Last week:
LR Parsing with Pushdown Automaton

- s = top of stack, t = next token, use ACTION[s][t] to determine what is the next move
- Shift move
  - Remove first token from input
  - Push t on the stack
  - Compute next state s' = GOTO[s][t] table
  - Push new state s' on the stack
  - If new state is error - report error
- Reduce move
  - Using a rule N → α
  - Symbols in α and their following states are removed from stack. Let q denote the state on top of stack after their removal
  - Push N on the stack
  - Compute next state s' = GOTO[q][N] table
  - Push new state s' on the stack (on top of N)
Last week: shift move

1. Remove first token from input
2. Push t on the stack
3. Compute s' = GOTO[s][t] table
4. Push s' state on the stack
5. If new state is error

Last week: reduce move

1. Using a rule N → ACTION[s][t]
2. Symbols in α and their following states are removed from stack. q = top afterwards.
3. Push N on the stack
4. New state s' = GOTO[q][N] table
5. Push new state s' on top of N

Constructing Parse Table: LR(0) Automaton Example

State i + ( ) E T
q0 s5 s7 s2
q1 s4 s1 s1 s1 s1 s1 s1
q2 s3 s4 s4 s4 s4 s4 s4
q3 s4 s4 s4 s4 s4 s4 s4
q4 s4 s4 s4 s4 s4 s4 s4
q5 s4 s4 s4 s4 s4 s4 s4
q6 s4 s4 s4 s4 s4 s4 s4
q7 s4 s4 s4 s4 s4 s4 s4
q8 s4 s4 s4 s4 s4 s4 s4
q9 s4 s4 s4 s4 s4 s4 s4

State i + ( ) s E T
q0 s5 s7 s2
q1 s4 s1 s1 s1 s1 s1 s1
q2 s3 s4 s4 s4 s4 s4 s4
q3 s4 s4 s4 s4 s4 s4 s4
q4 s4 s4 s4 s4 s4 s4 s4
q5 s4 s4 s4 s4 s4 s4 s4
q6 s4 s4 s4 s4 s4 s4 s4
q7 s4 s4 s4 s4 s4 s4 s4
q8 s4 s4 s4 s4 s4 s4 s4
q9 s4 s4 s4 s4 s4 s4 s4

Last week: GOTO/ACTION Table

<table>
<thead>
<tr>
<th>State</th>
<th>i</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>s</th>
<th>E</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>q0</td>
<td>s5</td>
<td>s7</td>
<td>s2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q1</td>
<td>s4</td>
<td>s1</td>
<td>s1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q2</td>
<td>s3</td>
<td>s4</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q3</td>
<td>s4</td>
<td>s4</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q4</td>
<td>s4</td>
<td>s4</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q5</td>
<td>s4</td>
<td>s4</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q6</td>
<td>s4</td>
<td>s4</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q7</td>
<td>s4</td>
<td>s4</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q8</td>
<td>s4</td>
<td>s4</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q9</td>
<td>s4</td>
<td>s4</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(1) i → E T
(2) i → T E
(3) i → T
(4) T → 1
(5) T → (E)

Warning: numbers mean different things!
rm = reduce using rule number
sm = shift to state m
LR Parsing with Pushdown Automaton
(superimposed GOTO/ACTION)
- \( s = \) top of stack, \( t = \) next token,
  \( \text{move}=\text{GOTO/ACTION}(s,t) \) to determine what is the next move
- If (move = Sm)
  - Remove first token \( t \) from input
  - Push \( t \) on the stack
  - Push new state \( m \) on the stack
- If (move = rn)
  - use rule number \( n \):
    - Symbols in \( \alpha \) and their following states are removed from stack. Let \( q \) denote the state on top of stack after their removal
    - Push \( N \) on the stack
    - Compute next state \( s' = \text{GOTO/ACTION}(q,N) \) table
    - Push new state \( s' \) on the stack (on top of \( N \))
- If (move = empty) report ERROR

Are we done?
- Can make a transition diagram for any grammar
- Can make a GOTO table for every grammar
- Cannot make a deterministic ACTION table for every grammar

LR(0) Conflicts
LR(0) Conflicts

- Any grammar with an $\varepsilon$-rule cannot be LR(0)
- Inherent shift/reduce conflict
  - $A \xrightarrow{\varepsilon} \cdot$ reduce item
  - $P \xrightarrow{\alpha \gamma} \cdot$ shift item
  - $A \xrightarrow{\varepsilon}$ can always be predicted from $P \xrightarrow{\alpha \gamma}$

Back to the GOTO/ACTIONS tables

<table>
<thead>
<tr>
<th>State</th>
<th>GOTO Table</th>
<th>ACTION Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$q_5$</td>
<td>$q_7$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$q_3$</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$Z \rightarrow \varepsilon$</td>
<td></td>
</tr>
<tr>
<td>$q_3$</td>
<td>$q_5$</td>
<td>$q_7$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>$E \rightarrow E + T$</td>
<td></td>
</tr>
<tr>
<td>$q_5$</td>
<td>$E \rightarrow T$</td>
<td></td>
</tr>
<tr>
<td>$q_6$</td>
<td>$T \rightarrow i$</td>
<td></td>
</tr>
<tr>
<td>$q_7$</td>
<td>$q_5$</td>
<td>$q_7$</td>
</tr>
<tr>
<td>$q_8$</td>
<td>$T \rightarrow i$</td>
<td></td>
</tr>
<tr>
<td>$q_9$</td>
<td>$q_3$</td>
<td>$q_9$</td>
</tr>
</tbody>
</table>

ACTION table determined only by transition diagram, ignores input

SLR Grammars

- Don't reduce if it will get you into trouble on the next token
- A handle should not be reduced to a non-terminal $N$ if the look-ahead is a token that cannot follow $N$
- A reduce item $N \rightarrow \alpha \cdot$ is applicable only when the look-ahead is in FOLLOW($N$)
- Differs from LR(0) only on the ACTION table
LR(0) Conflicts

Shift/reduce conflict

SLR ACTION Table

<table>
<thead>
<tr>
<th>State</th>
<th>i</th>
<th>*</th>
<th>(</th>
<th>l</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>q0</td>
<td>shift</td>
<td>shift</td>
<td>shift</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q1</td>
<td>shift</td>
<td>shift</td>
<td>shift</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q2</td>
<td>shift</td>
<td>shift</td>
<td>shift</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q3</td>
<td>E → E + T</td>
<td>E → E + T</td>
<td>E → E + T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q4</td>
<td>T → i</td>
<td>S</td>
<td>S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q5</td>
<td>shift</td>
<td>shift</td>
<td>shift</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q6</td>
<td>E → T</td>
<td>E → T</td>
<td>E → T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q7</td>
<td>shift</td>
<td>shift</td>
<td>shift</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q8</td>
<td>shift</td>
<td>shift</td>
<td>shift</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q9</td>
<td>T → (E)</td>
<td>T → (E)</td>
<td>T → (E)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SLR – use 1 token look-ahead

Are we done?

(a) S' → S
(b) S → L → R
(c) S → R
(d) L → a
(e) S → L
LR(1) Grammars

- In SLR: a reduce item \( \alpha \cdot \) is applicable only when the look-ahead is in FOLLOW(\( \alpha \cdot \))
- But FOLLOW(\( \alpha \cdot \)) merges look-ahead for all alternatives for \( \alpha \cdot \)
- LR(1) keeps look-ahead with each LR item
- Idea: a more refined notion of follows computed per item

Shift/reduce conflict

- \( S \rightarrow L \cdot \) vs. \( R \rightarrow L \cdot \)
- FOLLOW(R) contains =
  - \( S \Rightarrow L = R \Rightarrow * R \Rightarrow R \)
- SLR cannot resolve the conflict either

LR(1) Item

- LR(\( \alpha \cdot \)) item is a pair
  - LR(0) item
  - Look-ahead token
- Meaning
  - We matched the part left of the dot, looking to match the part on the right of the dot, followed by the look-ahead token.
- Example
  - The production \( L \rightarrow id \) yields the following LR(\( \alpha \cdot \)) items

\[
\begin{align*}
\langle L \rightarrow \bullet id, * \rangle \\
\langle L \rightarrow id, \bullet \rangle \\
\langle L \rightarrow id, * \rangle \\
\langle L \rightarrow id, id \rangle
\end{align*}
\]
\( \varepsilon \)-closure for LR(1)

- For every \([A \rightarrow \alpha \cdot \beta, c]\) in \(S\)
  - for every production \(B \rightarrow \delta\) and every token \(b\) in the grammar such that \(b \in \text{FIRST}(\beta c)\)
  - Add \([B \rightarrow \delta, b]\) to \(S\)

---

Back to the conflict

- Is there a conflict now?

---

LALR

- LR tables have large number of entries
- Often don’t need such refined observation (and cost)
- LALR idea: find states with the same LR(0) component and merge their look-ahead component as long as there are no conflicts
- LALR not as powerful as LR(1)
Summary: LR Grammars

- LR parsing techniques use item sets of proposed handles
  - Shift behavior similar
  - Differ on when to reduce

- LR(0) - any reduce item causes a reduction
- SLR – a reduce item $N \to \alpha \bullet$ causes a reduction only if the look-ahead token is in the FOLLOW set of $N$
- LR(1) - a reduce item $N \to \alpha \bullet \sigma$ causes a reduction only if the look-ahead token is in the set $\sigma$ (the look-ahead set computed for the item)

Summary

- Bottom up
  - LR Items
  - LR parsing with pushdown automata
  - LR(0), SLR, LR(1) – different kinds of LR items, same basic algorithm

Summary: LR Grammars

- ACTION table determines whether to shift or reduce
- On a shift, new state found using the GOTO table
- LR-parser with 1 token look-ahead, the ACTION and GOTO tables can be superimposed

You are here...
What we want

Potato potato;
Carrot carrot;
x = tomato + potato + carrot

Parser

tomato is undefined
potato used before initialized
Cannot add Potato and Carrot

Syntax vs. Semantics

- Syntax
  - Program structure
  - Formally described via context free grammars
- Semantics
  - Program meaning
  - Formally defined as various forms of semantics (e.g., operational, denotational)
  - It is actually NOT what “semantic analysis” phase does
  - Better name – “contextual analysis”

Contextual Analysis

- Often called “Semantic analysis”
- Properties that cannot be formulated via CFG
  - Type checking
  - Declare before use
    - Identifying the same word “w” re-appearing – wbw
  - Initialization
    - ...
- Properties that are hard to formulate via CFG
  - “break” only appears inside a loop
    - ...
- Processing of the AST

Abstract Syntax Tree (AST)

- Abstract away some syntactic details of the source language

$$S \rightarrow \text{if } E \text{ then } S \text{ else } S$$

```plaintext
if (x > 0)
  then \( y = 42 \)
  else \( y = 73 \)
```
Syntax Directed Translation

- Semantic attributes
  - Attributes attached to grammar symbols
- Semantic actions
  - (already mentioned when we did recursive descent)
  - How to update the attributes
- Attribute grammars

Attribute grammars

- Attributes
  - Every grammar symbol has attached attributes
    - Example: Expr.type
- Semantic actions
  - Every production rule can define how to assign values to attributes
    - Example:
      Expr → Expr + Term
      Expr.type = Expr1.type when (Expr1.type = Term.type)
      Error otherwise
Indexed symbols

- Add indexes to distinguish repeated grammar symbols
- Does not affect grammar
- Used in semantic actions
- Expr → Expr + Term
  Becomes
  Expr → Expr₁ + Term

Example

Dependency

- A semantic equation $a = b₁, ..., bm$ requires computation of $b₁, ..., bm$ to determine the value of $a$
- The value of $a$ depends on $b₁, ..., bm$
  - We write $a ← bᵢ$

Attribute Evaluation

- Build the AST
- Fill attributes of terminals with values derived from their representation
- Execute evaluation rules of the nodes to assign values until no new values can be assigned
  - In the right order such that
    - No attribute value is used before its available
    - Each attribute will get a value only once

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T L</td>
<td>L.in = T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type = float</td>
</tr>
<tr>
<td>L → L₁, id</td>
<td>L.in = L.in addType(id.entry,L.in)</td>
</tr>
<tr>
<td>L → id</td>
<td>addType(id.entry,L.in)</td>
</tr>
</tbody>
</table>
Cycles

- Cycle in the dependence graph
- May not be able to compute attribute values

Attribute Evaluation

- Build the AST
- Build dependency graph
- Compute evaluation order using topological ordering
- Execute evaluation rules based on topological ordering
- Works as long as there are no cycles

Building Dependency Graph

- All semantic equations take the form
  attr1 = func1(attr1.1, attr1.2, ...)
  attr2 = func2(attr2.1, attr2.2, ...)

- Actions with side effects use a dummy attribute
- Build a directed dependency graph G
  - For every attribute a of a node n in the AST create a node n.a
  - For every node n in the AST and a semantic action of the form b = f(c1,c2,...,ck) add edges of the form (ci,b)

Example

```
Prod.       Semantic Rule
D → TL     L.in = T.type
T → int    T.type = integer
T → float  T.type = float
L → L4      L.in = L.in
            addType(id.entry,L.in)
L → id     L.in = L.in
            addType(id.entry,L.in)
```
**Example**

```
float x,y,z
```

**Topological Order**

- For a graph \( G=(V,E) \), \(|V|=k\)
- Ordering of the nodes \( v_1,v_2,\ldots,v_k \) such that for every edge \((v_i,v_j) \in E, i < j\)

```
Example topological orderings: 1 4 3 2 5, 4 1 3 5 2
```

**But what about cycles?**

- For a given attribute grammar hard to detect if it has cyclic dependencies
  - Exponential cost
- Special classes of attribute grammars
  - Our “usual trick”
  - Sacrifice generality for predictable performance
Inherited vs. Synthesized Attributes

- Synthesized attributes
  - Computed from children of a node
- Inherited attributes
  - Computed from parents and siblings of a node
- Attributes of tokens are technically considered as synthesized attributes

example

```
Production  Semantic Rule
D → T L  L.in = T.type
T → int    T.type = integer
T → float  T.type = float
L → L1, id L.in = L.in
        addType(id.entry, L.in)
L → id    addType(id.entry, L.in)
```

S-attributed Grammars

- Special class of attribute grammars
- Only uses synthesized attributes (S-attributed)
- No use of inherited attributes
- Can be computed by any bottom-up parser during parsing
- Attributes can be stored on the parsing stack
- Reduce operation computes the (synthesized) attribute from attributes of children

S-attributed Grammar: example

```
Production  Semantic Rule
S → E;       print(E.val)
E → E1 + T   E.val = E1.val + T.val
E → T        E.val = T.val
T → T1 * F   T.val = T1.val * F.val
T → F        T.val = F.val
F → (E)      F.val = E.val
F → digit    F.val = digit.lexval
```
L-attributed grammars

- L-attributed attribute grammar when every attribute in a production $A \rightarrow X_1...X_n$ is
  - A synthesized attribute, or
  - An inherited attribute of $X_j$, $1 \leq j \leq n$ that only depends on
    - Attributes of $X_1...X_{j-1}$ to the left of $X_j$, or
    - Inherited attributes of $A$

Summary

- Contextual analysis can move information between nodes in the AST
  - Even when they are not “local”
- Attribute grammars
  - Attach attributes and semantic actions to grammar
- Attribute evaluation
  - Build dependency graph, topological sort, evaluate
- Special classes with pre-determined evaluation order: $S$-attributed, L-attributed

The End
Semantic Checks

- Scope rules
  - Use symbol table to check that
    - Identifiers defined before used
    - No multiple definition of same identifier
    - Program conforms to scope rules
- Type checking
  - Check that types in the program are consistent
  - How?

Type Checking

- Type rules specify
  - which types can be combined with certain operator
  - Assignment of expression to variable
  - Formal and actual parameters of a method call
- Examples

```java
string drive = "drink"
string
int 42 = "the answer"
ERROR
```
Type Checking Rules

- Specify for each operator
  - Types of operands
  - Type of result

- Basic Types
  - Building blocks for the type system (type rules)
  - e.g., int, boolean, string

- Type Expressions
  - Array types
  - Function types
  - Record types / Classes

Typing Rules

If $E_1$ has type int and $E_2$ has type int, then $E_1 + E_2$ has type int

$E_1 : \text{int}$ $E_2 : \text{int}$

$E_1 + E_2 : \text{int}$

(Generally, also use a context $A$)

More Typing Rules

$A : \text{true} : \text{boolean}$ $A : \text{false} : \text{boolean}$

$A : \text{int literal} : \text{int}$ $A : \text{string literal} : \text{string}$

$A : E_1 : \text{int}$ $A : E_2 : \text{int}$ $\text{op} \in \{+,\cdot,\ast,\%\}$

$A : E_1 \text{ op } E_2 : \text{int}$

$A : E_1 : \text{int}$ $A : E_2 : \text{int}$ $\text{op} \in \{\leq,<,>,>=\}$

$A : E_1 \text{ op } E_2 : \text{boolean}$

$A : E_1 : T$ $A : E_2 : T$ $\text{op} \in \{==,!=\}$

$A : E_1 \text{ op } E_2 : \text{boolean}$

$A : E_1 : \text{int}$


And Even More Typing Rules

$A : E_1 : \text{boolean}$ $A : E_2 : \text{boolean}$ $\text{op} \in \{\&\&,||\}$

$A : E_1 \text{ op } E_2 : \text{boolean}$

$A : E_1 : \text{int}$ $A : E_2 : \text{int}$

$A : E_1 : \text{int}$ $A : E_2 : \text{int}$

$A : E_1 : \text{int}$

$A : E_1 : T$ $A : E_2 : T$ $\text{op} \in \{==,!=\}$

$A : E_1 \text{ op } E_2 : \text{boolean}$

$A : \text{new } T() : T$ $A : \text{id} : T$
Type Checking

- Our approach --- Traverse AST bottom-up and assign types for AST nodes
  - Use typing rules to compute node types
- More complicated alternative --- type-check during parsing
  - But naturally also more efficient

Example

```
45 > 32 && !false
```

```
- op=AND
- op=NEG
- op=GT
- intLiteral
  - val=45
  - : int
- intLiteral
  - val=32
  - : int
- boolLiteral
  - val=false
  - : boolean
```