COMP 181

Lecture 10

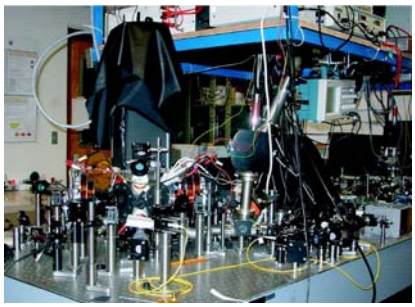
Parsing wrap-up, syntax-directed translation

October 5, 2006

Prelude

- What is a *qubit*?
  - Quantum bit
- Why quantum computing?
  - "*Superposition*" can search solutions to a problem simultaneously
  - 3 bits: 1 of 8 possible values
  - 3 qubits = all 8 values, with probabilities
- Is it fundamentally more powerful?
  - No. Just massively parallel.

Practical?



Today

- Issues with LR parsers
- Syntax-directed translation
- New homework assignment (on parsing)

Issues with LR parsers

- What happens if a state contains:

  $[X \rightarrow \alpha \bullet \underline{a}\beta, \underline{b}]$ and $[Y \rightarrow \gamma \bullet, \underline{a}]$
- Then on input "a" we could either
  - Shift into state $[X \rightarrow \alpha\underline{a} \bullet \beta, \underline{b}]$, or
  - Reduce with $Y \rightarrow \gamma$
- This is called a ***shift-reduce conflict***
  - Typically due to ambiguity

Shift/Reduce conflicts

- Classic example: the dangling else

  S → if E then S | if E then S else S | OTHER
- Will have DFA state containing

  [S → if E then S •, else]

  [S → if E then S • else S, x]
- Practical solutions:
  - Painful: modify grammar to reflect the precedence of else
  - Many LR parsers default to "shift"
  - Often have a precedence declaration

Another example

- Consider the ambiguous grammar
  $E \rightarrow E + E \mid E * E \mid int$
- Part of the DFA:

$$\begin{array}{l}[E \rightarrow E * \bullet E,\ +] \\ [E \rightarrow \bullet E + E,\ +] \\ \ldots\end{array} \xrightarrow{E} \begin{array}{l}[E \rightarrow E * E \bullet,\ +] \\ [E \rightarrow E \bullet + E,\ +] \\ \ldots\end{array}$$

- We have a shift/reduce on input +
- What do we want to happen?
  - Consider: **x * y + z**
  - We *need* to reduce (* binds more tightly than +)
  - Default action is shift

Precedence

- Declare relative precedence
  - Explicitly resolve conflict
  - Tell parser: we prefer the action involving * over +

$$\begin{array}{l}[E \rightarrow E * \bullet E,\ +] \\ [E \rightarrow \bullet E + E,\ +]\end{array} \xrightarrow{E} \begin{array}{l}[E \rightarrow E * E \bullet,\ +] \\ [E \rightarrow E \bullet + E,\ +]\end{array}$$

- In practice:
  - Parser generators support a precedence declaration for operators

More…

- Still a problem?

$$\begin{array}{l}[E \rightarrow E + \bullet E,\ +] \\ [E \rightarrow \bullet E + E,\ +]\end{array} \xrightarrow{E} \begin{array}{l}[E \rightarrow E + E \bullet,\ +] \\ [E \rightarrow E \bullet + E,\ +]\end{array}$$

- Shift/reduce conflict on +
  - Do we care?
  - Maybe: we want left associativity
    parse: "a+b+c" as "((a+b)+c)"
  - Which rule should we choose?
  - Also handled by a declaration "+ is left-associative"

Other problems

- If a DFA state contains both
  $[X \rightarrow \alpha \bullet, \underline{a}]$ and $[Y \rightarrow \beta \bullet, \underline{a}]$
  - What's the problem here?
  - Two reductions to choose from when next token is <u>a</u>
- This is called a ***reduce/reduce*** conflict
  - Usually a serious ambiguity in the grammar
  - Must be fixed in order to generate parser

Reduce/Reduce conflicts

- <u>Example</u>: a sequence of identifiers
  $S \rightarrow \varepsilon \mid id \mid id\ S$
- There are two parse trees for the string id
  $S \rightarrow id$
  $S \rightarrow id\ S \rightarrow id$
- How does this confuse the parser?

Reduce/Reduce conflicts

- Consider the DFA states:

$$\begin{array}{l}[G \rightarrow \bullet S,\ \$] \\ [S \rightarrow \bullet,\ \$] \\ [S \rightarrow \bullet\ \underline{id},\ \$] \\ [S \rightarrow \bullet\ \underline{id}\ S,\ \$]\end{array} \xrightarrow{\underline{id}} \begin{array}{l}[S \rightarrow \underline{id} \bullet,\ \$] \\ [S \rightarrow \underline{id} \bullet S,\ \$] \\ [S \rightarrow \bullet,\ \$] \\ [S \rightarrow \bullet\ \underline{id},\ \$] \\ [S \rightarrow \bullet\ \underline{id}\ S,\ \$]\end{array}$$

- Reduce/reduce conflict on input $
  $G \rightarrow S \rightarrow id$
  $G \rightarrow S \rightarrow id\ S \rightarrow id$
- Better rewrite the grammar: $S \rightarrow \varepsilon \mid id\ S$

Practical issues

We use an LR parser generator…

- Question: how many DFA states are there?
  - Does it matter?
  - What does that affect?
    - Parsing time is the same
    - Table size: occupies memory
- Even simple languages have 1000s of states

  Most LR parser generators don't construct the DFA as described

LR(1) Parsing tables

- But many states are similar, e.g.

1: E → int •, $/+ (E → int on $, +) and 5: E → int •,)/+ (E → int on), +)

- How can we exploit this?
  - Same reduction, different lookahead tokens
  - **Idea**: merge the states…

1′: E → int •, $/+/) (E → int on $, +,)

The core of a set of LR Items

- When can states be merged?
- *Def*: the ***core*** of a set of LR items is:
  - Just the production parts of the items
  - Without the lookahead terminals
- Example: the core of

  $\{ [X \to \alpha \bullet \beta, \underline{b}], [Y \to \gamma \bullet \delta, \underline{d}] \}$

  is

  $\{X \to \alpha \bullet \beta,\ \ Y \to \gamma \bullet \delta\}$

Merging states

- Consider for example the LR(1) states

  $\{[X \to \alpha \bullet, \underline{a}], [Y \to \beta \bullet, \underline{c}]\}$

  $\{[X \to \alpha \bullet, \underline{b}], [Y \to \beta \bullet, \underline{d}]\}$

- They have the same core and can be merged
- Resulting state is:

  $\{[X \to \alpha \bullet, \underline{a/b}], [Y \to \beta \bullet, \underline{c/d}]\}$

  Does this state do the same thing?

- These are called ***LALR(1) states***
  - Stands for LookAhead LR
  - Typically 10X fewer LALR(1) states than LR(1)

The LALR(1) DFA

- **Algorithm**:

repeat

  Choose two states with same core

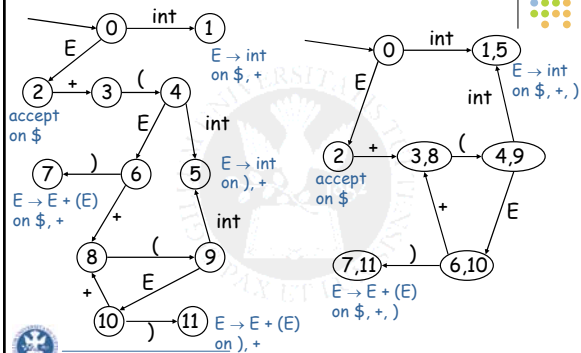
  Merge the states by combining the items

  Point edges from predecessors to new state

  New state points to all the previous successors

until all states have distinct core

Conversion LR(1) to LALR(1).



LALR states

- Consider the LR(1) states:

  $\{[X \rightarrow \alpha\bullet, \underline{a}], [Y \rightarrow \beta\bullet, \underline{b}]\}$

  $\{[X \rightarrow \alpha\bullet, \underline{b}], [Y \rightarrow \beta\bullet, \underline{a}]\}$

- And the merged LALR(1) state

  $\{[X \rightarrow \alpha\bullet, \underline{a/b}], [Y \rightarrow \beta\bullet, \underline{a/b}]\}$

- What's wrong with this?
  - Introduced a new reduce-reduce conflict
  - In practice such cases are rare

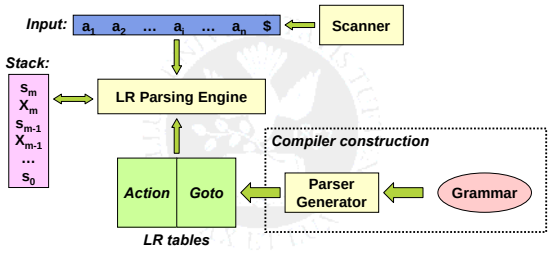
LALR vs. LR Parsing

- LALR is an efficiency hack on LR languages
- Any "reasonable" programming language has a LALR(1) grammar

  Languages that are not LALR(1) are weird, unnatural languages
- LALR(1) has become a standard for programming languages and for parser generators
- Variant: SLR

  LR(0), with special rule: reduce A --> b only if next token is in FOLLOW of A

LR parsing



Parser generators

- Example: JavaCUP
  - LALR(1) parser generator
  - Input: grammar specification
  - Output: Java classes
    - Generic engine
    - Action/goto tables
- Separate scanner specification
- Similar tools:
  - SableCC
  - yacc and bison generate C/C++ parsers
  - JavaCC: similar, but generates LL(1) parser

JavaCUP example

- Simple expression grammar
  - Operations over numbers only

```
// Import generic engine code
import java_cup.runtime.*;

/* Preliminaries to set up and use the scanner. */
init with {: scanner.init(); :};
scan with {: return scanner.next_token(); :};
```

  - Note: interface to scanner

    One issue: how to agree on names of the tokens

Example

- Define terminals and non-terminals
- Indicate operator precedence

```
/* Terminals (tokens returned by the scanner). */
terminal SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
terminal UMINUS, LPAREN, RPAREN;
terminal Integer NUMBER;

/* Non terminals */
non terminal          expr_list, expr_part;
non terminal Integer expr, term, factor;

/* Precedences */
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
```

Example

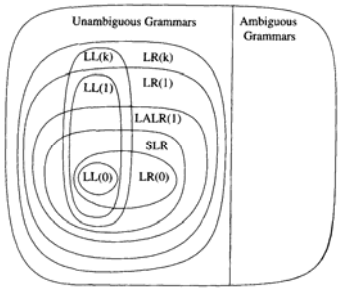
- Grammar rules

```
expr_list ::= expr_list expr_part
            |  expr_part ;

expr_part ::= expr SEMI ;

expr ::= expr PLUS expr
       | expr MINUS expr
       | expr TIMES expr
       | expr DIVIDE expr
       | expr MOD expr
       | LPAREN expr RPAREN
       | NUMBER ;
```


Notes on Parsing

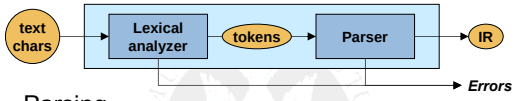
- Parsing
  - A solid foundation: context-free grammars
  - A simple parser: LL(1)
  - A more powerful parser: LR(1)
  - An efficiency hack: LALR(1)
  - LALR(1) parser generators

A Hierarchy of Grammar Classes



From Andrew Appel, "Modern Compiler Implementation in Java"

Overview



- Parsing
  - Tells us if input is syntactically correct
  - Gives us derivation or parse tree
  - But we want to do more:
    - Build some data structure – the IR
    - Perform other checks and computations

Syntax-directed translation

- In practice:
  - Fold some computations into parsing
  - Computations are triggered by parsing steps
  - ➡ ***Syntax-directed translation***
- Parser generators
  - Add action code to do something
  - Typically build the IR
- How much can we do during parsing?

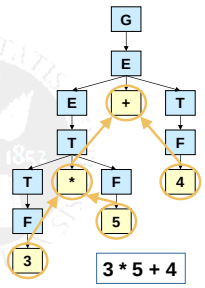
Syntax-directed translation

- General strategy
  - Associate values with grammar symbols
  - Associate computations with productions
- Implementation approaches
  - **Formal**: attribute grammars
  - **Informal**: ad-hoc translation schemes
- Some things cannot be folded into parsing

Example

- Desk calculator
  - Expression grammar
  - Build parse tree
  - Evaluate the resulting tree

#	Production rule
1	$G \rightarrow E$
2	$E \rightarrow E_1 + T$
3	$E \rightarrow T$
4	$T \rightarrow T_1 * F$
5	$T \rightarrow F$
6	$F \rightarrow (E)$
7	$F \rightarrow$ <u>num</u>

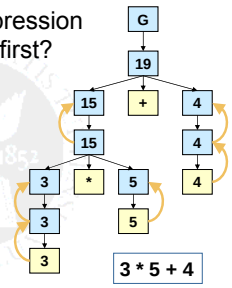


Example

- Can we evaluate the expression without building the tree first?

  "Piggyback" on parsing

#	Production rule
1	$G \rightarrow E$
2	$E \rightarrow E_1 + T$
3	$E \rightarrow T$
4	$T \rightarrow T_1 * F$
5	$T \rightarrow F$
6	$F \rightarrow (E)$
7	$F \rightarrow$ <u>num</u>



Example

- Codify:
  - Store intermediate values with non-terminals
  - Perform computations in each production

#	Production rule	Computation
1	$G \rightarrow E$	print(E.val)
2	$E \rightarrow E_1 + T$	E.val ← E_1.val + T.val
3	$E \rightarrow T$	E.val ← T.val
4	$T \rightarrow T_1 * F$	T.val ← T_1.val * F.val
5	$T \rightarrow F$	T.val ← F.val
6	$F \rightarrow (E)$	F.val ← E.val
7	$F \rightarrow$ <u>num</u>	F.val ← valueof(<u>num</u>)

Attribute grammars

- A context-free grammar with a set of rules
  - Each symbol has a set of values, or ***attributes***
  - ***Semantic rules*** : how to compute each attribute
- The bad news:

  Attribute grammars never widely adopted
- Why study them?
  - The attribute grammar formalism is important
    - Succinctly makes many points clear
    - Sets the stage for actual, *ad-hoc* practice
  - The problems motivate practice

Example

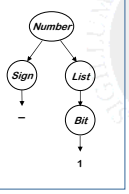
- Grammar:
  - Describes signed binary numbers
  - We would like to augment it with rules that compute the ***decimal value*** of each valid input string

#	Production rule
1	*Number* → *Sign List*
2	*Sign* → +
3	| -
4	*List* → *List Bit*
5	| *Bit*
6	*Bit* → <u>0</u>
7	| <u>1</u>

Example derivations

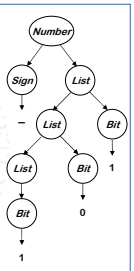
For "–1"

Number → *Sign List*
→ *– List*
→ *– Bit*
→ *– 1*



For "–101"

Number → *Sign List*
→ *Sign List Bit*
→ *Sign List 1*
→ *Sign List Bit 1*
→ *Sign List 1 1*
→ *Sign Bit 0 1*
→ *Sign 1 0 1*
→ *– 101*



Attribute grammar

- Goal:
  Compute the value of the binary number
- Information we need
  - Position of each 1 bit – to compute place value
    101 = 1*2² + 0*2¹ + 1*2⁰
  - Sum of bit values
- Computation
  - Propagate position information
  - Accumulate the sums

Attributes

Rules

Attribution rules

#	Production rule
1	***Number*** → ***Sign List***
2	***Sign*** → **+**
3	| **-**
4	***List$_0$*** → ***List$_1$ Bit***
5	| ***Bit***
6	***Bit*** → **0**
7	| **1**

How to compute Number from Sign and List?

if Sign.neg
  then Number.val ← -1 * List.val
  else Number.val ← List.val

How to compute List value?

List$_0$.val ← List$_1$.val + Bit.val ***or***
List$_0$.val ← Bit.val

How to compute Bit value?

Bit.val ← 0 ***or***
Bit.val ← $2^{(\text{bit position})}$

Where does pos come from?

Attribution rules

#	Production rule
1	***Number*** → ***Sign List***
2	***Sign*** → **+**
3	| **-**
4	***List$_0$*** → ***List$_1$ Bit***
5	| ***Bit***
6	***Bit*** → **0**
7	| **1**

Start at bit position 0

List.pos = 0

Push position information down

List$_1$.pos ← List$_0$.pos + 1
Bit.pos ← List$_0$.pos
or
Bit.pos ← List$_0$.pos

Now we can compute Bit value

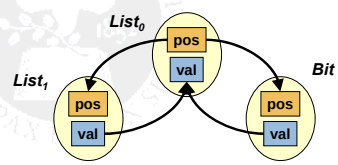
Bit.val ← 0 ***or***
Bit.val ← $2^{(\text{Bit.pos})}$

Attribution rules

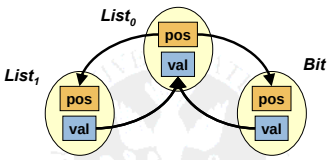
#	Production rule	Attribution rules
4	***List$_0$*** → ***List$_1$ Bit***	List$_0$.val ← List$_1$.val + Bit.val List$_1$.pos ← List$_0$.pos + 1 Bit.pos ← List$_0$.pos

Notice: Information can flow top-down or bottom-up

(Also: Left-to-right or Right-to-left)



Attribution rules



- Top-down values are ***inherited attributes***
- Bottom-up values are ***synthesized attributes***
- Values with no dependence are called ***independent attributes***

Attribute grammars

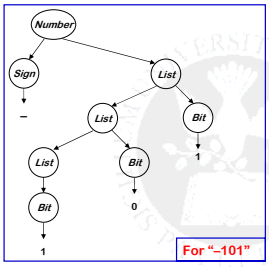
Specification:

- **Attributes**
  - Associated with nodes in parse tree
  - Distinguish multiple non-terminals with index
- **Rules**
  - Value assignments associated with productions
  - Information is entirely local: it can only refer to values in the given production
- Given a parse tree
  - Rules form a ***dependence graph*** between attributes
  - Result: a high-level, functional specification

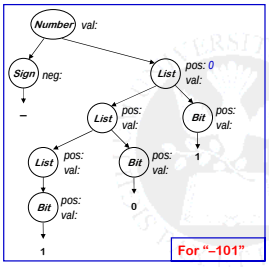
Evaluation

- Tricky part
  - Values flowing both up and down in tree
  - How do we order the computation?
  - And, how does that relate to parsing order?
    (i.e., the order in which parse tree nodes are created)
- Key
  - Must obey the dependence graph
  - What other constraints?

Dependence graph

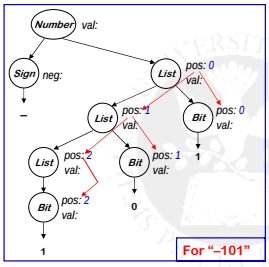


Dependence graph



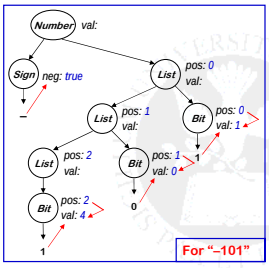
Annotate parse tree with attributes

Dependence graph



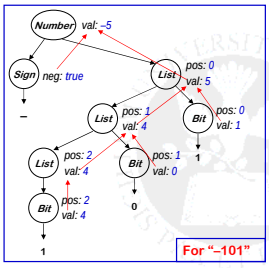
Inherited attributes flow down in the tree

Dependence graph



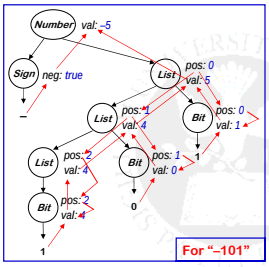
At leaves, add dependences between inherited and synthesized attributes

Dependence graph



Collect the synthesized attributes

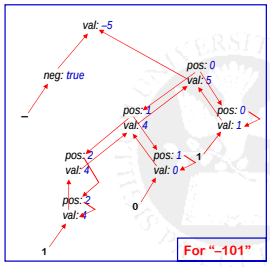
Dependence graph



Complete graph

Now, throw away the parse tree…

Dependence graph



- Need a method to solve the set of constraints described by this dependence graph

Evaluation

- Dynamic, dependence-based methods
  - Build the parse tree, dependence graph
  - Topologically sort the graph
  - Gives us an order of evaluation
- Rule-based methods
  - Analyze rules at compiler-generation time
  - Determine a fixed (static) ordering
  - Evaluate nodes in that order
- Oblivious methods
  - Ignore rules & parse tree
  - Pick a convenient order (at design time) & use it

Syntax-directed translation

- Attribute grammars
  - Clean, declarative
  - Handle a wide variety of problems
  - BUT, have limitations and evaluation issues
  - ⇨ Never widely adopted
- Reality
  - In practice:
    - *Apply arbitrary code actions on attributes*
  - Order of evaluation dictated by parsing algorithm
    - *Only works for limited classes of attribute grammars*

Adding actions

- ***L-attributed*** definition
  - Use values from parent and siblings
  - For production $\mathbf{A \rightarrow X_1\ X_2 \ldots X_n}$
  - Each attribute of X_i depends on
    - Attributes of $X_1\ X_2 \ldots X_{i-1}$, and
    - Inherited attributes of A
- Suited to LL parsing
  - Evaluate in a single top-down pass (left to right)
  - Pass values down through recursive descent
  - Table driven: store intermediate values on stack

Adding actions

- ***S-attributed*** definition
  - All attributes are synthesized
  - For production $\mathbf{A \rightarrow X_1\ X_2 \ldots X_n}$
  - Value of **A** is computed as a function of the attributes already computed for $\mathbf{X_1\ X_2 \ldots X_n}$
- Suited to LR parsing
  - Can be computed in a single bottom-up pass
  - Associate pieces of code with each production
  - At each reduction, the code is executed

Example

Name for the value associaed with this production

Arbitrary code between {: and :}

```
expr_part ::= expr SEMI ;

expr ::= expr:e1 PLUS expr:e2
         {: RESULT = new Integer(e1.intValue() +
                                 e2.intValue()); :}
       | expr:e1 MINUS expr:e2
         {: RESULT = new Integer(e1.intValue() -
                                 e2.intValue()); :}
         ...

       | LPAREN expr:e RPAREN
         {: RESULT = e; :}

       | NUMBER:n
         {: RESULT = n; :}
       ;
```

RESULT refers to the attribute of the LHS non-terminal

Another example

- Build an abstract syntax tree

```
expr_part ::= expr SEMI ;

expr ::= expr:e1 PLUS expr:e2
         {: RESULT = new AddNode(e1, e2); :}
       | expr:e1 MINUS expr:e2
         {: RESULT = new SubNode(e1, e2); :}
         ...

       | LPAREN expr:e RPAREN
         {: RESULT = e; :}

       | NUMBER:n
         {: RESULT = new NumberNode(n); :}
       ;
```


Implementation

- How does this work?
  - Where are the attributes stored?
  - What do e1, e2, n, RESULT refer to?
- **Key**: store attributes on stack

  At a reduction of $A \rightarrow \beta$
  - Pop 3 × |β| symbols – 1 symbol, 1 state, **1 value**
  - Map values to names:
    - expr:**e1** PLUS expr:**e2**
    - **e2** = top of stack, then PLUS, then **e1**
  - Invoke action code on values – store in RESULT
  - Push RESULT back on stack with new symbol, state

Next time…

- We've built our abstract syntax tree
- Now what?
  - Type checking
  - Symbol tables
  - Semantic checking