Bottom-Up Parsing

CS 132: Compiler Design
January 29, 2001

Review: Predictive Parsing

Prediction Stack	Input Stream	Action
S	a a b b	Predict S→aB
a B	a a b b	Match
B	a b b	Predict B→aBb
a B b	a b b	Match
B b	b b	Predict B→b
b b	b b	Match
b	b	Match
success		

$S \rightarrow a\ B$
$B \rightarrow b\ |\ a\ B\ b$

What to Predict?

- Assume the current prediction is $A\beta$, the next input symbol is `t`, and $A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$
- New prediction will be on of $\alpha_1\beta, \ldots, \alpha_n\beta$
- So,
  - if $t \in \text{First}(\alpha_1\beta)$ then predict $A \rightarrow \alpha_1$
  - if $t \in \text{First}(\alpha_2\beta)$ then predict $A \rightarrow \alpha_2$, and so on.
- A grammar is LL(1) if we *always* get a unique prediction (i.e., these sets are always disjoint).
  - Would prefer to avoid computing First sets at run-time
    - But in general these depend on β, which is only known during parsing
  - How can we tell statically whether a grammar is LL(1)?

Static Predictions

- Consider the sets $\mathit{First}(\alpha_1\beta), \ldots, \mathit{First}(\alpha_n\beta)$
  - $\mathit{First}(\alpha_i\beta) = \mathit{First}(\alpha_i)$ if α_i is not nullable
  - In an LL(1) grammar, at most one α_i can be nullable
    - Otherwise prediction is not unique
    - Hence parse tree is not unique and grammar is ambiguous.
  - So there's at most one set $\mathit{First}(\alpha_m\beta)$ that we can't know when building the parser (the one where α_m is nullable)

- What do we know about $\mathit{First}(\alpha_m\beta)$?

Static Predictions

- Assume α_m is nullable. Then
  - *First*($\alpha_m\beta$) = *First*(α_m) $\cup$ *First*(β)
  - *First*(β) $\subseteq$ *Follow*(A) [since $S \to^* A\beta$]
- Hence *First*($\alpha_m\beta$) $\subseteq$ *First*(α_m) $\cup$ *Follow*(A)
  - Idea: use this as an approximation.
- New algorithm: Define

$$P_i := First(\alpha_i) \quad \text{if } \alpha_i \text{ is not nullable}$$

$$P_i := First(\alpha_i) \cup Follow(A) \quad \text{if } \alpha_i \text{ is nullable}$$

  - These sets can be computed
  - Choose the production $A \to \alpha_i$ if lookahead token $t \in P_i$.

Building a Predictive Parser

- Given $A \to \alpha_1 | \ldots | \alpha_n$, define the function:

```
fun A() =
   if (!tok ∈ FIRST(α1)) or
      (nullable(α1) and (!tok ∈ FOLLOW(A)) then
       ...match α1 against input stream...
   else if (!tok ∈ FIRST(α2)) or
      (nullable(α2) and (!tok ∈ FOLLOW(A)) then
       ...match α2 against input stream...
   ...
   else if (!tok ∈ FIRST(αn)) or
      (nullable(αn) and (!tok ∈ FOLLOW(A)) then
       ...match αν against input stream...
   else ERROR
```

Is a Grammar LL(1) ?

- A grammar is Strong-LL(1) if and only if the following conditions hold for any pair of distinct productions $A \rightarrow \alpha_1$ and $A \rightarrow \alpha_2$.

  1. *First*(α_1) and *First*(α_2) are disjoint.
  2. At most one of α_1 and α_2 are nullable.
  3. If α_1 is nullable, then *First*(α_2) is disjoint from *Follow*(A).

- Theorem
  – A grammar is LL(1) if and only if it is Strong-LL(1).

Bottom-Up Parsing

- Also called shift-reduce parsing.
- More general than top-down (predictive) parsing.
  – Handles non-LL(1) grammars.
- However, also more complex
  – Impractical to write a shift-reduce parser by hand
  – Implementations generally use parser generators

Shift-Reduce Example

```
S  → E $
E  → ( E ) | E' | E' + E
E' → int * E' | int
```

Stack	Input Stream	Action
	int * int + int $	Shift
int	* int + int $	Shift
int *	int + int $	Shift
int * int	+ int $	Reduce E' → int
int * E'	+ int $	Reduce E' → int * E'
E'	+ int $	Shift
E' +	int $	Shift
E' + int	$	Reduce E' → int
E' + E'	$	Reduce E → E'
E' + E	$	Reduce E → E' + E
E	$	Shift
E $		Reduce S → E $
S		

Rightmost Reductions

- The derivation on the previous slide is a rightmost derivation, constructed in reverse order.

```
E ® E' + E
  ® E' + E'
  ® E' + int
  ® int * E' + int
  ® int * int + int
```

Why Didn't We Get Stuck?

```
S  → E $
E  → ( E ) | E' | E' + E
E' → int * E' | int
```

Stack	Input Stream	Action
	`int * int + int`	Shift
`int`	`* int + int`	**Reduce E' ® int**
`E'`	`* int + int`	...

Handles

- We want to reduce only when the resulting sentential form can still be reduced to the start symbol S.
- A *handle* is a production $B \rightarrow \beta$ and a position in a sentential form $\alpha\beta w$ (where w contains only nonterminals) such that $S \rightarrow^* \alpha B w \rightarrow \alpha\beta w$ by a rightmost derivation.
- "Algorithm" for shift-reduce parsing:
  1. if there is no handle on top of the stack, shift
  2. if there is a handle on top of the stack, reduce.
- Because we want a rightmost derivation, sufficient to look for handles only on top of the stack.

Detecting Handles

- Just because the RHS of a production occurs on the stack doesn't mean it is a handle.
  – Necessary, but not sufficient condition.
  – Fact: if the grammar is unambiguous then every sentential form has at most one handle.

Viable Prefixes

- A *viable prefix* is a string $\alpha\beta$ such that

$$S \rightarrow^{+} \alpha B w \quad \rightarrow \alpha\beta w$$

  is a rightmost derivation for some *w* (where *w* contains only nonterminals).
- Facts about viable prefixes:
  – If the parser's stack contains a viable prefix then no error can have been detected yet.
  – If we can find a viable prefix then we might have a handle.

Detecting Viable Prefixes

- What does a viable prefix look like?
  - Contains bits of the right-hand sides of rewrite rules, followed by the complete right-hand side of a rewrite rule.
- How can we detect viable prefixes?
  - The viable prefixes of a CFG form a regular language!
  - This is called the *characteristic language* of the grammar.

Items

- An *LR(0) item* (or just *item*) is a production rule from a grammar with a new symbol "." somewhere on the right-hand side.

```
E  →  . ( E )
E  →  ( . E )
E  →  ( E . )
E  →  ( E ) .
```

- Important: If the grammar already has "." as a terminal symbol, choose some other marker

Intuition about Items

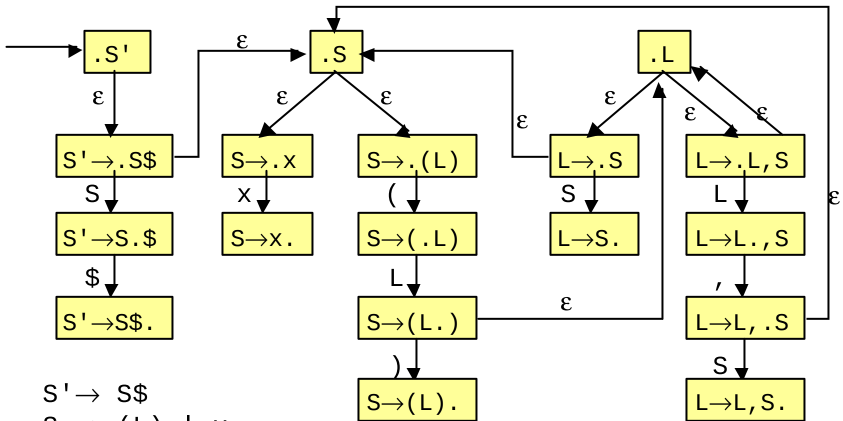
- An item like $E \rightarrow . (E)$ means that expect to see input matching (E)
- An item like $E \rightarrow (. E)$ means that we have seen (and are expecting input matching $E)$ to follow.
- An item like $E \rightarrow (E) .$ means that we have just seen input that can be reduced to E.

Recognizing Viable Prefixes

Algorithm for constructing an NFA:

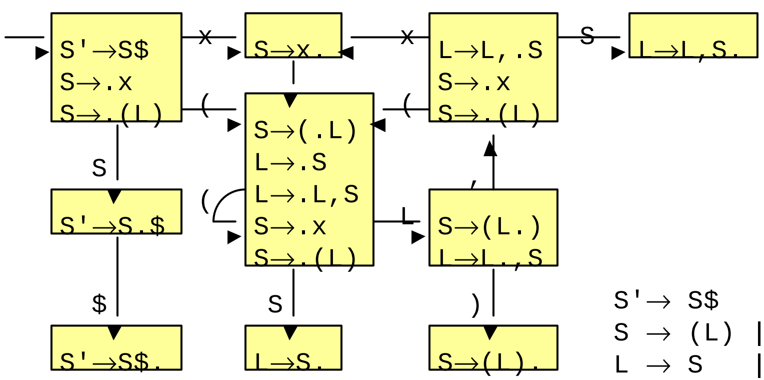
1. Add a dummy production $S' \rightarrow S\$$ to the grammar, and make S' the start symbol.
2. The NFA states are the items of the grammar.
3. For each item $E \rightarrow \alpha . z\beta$ add an edge labeled z from this item to $E \rightarrow \alpha z . \beta$ (where z is a terminal or a nonterminal)
4. For each item $E \rightarrow \alpha . X\beta$ and rule $X \rightarrow \gamma$, add an ε-edge from this item to $X \rightarrow . \gamma$

NFA Example: LR(0)



S'→ S$
S → (L) | x
L → S | L,S

Corresponding DFA



S'→ S$
S → (L) | x
L → S | L,S

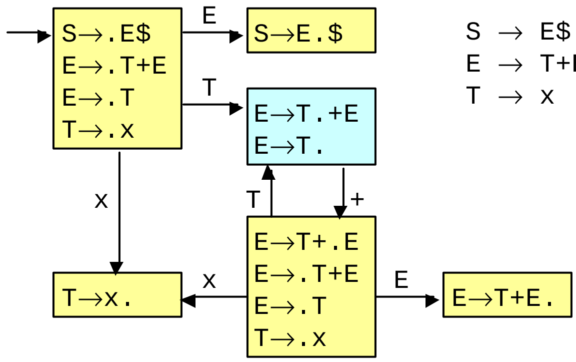
LR(0) Parsing Algorithm

- Let α be the current stack.
  - If α is simply `S'` then we're done.
  - Otherwise, run the DFA on α (the stack).
    - If it rejects, we've detected an error.
    - If the resulting state contains an item $E \to \alpha.$ then reduce, and repeat.
    - If the resulting state contains an item with . in the middle, then shift, and repeat.
- A grammar is LR(0) if this algorithm is deterministic.

A Non-LR(0) Grammar

$S \to E\$$
$E \to T{+}E \mid T$
$T \to x$

LR(0) DFA

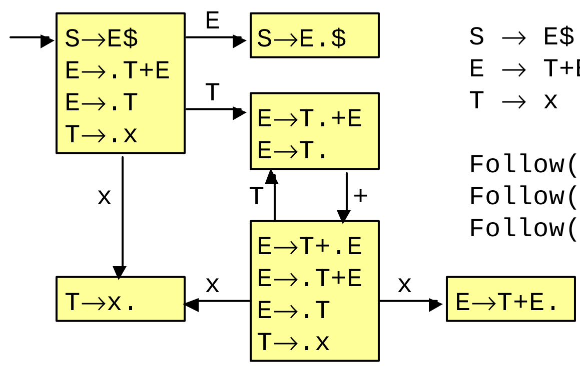


S → E$
E → T+E | T
T → x

SLR(1) Algorithm

- Uses the same LR(0) automaton, but parsing differs.
- Let α be the current stack and `t` the next input token.
  - If we've reduced the entire input to the start symbol we're done.
  - Otherwise, run the DFA on α (the stack).
    - If it rejects, we've detected an error.
    - If the resulting state contains an item $E \to \alpha.$ and $t \in$ Follow(E) then reduce, and repeat.
    - If the resulting state contains an item containing `.t` then shift, and repeat.
    - Otherwise, an error has been detected
- A grammar is SLR(1) if this algorithm is deterministic.

DFA: SLR(1)



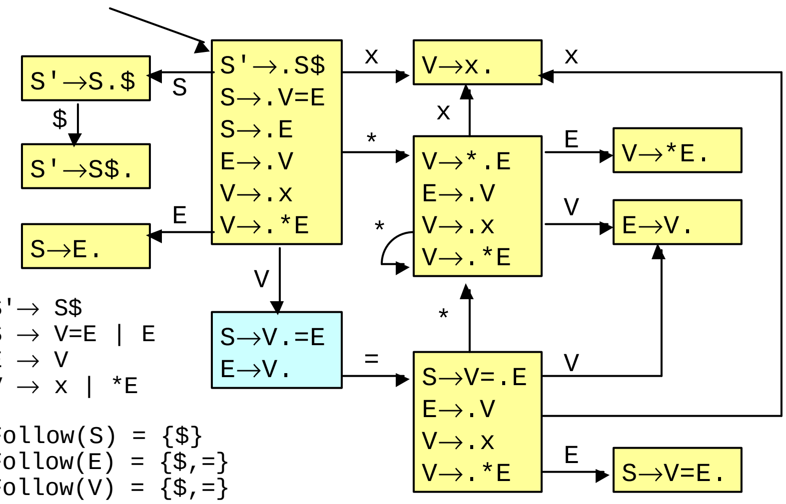
S → E$
E → T+E | T
T → x

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

A Non-SLR(1) Grammar

S'→ S$
S → V=E | E
E → V
V → x | *E

DFA: not SLR(1)




```
S'→ S$
S → V=E | E
E → V
V → x | *E

Follow(S) = {$}
Follow(E) = {$,=}
Follow(V) = {$,=}
```

LR(1) Parsing

- An *LR(1) item* is a pair consisting of an LR(0)-item and a terminal symbol.
- An item like [E→(.E),a] means that we have seen (and are expecting input matching E) a to follow.
- An item like [E→(E).,a] means that we have just seen input that should be reduced to E *if* the next input token is a.

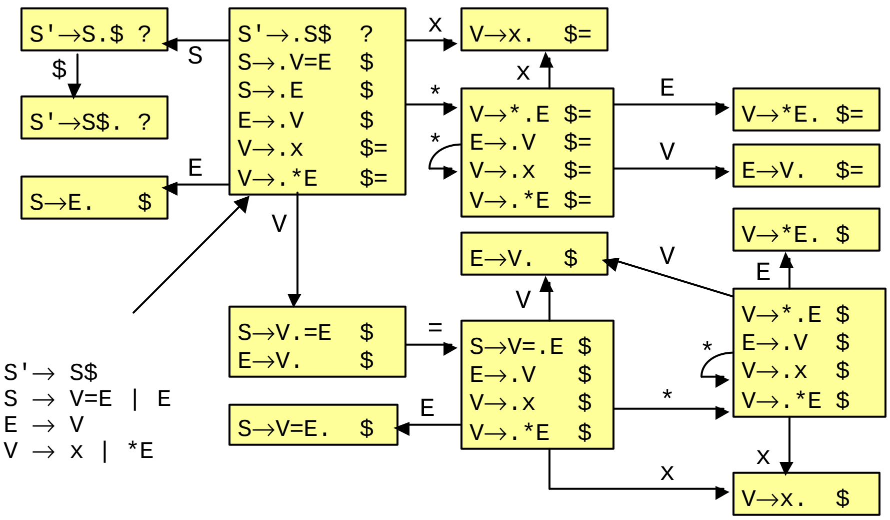
LR(1) Automaton

- Algorithm for NFA construction:
  1. Add a dummy production $S' \to S\$$ to the grammar, and make S' the start symbol.
  2. The states are the LR(1) items of the grammar.
  3. For each item $[E \to \alpha . z\beta, a]$ add an edge labeled z from this item to $[E \to \alpha z . \beta, a]$ (where z is a terminal or nonterminal).
  4. For each item $[E \to \alpha . X\beta, a]$ and rule $X \to \gamma$ and terminal $b \in \mathtt{First}(\beta a)$ add an ε-edge from this item to $[X \to . \gamma, b]$.
  5. The start state is $[S' \to . S, \$]$

LR(1) Algorithm

- Let α be the current stack and t the next input token.
  - If we've reduced the entire input to the start symbol: success.
  - Otherwise, run the DFA on α (the stack)
    - If it rejects, we've detected an error.
    - If the resulting state contains an item $[E \to \alpha ., t]$ then reduce, and repeat.
    - If the resulting state contains an item containing $.t$ then shift, and repeat.
    - Otherwise, an error has been detected
- A grammar is LR(1) if this algorithm is deterministic.

DFA: LR(1)



S'→ S$
S → V=E | E
E → V
V → x | *E

SLR(1) vs. LR(1)

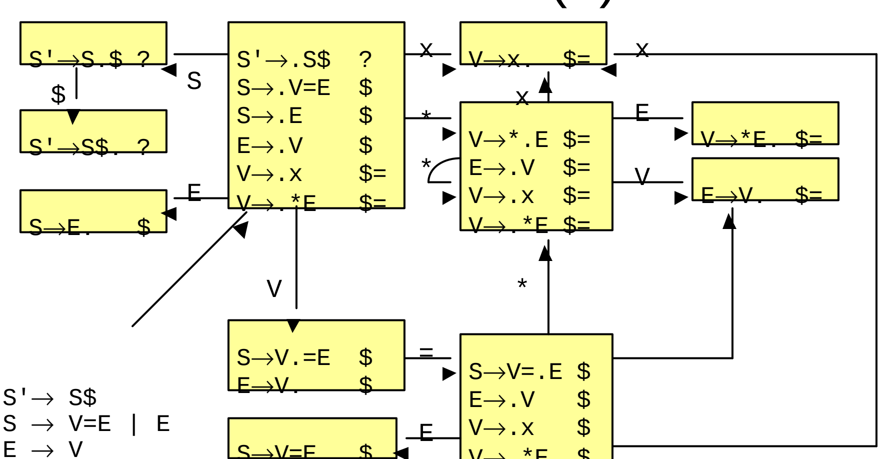
- SLR(1)
  - Advantage: 100's instead of 1000's of states
- LR(1)
  - Advantage: more powerful; every SLR(1) grammar is also LR(1) but not vice versa.

- Which is used in practice?

LALR(1)

- Look-Ahead LR(1)
  - Take the LR(1) automaton and merge all states that contain the same sets of LR(0) items
  - i.e., merge LR(1) states that differ only in the lookahead tokens.
- Advantages
  - Size of LR(0) or SLR(1) automata
  - Includes most of the LR(1) grammars

DFA: LALR(1)



S' → S$
S → V=E | E
E → V
V → x | *E