

# *CS415 Compilers*

## *Procedure Abstraction Part 4*

### *Syntax Analysis Wrap-Up*

These slides are based on slides copyrighted by  
Keith Cooper, Ken Kennedy & Linda Torczon at Rice  
University

## Last class

- Project #3 - Local Dead-Code Elimination  
Due date: Wednesday May 4
- Midterm has been graded. Please see sample solution.  
Need to ask for regrade by Wednesday, May 4
- Final exam on May 10 , 1:00pm (60 minutes in class)
  - HW#5 and HW#6
  - Parameter passing
- Grading Scheme
  - Exams: 2 x 30% ( best two exams count )
  - Projects: 3 x 10%
  - Homeworks: 5 x 2% ( best five homeworks count )

LR(1) parsing

Type systems

- type checking

Syntax-Directed translation schemes

- Yacc notation
- Second project

Code generation

- loops
- arrays

Optimizations

- local vs. global optimizations
- Third project

Procedure abstraction

- dynamic runtime stack
- non-local accesses
  - lexical scoping (access links)
  - dynamic scoping
- parameter passing

### Material to Study

- Lectures 16 through 26 (with readings)
- Homeworks #5 and #6
- Projects #2 and #3

Most languages provide a parameter passing mechanism:  
actual parameters are mapped to formal parameters

Common binding mechanisms:

- **Call-by-reference** passes a pointer to actual parameter
  - Requires slot in the AR (for **address** of parameter)
  - Expression used at "call site" becomes a variable in callee
  - Multiple names with the same address (aliasing)?

e.g: call fee(x,x,x)

- **Call-by-value** passes a copy of its value at time of call
  - Requires slot in the AR
  - Each name gets a unique location
  - Arrays are mostly passed by reference, not value

Most languages provide a parameter passing mechanism

actual parameters are mapped to formal parameters

- **Call-by-value-result** passes the value of and a pointer to the actual parameter; at the end of the call, value of formal parameter is copied back into actual parameter.
  - Requires two slots in the AR
  - During execution of procedure body, formal parameter is treated as a call-by-value parameter,
  - Order of write-back is important
- Can always use global variables, which makes reasoning about programs harder

How do procedure calls actually work?

- At compile time, callee may not be available for inspection
  - Different calls may be in different compilation units
  - Compiler may not know system code from user code
  - All calls must use the same protocol

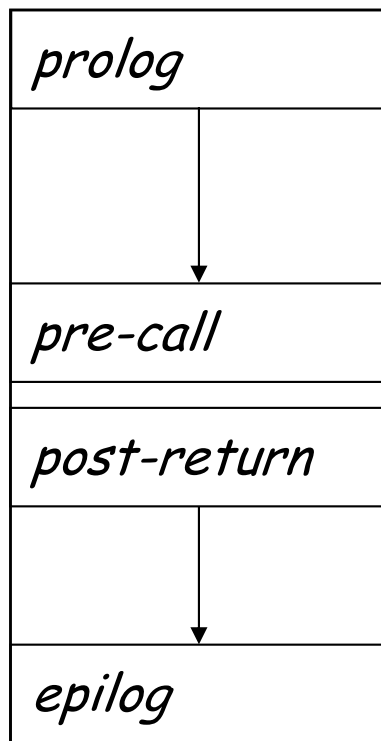
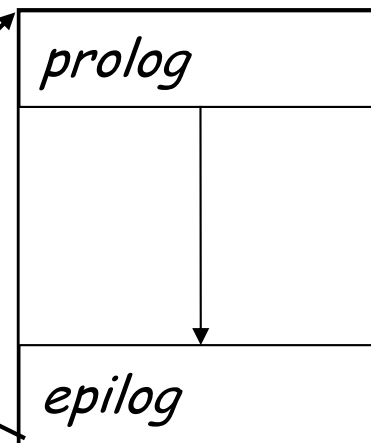
Compiler must use a standard sequence of operations

- Enforces control & data abstractions
- Divides responsibility between caller & callee

Usually a system-wide agreement

*(for interoperability)*

## Standard procedure linkage

*procedure p**procedure q*

Procedure has

- standard **prolog**
- standard **epilog**

Each call involves a

- **pre-call** sequence
- **post-return** sequence

These are completely predictable from the call site  $\Rightarrow$  depend on the number & type of the actual parameters

### Pre-call Sequence

- Sets up callee's basic AR
- Helps preserve its own environment

### The Details

- Allocate space for the callee's AR
  - except space for local variables
- Evaluates each parameter & stores value and/or address
- Saves return address, caller's ARP (control link) into callee's AR
- If access links are used
  - Find appropriate lexical ancestor & copy into callee's AR
- Save any caller-save registers
  - Save into space in caller's AR
- Jump to address of callee's prolog code

### Post-return Sequence

- Finish restoring caller's environment
- Place any value back where it belongs

### The Details

- Copy return value from callee's AR, if necessary
- Free the callee's AR
- Restore any caller-save registers
- Copy back call-by-value-result parameters
- Continue execution after the call

### Prolog Code

- Finish setting up callee's environment
- Preserve parts of caller's environment that will be disturbed

### The Details

- Preserve any callee-save registers
- If display is being used
  - Save display entry for current lexical level
  - Store current ARP into display for current lexical level
- Allocate space for local data
  - Easiest scenario is to extend the AR
- Handle any local variable initializations

With heap allocated AR, may need to use a separate heap object for local variables

### Epilog Code

- Wind up the business of the callee
- Start restoring the caller's environment

If ARs are stack allocated, this may not be necessary. (Caller can reset stacktop to its pre-call value.)

### The Details

- Store return value?
  - Some implementations do this on the return statement
  - Others have return assign it & epilog store it into caller's AR
- Restore callee-save registers
- Free space for local data, if necessary (on the heap)
- Load return address from AR
- Restore caller's ARP
- Jump to the return address

# Bottom-up Parsing (Syntax Analysis)

EAC Chapters 3.4  
ALSU Chapter 4.5

Example:

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow S ; a \mid a \end{aligned}$$

LR(0) ?  $s_0 = \{[S' \rightarrow .S], [S \rightarrow . S ; a], [S \rightarrow . a]\}$

$s_1 = \text{goto}(s_0, S) = \{[S' \rightarrow S.], [S \rightarrow S. ; a]\}$  \*\*conflict\*\*

LR(1) ? YES - check at home or in recitation

SLR(1) ? SIMPLE LR(1) FOLLOW(S) = {eof, ; }

$s_1 = \{[S' \rightarrow S., \text{eof}], [S \rightarrow S. ; a, \{\text{eof}, ;\}]\}$  \*\*no conflict\*\*

SLR(1): add FOLLOW(A) to each LR(0) item  $[A \rightarrow \gamma \cdot]$  as its second component:  $[A \rightarrow \gamma \cdot, \underline{a}], \forall a \in \text{FOLLOW}(A)$ ;

Note: Can also add to other items, but does not really matter.

1:  $S' \rightarrow S$

2:  $S \rightarrow S ; a$

3:  $S \rightarrow a$

LR(0):

$s_0 = \{[S' \rightarrow .S], [S \rightarrow . S;a], [S \rightarrow . a]\}$

$s_1 = \text{Goto}(s_0, S) = \{[S' \rightarrow S.], [S \rightarrow S . ;a]\}$

$s_2 = \text{Goto}(s_0, a) = \{[S \rightarrow a .]\}$

$s_3 = \text{Goto}(s_1, ;) = \{[S \rightarrow S; . a]\}$

$s_4 = \text{Goto}(s_3, a) = \{[S \rightarrow S; a .]\}$

LR(0) parse table

|                      |                                  |
|----------------------|----------------------------------|
| <b>s<sub>0</sub></b> | shift                            |
| <b>s<sub>1</sub></b> | shift/reduce <b>**conflict**</b> |
| <b>s<sub>2</sub></b> | reduce rule 3                    |
| <b>s<sub>3</sub></b> | shift                            |
| <b>s<sub>4</sub></b> | reduce rule 2                    |

Grammar is not LR(0)!

SLR(1)

$\text{Follow}(S') = \{\text{eof}\}$

$\text{Follow}(S) = \{\text{eof}, ;\}$

Grammar is SLR(1)!

SLR(1) parse table

|                      | <b>a</b> | <b>;</b>      | <b><u>eof</u></b>      |
|----------------------|----------|---------------|------------------------|
| <b>s<sub>0</sub></b> | shift    |               |                        |
| <b>s<sub>1</sub></b> |          | shift         | reduce rule 1 (accept) |
| <b>s<sub>2</sub></b> |          | reduce rule 3 | reduce rule 3          |
| <b>s<sub>3</sub></b> | shift    |               |                        |
| <b>s<sub>4</sub></b> |          | reduce rule 2 | reduce rule 2          |

Example:

$$S' \rightarrow S$$
$$S \rightarrow aAd \mid bBd \mid aBe \mid bAe$$
$$A \rightarrow c$$
$$B \rightarrow c$$

LR(0) ?

LR(1) ?

LALR(1) ?

**LALR(1)**: Merge two sets of LR(1) items (states), if they have the same **core**.

**core** of set of LR(1) items: set of LR(0) items derived by ignoring the lookahead symbols

$$s_0 = \text{Closure}(\{[S' \rightarrow .S, \text{eof}]\}) = \{[S \rightarrow .aAd, \text{eof}], [S \rightarrow .aBe, \text{eof}], \\ [S \rightarrow .bAe, \text{eof}], [S \rightarrow .bBd, \text{eof}], \\ [S' \rightarrow .S, \text{eof}]\}$$

$$s_1 = \text{Closure}(\text{GoTo}(s_0, a)) = \\ \{[S \rightarrow a .Ad, \text{eof}], \\ [S \rightarrow a .Be, \text{eof}], \\ [A \rightarrow .c, d], [B \rightarrow .c, e]\}$$

$$s_2 = \text{Closure}(\text{GoTo}(s_0, b)) = \\ \{[S \rightarrow b .Ae, \text{eof}], \\ [S \rightarrow b .Bd, \text{eof}], \\ [A \rightarrow .c, e], [B \rightarrow .c, d]\} \quad \dots \text{ /* other states */}$$

$$s_3 = \text{Closure}(\text{GoTo}(s_1, c)) = \\ \{[A \rightarrow c ., d], \\ [B \rightarrow c ., e]\}$$

$$s_4 = \text{Closure}(\text{GoTo}(s_2, c)) = \\ \{[A \rightarrow c ., e], \\ [B \rightarrow c ., d]\}$$

There are other states that are not listed here!  
 Grammar is LR(1), but not LALR(1), since collapsing  
 $s_3$  and  $s_4$  (same core) will introduce reduce-reduce conflict.

Example:

$$S' \rightarrow S$$
$$S \rightarrow aAd \mid bBd \mid aBe \mid bAe$$
$$A \rightarrow c$$
$$B \rightarrow c$$

LR(0) ?            NO

LR(1) ?            YES

LALR(1) ?        NO, since introduces a reduce/reduce conflict

**LALR(1)**: Merge two sets of LR(1) items (states), if they have the same **core**.

**core** of set of LR(1) items: set of LR(0) items derived by ignoring the lookahead symbols

**FACT**: collapsing LR(1) states into LALR(1) states cannot introduce shift/reduce conflicts

Three options:

- Combine terminals such as number & identifier, + & -, \* & /
  - Directly removes a column, may remove a row
  - For expression grammar, 198 (vs. 384) table entries
- Combine rows or columns *(table compression)*
  - Implement identical rows once & remap states
  - Requires extra indirection on each lookup
  - Use separate mapping for ACTION & for GOTO
- Use another construction algorithm
  - Both LALR(1) and SLR(1) produce smaller tables
  - Implementations are readily available

## Finding Reductions

$LR(k) \Rightarrow$  Each reduction in the parse is detectable with

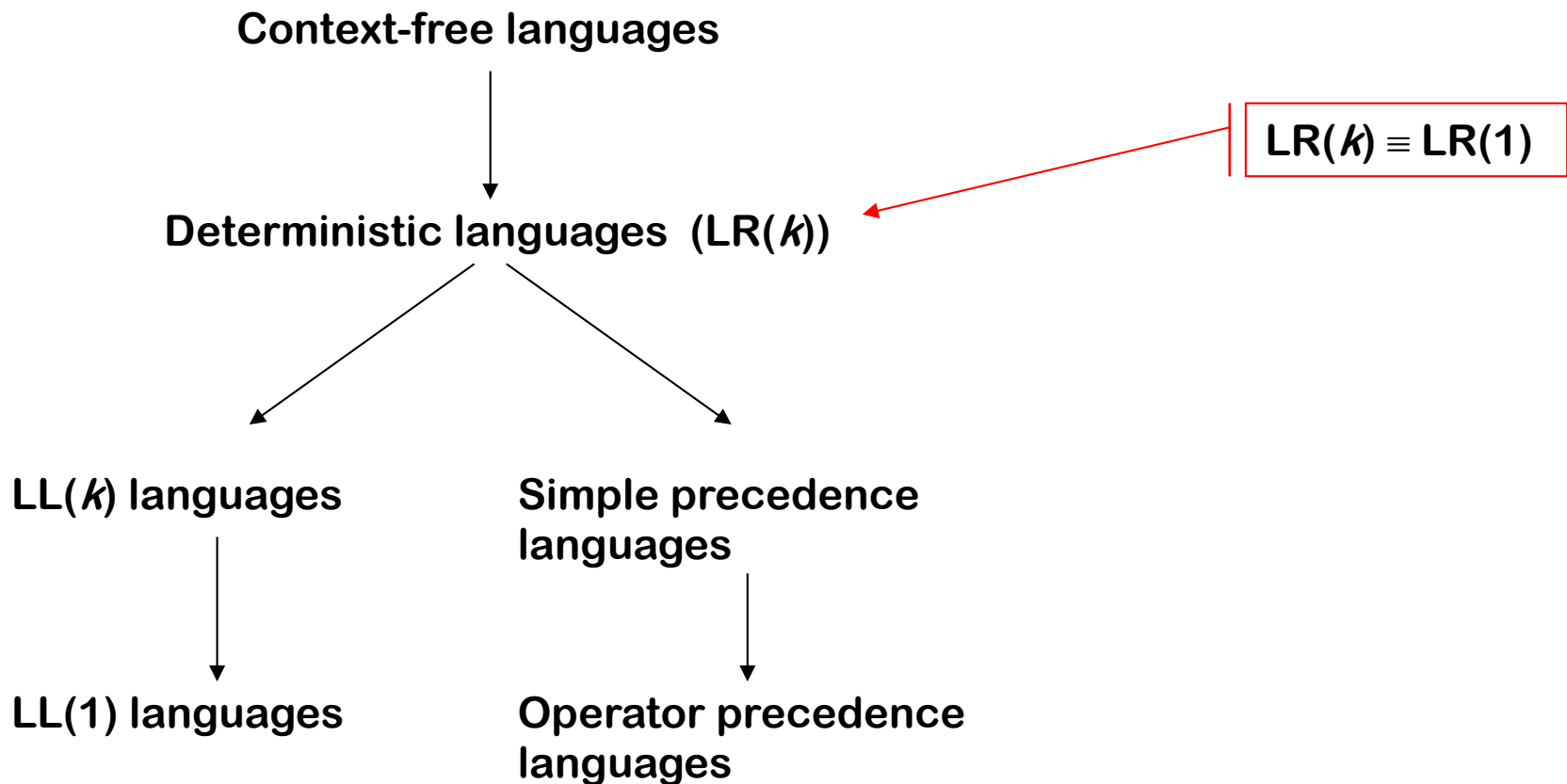
- the complete left context,
- the reducible phrase, itself, and
- the  $k$  terminal symbols to its right

$LL(k) \Rightarrow$  Parser must select the next rule based on

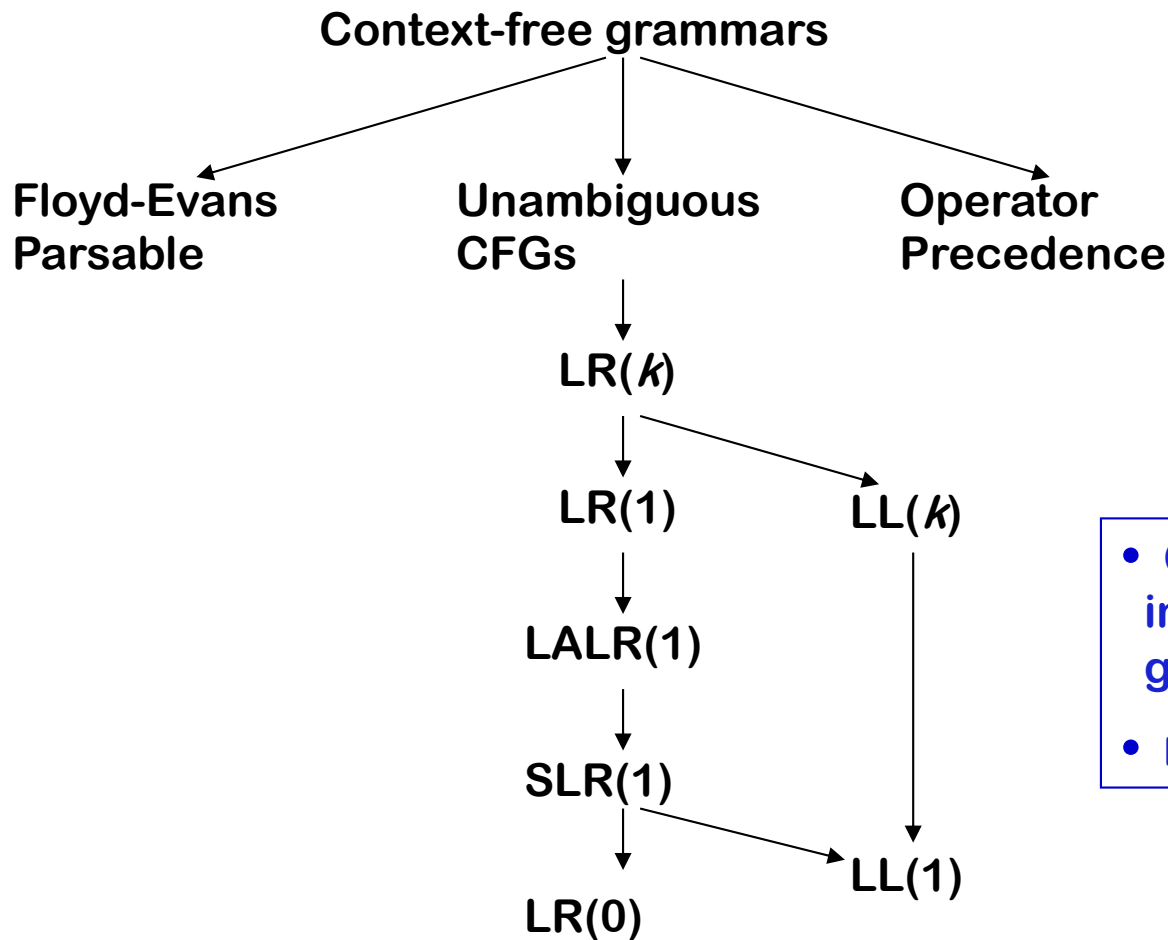
- The complete left context
- The next  $k$  terminals

Thus,  $LR(k)$  examines more context

|                            | <i>Advantages</i>  | <i>Disadvantages</i>                                  |
|----------------------------|--|---|
| Top-down recursive descent | Easy to implement<br>Good locality (fast)<br>Simplicity<br>Easy to embed actions (code access) | Hand-coded<br>High maintenance<br>Right associativity |
| LR(1)                      | Fast<br>Deterministic langs.<br>Automatable (tool support)<br>Left associativity               | Large working sets<br>Large table sizes               |



*The inclusion hierarchy for  
context-free languages*



- Operator precedence includes some ambiguous grammars
- $LL(1)$  is a subset of  $SLR(1)$

*The inclusion hierarchy for context-free grammars*

Work on the project!

See you at the midterm on May 10, at 1:00pm, in class

Will keep additional office hours before exam. Will announce via piazza.

**GOOD LUCK WITH STUDYING!**