even if there are symbols on the stack that can be reduced, that doesn't mean there is a handle.

CS780(Prasad)

L134ALG

3

Example: Reduction w/o a Handle

- Recall our favorite grammar:
$$E \rightarrow T + E \mid T$$
$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$
- Now the step
$$T * \text{int} + \text{int} \rightarrow \text{int} * \text{int} + \text{int}$$
is not part of any rightmost derivation.
- Thus, **int** is *not* a handle of **int * int + int**.

CS780(Prasad)

L134ALG

4

Notes on Handles

- Every handle has some production rhs on top of the stack. But a production rhs on top of the stack is not necessarily a handle.
- In other words, whether a given stack has handle can depend on the contents of the entire stack.
- **Unique Handles**: If a grammar is unambiguous, then every step of a rightmost derivation has a unique handle.

CS780(Prasad)

L134ALG

5

Viable Prefixes

- It is not obvious how to detect handles.
- At each step the parser sees only the stack, not the entire input.

α is a **viable prefix** if there is an ω such that $\alpha|\omega$ is a state of a shift-reduce parser.

CS780(Prasad)

L134ALG

6

Huh?

- What does this mean? A few things:
 - A viable prefix does not extend past the right end of the handle.
 - It's a viable prefix because it is a prefix of the handle.
 - A viable prefix can be extended to a sentential form by adding terminals to the right.
 - As long as a parser has viable prefixes on the stack, no parsing error has been detected.

CS780(Prasad)

L134ALG

7

Important Fact #3

Important Fact #3 about bottom-up parsing:

For any grammar, the set of viable prefixes is a regular language.

- This fact is non-obvious.
- We show how to compute automata that accept viable prefixes.

CS780(Prasad)

L134ALG

8

Items

- An **item** is a production with a "." somewhere on the rhs.
- The items for $T \rightarrow (E)$ are
 - $T \rightarrow \cdot(E)$
 - $T \rightarrow (\cdot E)$
 - $T \rightarrow (E\cdot)$
 - $T \rightarrow (E)\cdot$
- The only item for $X \rightarrow \epsilon$ is $X \rightarrow \cdot$.
- Items are often called "LR(0) items".

CS780(Prasad)

L134ALG

9

Intuition

- The problem in recognizing viable prefixes is that the stack has only bits and pieces of the rhs of productions.
 - If it had a complete rhs, we could **reduce**.
- These bits and pieces are always *prefixes* of rhs of productions.

CS780(Prasad)

L134ALG

10

Example

Consider the input (int)

- Then (E) is a state of a shift-reduce parse.
- (E is a prefix of the rhs of $T \rightarrow (E)$
 - Will be reduced after the next shift.
- Item $T \rightarrow (E\cdot)$ says that so far we have seen (E of this production and hope to see)

CS780(Prasad)

L134ALG

11

Generalization

- The stack may have many prefixes of rhs's
 $Prefix_1 Prefix_2 \dots Prefix_{n-1} Prefix_n$
- Let $Prefix_i$ be a prefix of rhs of $X_i \rightarrow \alpha_i$
 - Now $Prefix_n$ is a prefix of α_n
 - Which eventually reduces to X_n
 - Which should be a prefix of the missing part of α_{n-1}
- Recursively, $Prefix_{k+1} \dots Prefix_n$ eventually reduces to the missing part of α_k

CS780(Prasad)

L134ALG

12

An Example

Consider the string $(int * int)$:

$(int * | int)$ is a state of a shift-reduce parse.

"(" is a prefix of the rhs of $T \rightarrow (E)$

"ε" is a prefix of the rhs of $E \rightarrow T$

"int *" is a prefix of the rhs of $T \rightarrow int * T$

An Example (Cont.)

The "stack of items"

$T \rightarrow (.E)$

$E \rightarrow .T$

$T \rightarrow int * .T$

Says

We've seen "(" of $T \rightarrow (E)$

We've seen ε of $E \rightarrow T$

We've seen $int *$ of $T \rightarrow int * T$

Recognizing Viable Prefixes

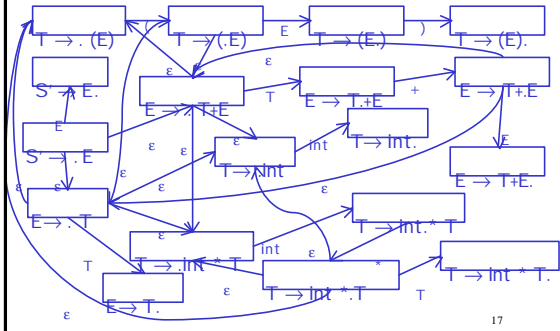
Idea: To recognize viable prefixes, we must

- Recognize a sequence of partial rhs's of productions, where
- Each sequence can eventually **reduce** to part of the missing suffix of its predecessor.

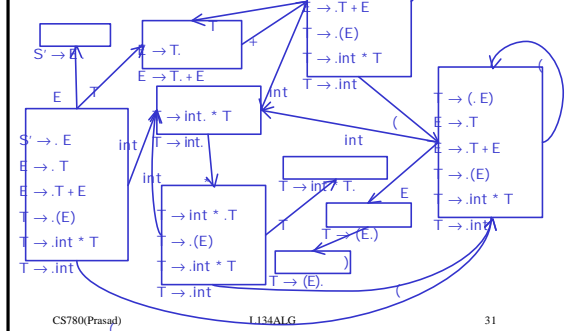
An NFA Recognizing Viable Prefixes

1. Add a dummy production $S' \rightarrow S$ to G
2. The NFA states are the items of G
 - Including the extra production
3. For item $E \rightarrow \alpha.X\beta$ add transition $E \rightarrow \alpha.X\beta \xrightarrow{X} E \rightarrow \alpha X\beta$
4. For item $E \rightarrow \alpha.X\beta$ and production $X \rightarrow \gamma$ add $E \rightarrow \alpha.X\beta \xrightarrow{\epsilon} X \rightarrow \gamma$
5. Every state is an accepting state
6. Start state is $S' \rightarrow .S$

NFA for Viable Prefixes of the Example



Translation to the DFA



Valid Items

An item $X \rightarrow \beta \cdot \gamma$ is *valid* for a viable prefix $\alpha\beta$ if $S' \Rightarrow^* \alpha X \omega \rightarrow \alpha\beta\gamma\omega$ by a right-most derivation.

Intuition: The valid items are the prefixes of productions we might see after α

Items Valid for a Prefix

An item I is valid for a viable prefix α if the DFA recognizing viable prefixes terminates on input α in a state containing I .

CS780(Prasad)

L134ALG

32

Valid Items Example

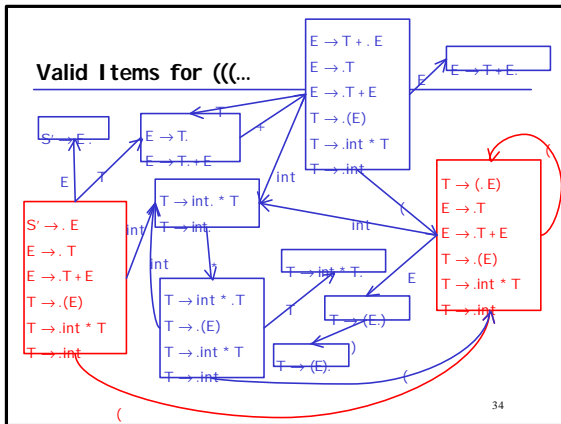
- An item is often valid for many prefixes.
- Example:** The item $T \rightarrow (\cdot E)$ is valid for prefixes

(
((
((
(((
...

CS780(Prasad)

L134ALG

33



34

Lingo

The states of the DFA are
"canonical collections of items"

or

"canonical collections of LR(0) items"

(There are other ways of constructing LR(0) items.)

CS780(Prasad)

L134ALG

35

SLR Parsing

- LR = "Left-to-right scan"
- SLR = "Simple LR"
- Idea:** Assume
 - stack contains $\alpha\beta$
 - next input is t
 - DFA on input $\alpha\beta$ terminates in state s

CS780(Prasad)

L134ALG

36

SLR Moves

- Reduce** by $X \rightarrow \beta$ if
 - s contains item $X \rightarrow \beta$.
 - $t \in \text{Follow}(X)$
- Shift** if
 - s has a transition labeled t
- If there are conflicts under these rules, the grammar is not SLR.
- The rules amount to a heuristic for detecting handles.
 - The SLR grammars are those where the heuristics detect exactly the handles.

CS780(Prasad)

L134ALG

37

Naive SLR Parsing Algorithm

- Let M be DFA for viable prefixes of G .
- Let $|x_1 \dots x_n \$$ be the initial configuration.
- Repeat until configuration is $S | \$$
 - Let $\alpha | \omega$ be current configuration.
 - Run M on current stack α
 - If M rejects α , report parsing error
 - Stack α is not a viable prefix
 - If M accepts α with items I , let a be next input
 - Shift if $X \rightarrow \beta . a \gamma \in I$
 - Reduce if $X \rightarrow \beta . \in I$ and $a \in \text{Follow}(\alpha)$
 - Report parsing error if neither applies

CS780(Prasad)

L134ALG

38

Notes

- If there is a conflict in the last step, grammar is not SLR(k).
- k is the amount of lookahead
 - In practice, $k = 1$.

CS780(Prasad)

L134ALG

39

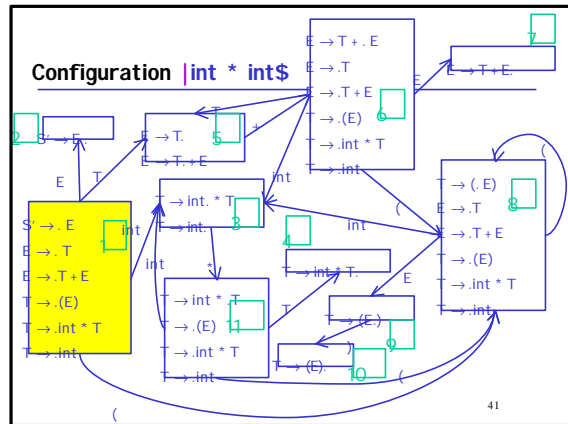
SLR Example

Configuration	DFA	Halt State	Action
$ int * int \$$	1		shift

CS780(Prasad)

L134ALG

40



41

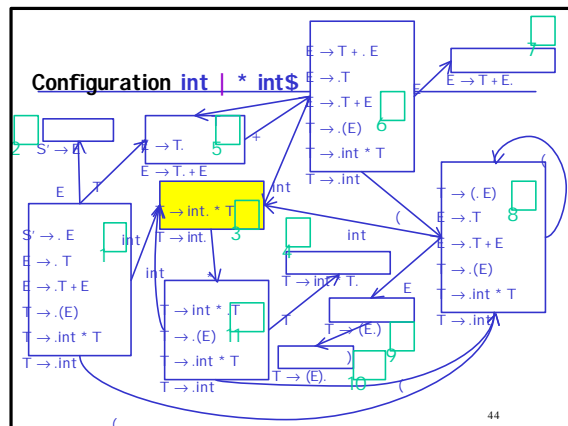
SLR Example

Configuration	DFA	Halt State	Action
$ int * int \$$	1		shift
$int * int \$$	3	* not in Follow(T)	shift

CS780(Prasad)

L134ALG

42



44

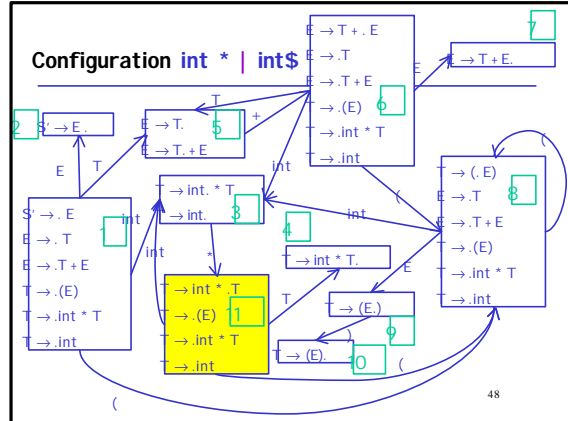
SLR Example

Configuration	DFA	Halt State	Action
int * int\$	1		shift
int * int\$	3	* not in Follow(T)	shift
int * int\$	11		shift

CS780(Prasad)

L134ALG

45



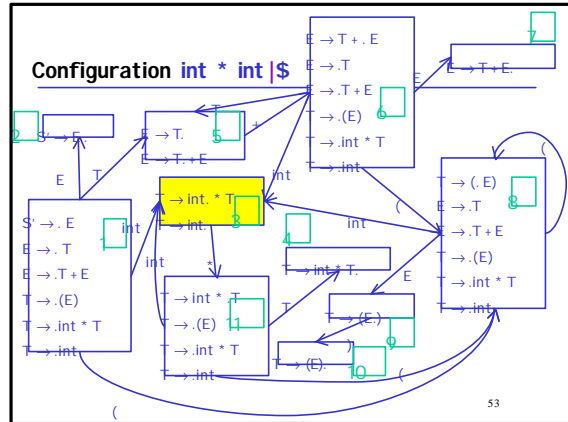
SLR Example

Configuration	DFA	Halt State	Action
int * int\$	1		shift
int * int\$	3	* not in Follow(T)	shift
int * int\$	11		shift
int * int \$	3	\$ ∈ Follow(T)	red. T → int

CS780(Prasad)

L134ALG

49



SLR Example

Configuration	DFA	Halt State	Action
int * int\$	1		shift
int * int\$	3	* not in Follow(T)	shift
int * int\$	11		shift
int * int \$	3	\$ ∈ Follow(T)	red. T → int
int * T \$	4	\$ ∈ Follow(T)	red. T → int*T

CS780(Prasad)

L134ALG

54

SLR Example

Configuration	DFA	Halt State	Action
int * int\$	1		shift
int * int\$	3	* not in Follow(T)	shift
int * int\$	11		shift
int * int \$	3	\$ ∈ Follow(T)	red. T → int
int * T \$	4	\$ ∈ Follow(T)	red. T → int*T
T \$	5	\$ ∈ Follow(T)	red. E → T

CS780(Prasad)

L134ALG

59

SLR Example

Configuration	DFA	Halt State	Action
int * int\$	1		shift
int * int\$	3	* not in Follow(T)	shift
int * int\$	11		shift
int * int \$	3	\$ ∈ Follow(T)	red. T → int
int * T \$	4	\$ ∈ Follow(T)	red. T → int*T
T \$	5	\$ ∈ Follow(T)	red. E → T
E \$			accept

62

Notes

- Skipped using extra start state S' in this example to save space on slides.
- Rerunning the automaton at each step is wasteful
 - Most of the work is repeated.

An Improvement

- Remember the state of the automaton on each prefix of the stack.
- Change stack to contain pairs
(Symbol, DFA State)

CS780(Prasad)

L134ALG

63

An Improvement (Cont.)

- For a stack
(sym₁, state₁) ... (sym_n, state_n)
state_n is the final state of the DFA on sym₁ ... sym_n
- **Detail:** The bottom of the stack is (any, start)
where
 - any is any dummy symbol
 - start is the start state of the DFA

CS780(Prasad)

L134ALG

64

Goto Table

- Define $Goto[i,A] = j$ if state_i →^A state_j
- **Goto** is just the transition function of the DFA
 - One of two parsing tables.

CS780(Prasad)

L134ALG

65

Refined Parser Moves

- **Shift x**
 - Push (a, x) on the stack
 - a is current input
 - x is a DFA state
- **Reduce X → α**
 - As before
- **Accept**
- **Error**

CS780(Prasad)

L134ALG

66

Action Table

- For each state s_i and terminal a
- If s_i has item $X \rightarrow \alpha.a\beta$ and $Goto[i,a] = j$ then
 $action[i,a] = shift\ j$
 - If s_i has item $X \rightarrow \alpha.$ and $a \in Follow(X)$ and $X \neq S'$ then
 $action[i,a] = reduce\ X \rightarrow \alpha$
 - If s_i has item $S' \rightarrow S.$ then $action[i,$] = accept$
 - Otherwise, $action[i,a] = error$

CS780(Prasad)

L134ALG

67

SLR Parsing Algorithm

```
Let I = w$ be initial input
Let j = 0
Let DFA state 1 have item S' → .S
Let stack = ( dummy, 1 )
repeat
  case action[top_state(stack), I[j]] of
  shift k: push ( I[j++], k )
  reduce X → A:
    pop |A| pairs,
    push (X, Goto[X, top_state(stack)])
  accept: halt normally
  error: halt and report error
```

CS780(Prasad)

L134ALG

68

Notes on SLR Parsing Algorithm

- Note that the algorithm uses only the DFA states and the input
 - The stack symbols are never used!
- However, we still need the symbols for semantic actions.

CS780(Prasad)

L134ALG

69