450 COMPILERS

COMPUTER SCIENCE

News & Info

- Who's Hiring May 2016
  - https://news.ycombinator.com/item?id=11611867
- SoCal Code Camp | San Diego, CA 6/25-6/26
  - http://www.socalcodecamp.com/

Administrivia

- Lab 06
  - Due Thursday

Review: Shift Reduce Parsing

Bottom-up parsing uses two actions:

Shift

ABC|xyz ⇒ ABCx|yz

Reduce

Cbxy|ijk ⇒ CbA|ijk

Recall: The Stack

- Left string can be implemented by a stack
  - Top of the stack is the |
- Shift pushes a terminal on the stack
- Reduce
  - pops 0 or more symbols off of the stack
    - production rhs
  - pushes a non-terminal on the stack
    - production lhs

Key Issue

- How do we decide when to shift or reduce?
- Example grammar:

  $E \rightarrow T + E \mid T$

  $T \rightarrow \text{int} * T \mid \text{int} \mid (E)$

- Consider step int | * int + int
  - We could reduce by $T \rightarrow \text{int}$ giving T | * int + int
  - A fatal mistake!
    - No way to reduce to the start symbol E

Handles

- Intuition: Want to reduce only if the result can still be reduced to the start symbol

- Assume a rightmost derivation

$$S \rightarrow^* \alpha X \omega \rightarrow \alpha\beta\omega$$

- Then $X \rightarrow \beta$ in the position after α is a *handle* of $\alpha\beta\omega$

Handles continued

- Handles formalize the intuition
  - A handle is a string that can be reduced and also allows further reductions back to the start symbol (using a particular production at a specific spot)

- We only want to reduce at handles

- Note: We have said what a handle is, not how to find handles

Important Fact #2

Important Fact #2 about bottom-up parsing:

In shift-reduce parsing, handles appear only at the top of the stack, never inside

Why?

- Informal induction on # of reduce moves:
- True initially, stack is empty
- Immediately after reducing a handle
  - right-most non-terminal on top of the stack
  - next handle must be to right of right-most non-terminal, because this is a right-most derivation
  - Sequence of shift moves reaches next handle

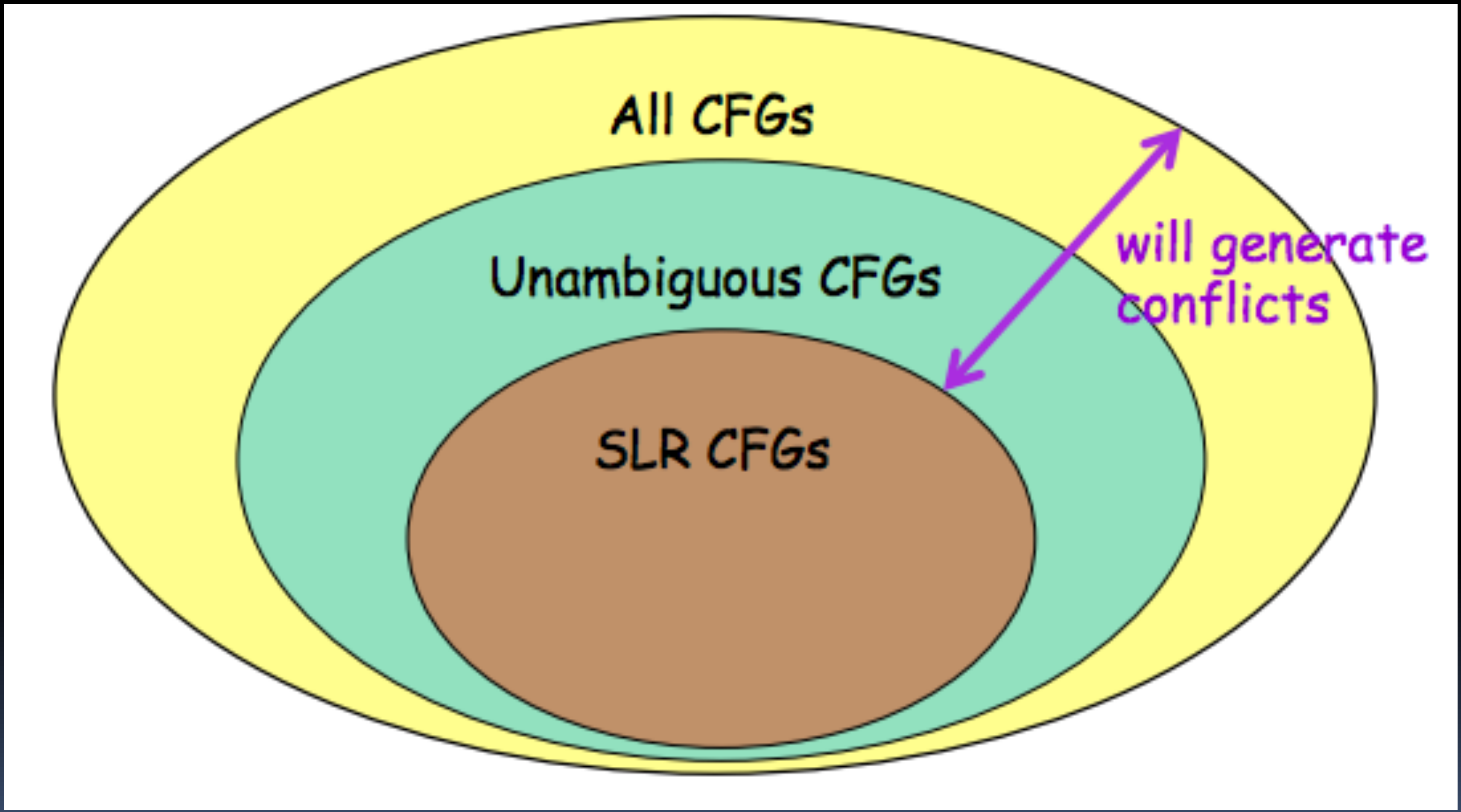
Summary of Handles

- In shift-reduce parsing, handles always appear at the top of the stack
- Handles are never to the left of the rightmost non-terminal
  - Therefore, shift-reduce moves are sufficient; the | need never move left
- Bottom-up parsing algorithms are based on recognizing handles

Recognizing Handles

- There are no known efficient algorithms to recognize handles
- Solution: use heuristics to guess which stacks are handles
- On some CFGs, the heuristics always guess correctly
  - For the heuristics we use here, these are the SLR grammars
  - Other heuristics work for other grammars

Grammars


All CFGs
Unambiguous CFGs
SLR CFGs
will generate conflicts

Viable Prefixes

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

α is a viable prefix if there is an ω such that

$\alpha | \omega$ is a state of a shift-reduce parser

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
  - As long as a parser has viable prefixes on the stack no parsing error has been detected

Important Fact #3

Important Fact #3 about bottom-up parsing:

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

Items

- An item is a production with a "." somewhere on the rhs

- The items for $T \rightarrow (E)$ are
  $T \rightarrow .(E)$
  $T \rightarrow (.E)$
  $T \rightarrow (E.)$
  $T \rightarrow (E).$

Items continued

- The only item for $X \rightarrow \varepsilon$ is $X \rightarrow .$

- Items are often called "LR(0) items"

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

Example

Consider the input (int)

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

Generalization

- The stack may have many prefixes of rhs's
  $Prefix_1\ Prefix_2 \ldots Prefix_{n-1} Prefix_n$
- Let $Prefix_i$ be a prefix of rhs of $X_i \rightarrow \alpha_i$
  - $Prefix_i$ will eventually reduce to X_i
  - The missing part of α_{i-1} starts with X_i
  - i.e. there is a $X_{i-1} \rightarrow Prefix_{i-1}\ X_i\ \beta$ for some β
- Recursively, $Prefix_{k+1} \ldots Prefix_n$ eventually reduces to the missing part of α_k

Example

Consider the string (int * int):

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

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

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

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

Example continued

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

$$E \rightarrow \alpha.X\beta \;\rightarrow^{X}\; E \rightarrow \alpha X.\beta$$

4. For item $E \rightarrow \alpha.X\beta$ and production $X \rightarrow \gamma$ add

$$E \rightarrow \alpha.X\beta \;\rightarrow^{\varepsilon}\; X \rightarrow .\gamma$$

An NFA Recognizing Viable Prefixes

5. Every state is an accepting state

6. Start state is $S' \rightarrow .S$

Lingo

The states of the DFA are

"canonical collections of items"

or

"canonical collections of LR(0) items"

The Dragon book gives another way of constructing LR(0) items

Valid Items

Item $X \rightarrow \beta.\gamma$ is *valid* for a viable prefix $\alpha\beta$ if

$$S' \rightarrow^* \alpha X \omega \rightarrow \alpha\beta\gamma\omega$$

by a right-most derivation

After parsing $\alpha\beta$, the valid items are the possible tops of the stack of items

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 s containing I

The items in s describe what the top of the item stack might be after reading input α

Valid Items Exampe

- An item is often valid for many prefixes

- Example: The item $T \rightarrow (.E)$ is valid for prefixes

(
((
(((
((((
. . .

LR(0) Parsing

- Idea: Assume
  - stack contains α
  - next input is t
  - DFA on input α terminates in state s
- Reduce by $X \rightarrow \beta$ if
  - s contains item $X \rightarrow \beta.$
- Shift if
  - s contains item $X \rightarrow \beta.t\omega$
  - equivalent to saying s has a transition labeled t

LR(0) Conflicts

- LR(0) has a reduce/reduce conflict if:
  - Any state has two reduce items:
  - $X \rightarrow \beta.$ and $Y \rightarrow \omega.$
- LR(0) has a shift/reduce conflict if:
  - Any state has a reduce item and a shift item:
  - $X \rightarrow \beta.$ and $Y \rightarrow \omega.t\delta$

SLR

- LR = “Left-to-right scan”
- SLR = “Simple LR”

- SLR improves on LR(0) shift/reduce heuristics
  - Fewer states have conflicts

SLR Parsing

- Idea: Assume
  - stack contains α
  - next input is t
  - DFA on input α terminates in state s
- Reduce by $X \rightarrow \beta$ if
  - s contains item $X \rightarrow \beta.$
  - $t \in \text{Follow}(X)$
- Shift if
  - s contains item $X \rightarrow \beta.t\omega$

SLR Parsing continued

- 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

Precedence Declarations Digression

- Lots of grammars aren't SLR
  - including all ambiguous grammars
- We can parse more grammars by using precedence declarations
  - Instructions for resolving conflicts

Precedence Declarations continued

- Consider our favorite ambiguous grammar:
  - E → E + E | E * E | (E) | int
- The DFA for this grammar contains a state with the following items:
  - E → E * E .   E → E . + E
  - shift/reduce conflict!
- Declaring "* has higher precedence than +" resolves this conflict in favor of reducing

Precedence Declarations continued

- The term "precedence declaration" is misleading
- These declarations do not define precedence; they define conflict resolutions
  - Not quite the same thing!

Naïve SLR Parsing Algorithm

1. Let M be DFA for viable prefixes of G
2. Let $|x_1 \ldots x_n\$$ be initial configuration
3. 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 $X \to \beta.\, a\, \gamma \in I$
     - Reduce if $X \to \beta. \in I$ and $a \in Follow(X)$
     - Report parsing error if neither applies

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