

Shift-reduce Parsing

- The last lecture introduced shift-reduce parsing for doing rightmost derivations in reverse.
- Split input into
 - A *stack* of processed input (terminals and non-terminals)
 - A suffix containing only non-terminals. First character in the suffix is called the *lookahead*.
- Perform *reduction* (reverse production) at end of stack (to produce a new **rightmost** non-terminal) if the stack is a *handle* ...
- ...or *shift* the lookahead to the end (top) of the stack.
- To decide which to do, build a FSA to run over the stack and recognize handles.

Recognizing Handles

- We will build a *grammar* for handles.
- **FACT: this grammar will be regular!**
- So we can build a FSA for it.
- Sample grammar (⊥ is end of input, p is start):

- A. $p \rightarrow e \perp$
- B. $e \rightarrow e '+' t$
- C. $e \rightarrow t$
- D. $t \rightarrow '(' e ')'$
- E. $t \rightarrow i$

A Handle Grammar

- A *handle for B* (B non-terminal) is the part of a rightmost sentential form created by expanding a B up to and including that expansion.
- What does it look like? Let's consider symbol p .
- After first step of a rightmost derivation (or last step of reverse derivation), stack is ' $e \perp$ ' (a handle)
- So add

$$H_p \rightarrow e \perp$$

- **In this grammar, e , t , and p as well as $'+'$, etc. are all terminals.**
- H_p means "handle occurring during derivation from p ."
- But this means that at a later step, e will be expanded.
- So also add

$$H_p \rightarrow H_e$$

where H_e is "handle occurring during derivation from e ."

Handle Grammar II

- What about H_e ? Obviously, we first have

$$\begin{aligned} H_e &\rightarrow e '+' t \\ H_e &\rightarrow t \end{aligned}$$

- First production implies that at later step

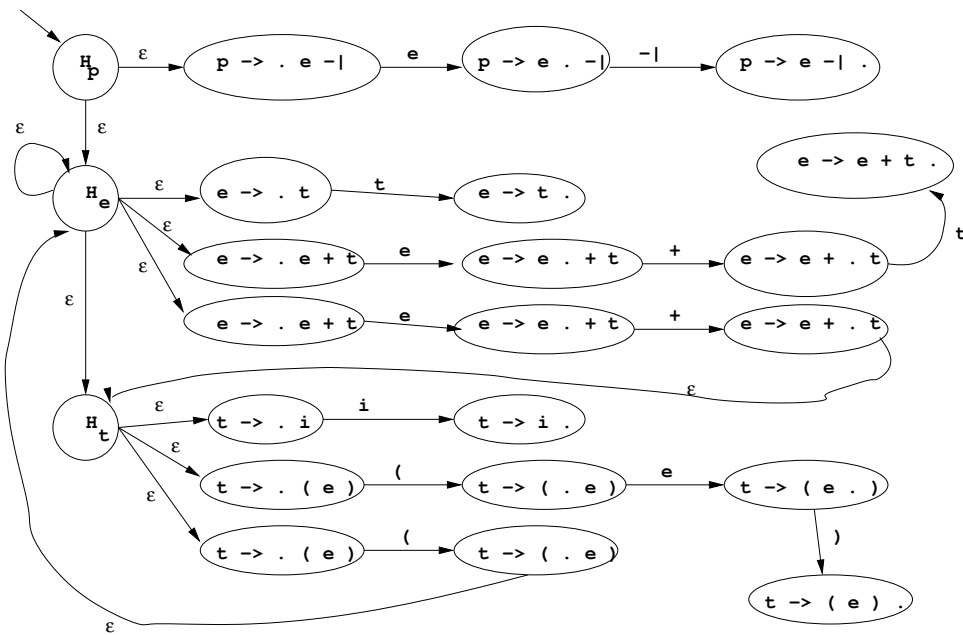
$$\begin{aligned} H_e &\rightarrow e '+' H_t \\ H_e &\rightarrow H_e \quad (\text{useless, but harmless}) \end{aligned}$$

- Likewise for H_t :

$$\begin{aligned} H_t &\rightarrow '(' e ')'' \\ H_t &\rightarrow '(' H_e \\ H_t &\rightarrow i \end{aligned}$$

- This handle grammar is *regular*, (any non-terminals are always at the end) so can produce NFA.

NFA for Handle Grammar



Last modified: Fri Feb 18 00:48:26 2005

CS164: Lecture #13 5

Items and Item sets

- The state labels on the NFA are called *LR(0) items*.
- For example:

$$e \rightarrow e ' + ' \cdot t$$

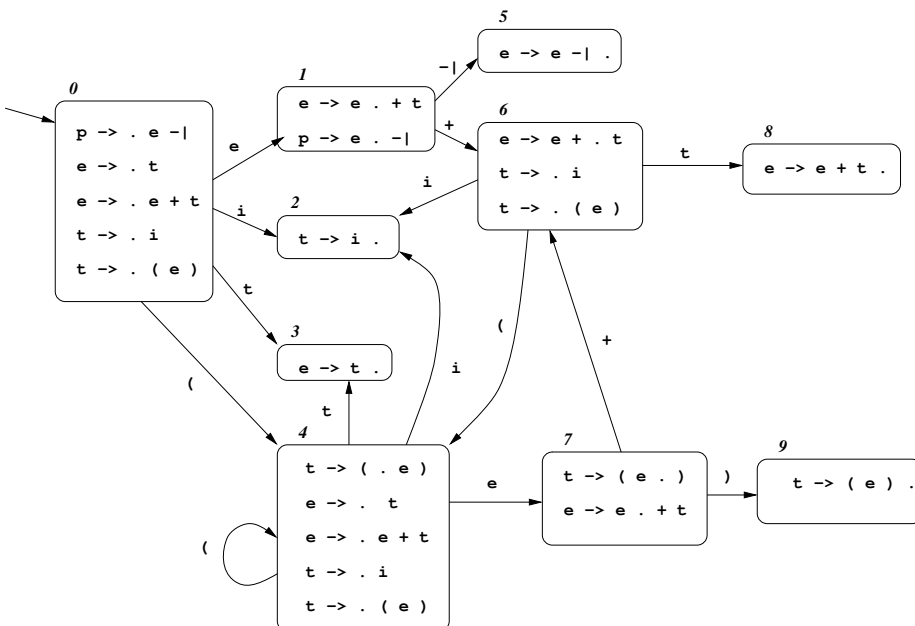
means "the state of being in a sentential form immediately after an "e '+'" that might be part of the end of a handle producing e."

- We can convert the NFA to a DFA using set-of-states construction we showed before.
- The labels on the DFA states are then *item sets*.

Last modified: Fri Feb 18 00:48:26 2005

CS164: Lecture #13 6

DFA for Handle Grammar



Last modified: Fri Feb 18 00:48:26 2005

CS164: Lecture #13 7

Interpretation

- An item like

$$t \rightarrow '(e ') \cdot$$

means "if scanning the stack ends you up here, you could reduce the top three symbols to a 't'."

- An item like

$$e \rightarrow e \cdot ' + ' t$$

means "if scanning the stack ends you up here, and the lookahead is '+' , you could shift it."

- If there are no choices anywhere, grammar is LR(0) (rare).
- Otherwise, must decide in places between shifting and reducing or between several possible reductions.

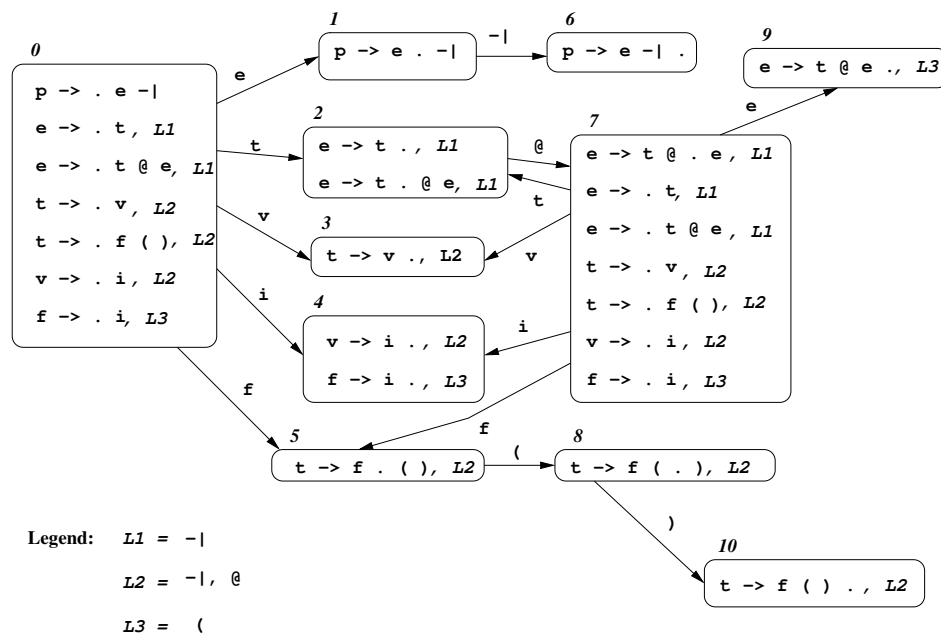
Last modified: Fri Feb 18 00:48:26 2005

CS164: Lecture #13 8

Deciding Between Shifts and Reductions

- At states with a reduction item (dot at the end) and a transition out, there *might* be strings in the language allowing either (LR(0) shift-reduce conflict).
- At states with two reduction items, there *might* be strings in the language allowing either (LR(0) reduce-reduce conflict).
- If the lookahead symbol suffices to resolve conflict, the grammar is **LALR(1)** (LookAhead LR(1)). If *some* DFA can make the decision with on symbol of lookahead, the grammar is **LR(1)**.
- Simple way to decide (SLR(1)): For an item like "e → e '+' t.", take this reduction if lookahead is in FOLLOW(e).
- Works if FOLLOW sets for a state don't overlap and don't contain any of the shift symbols.
- Full LALR(1) strategy is more refined: compute specialized FOLLOW sets for each state by looking backwards in the machine.

The LALR(1) machine



LR(0) Machine for Last Lecture's Grammar

