MIT 6.035
Parse Table Construction

Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
Parse Tables (Review)

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>$</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>shift to s2</td>
<td>$</td>
<td>X</td>
</tr>
<tr>
<td>s1</td>
<td>error</td>
<td>$</td>
<td>goto s1</td>
</tr>
<tr>
<td>s2</td>
<td>shift to s2</td>
<td>$</td>
<td>goto s3</td>
</tr>
<tr>
<td>s3</td>
<td>error</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>s4</td>
<td>reduce (2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s5</td>
<td>reduce (3)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Implements finite state control
- At each step, look up
  - Table[top of state stack] [ input symbol]
- Then carry out the action
### Parse Tables (Review)

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>$</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>shift to s2</td>
<td>error</td>
<td>error</td>
</tr>
<tr>
<td>s1</td>
<td>error</td>
<td>error</td>
<td>accept</td>
</tr>
<tr>
<td>s2</td>
<td>shift to s2</td>
<td>shift to s5</td>
<td>error</td>
</tr>
<tr>
<td>s3</td>
<td>error</td>
<td>shift to s4</td>
<td>error</td>
</tr>
<tr>
<td>s4</td>
<td>reduce (2)</td>
<td>reduce (2)</td>
<td>reduce (2)</td>
</tr>
<tr>
<td>s5</td>
<td>reduce (3)</td>
<td>reduce (3)</td>
<td>reduce (3)</td>
</tr>
</tbody>
</table>

- **Shift to s\(n\)**
  - Push input token into the symbol stack
  - Push s\(n\) into state stack
  - Advance to next input symbol
Parse Tables (Review)

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>shift to s2</td>
<td>error goto s1</td>
</tr>
<tr>
<td>s1</td>
<td>error</td>
<td>error accept</td>
</tr>
<tr>
<td>s2</td>
<td>shift to s2</td>
<td>shift to s5 error goto s3</td>
</tr>
<tr>
<td>s3</td>
<td>error</td>
<td>shift to s4 error</td>
</tr>
<tr>
<td>s4</td>
<td>reduce (2)</td>
<td>reduce (2) reduce (2)</td>
</tr>
<tr>
<td>s5</td>
<td>reduce (3)</td>
<td>reduce (3) reduce (3)</td>
</tr>
</tbody>
</table>

- **Reduce ($n$)**
  - Pop both stacks as many times as the number of symbols on the RHS of rule $n$
  - Push LHS of rule $n$ into symbol stack
Parser Generators and Parse Tables

- Parser generator (YACC, CUP)
  - Given a grammar
  - Produces a (shift-reduce) parser for that grammar
- Process grammar to synthesize a DFA
  - Contains states that the parser can be in
  - State transitions for terminals and non-terminals
- Use DFA to create an parse table
- Use parse table to generate code for parser
Example

• The grammar

\[ S \rightarrow X \; \$ \quad (1) \]
\[ X \rightarrow (X) \quad (2) \]
\[ X \rightarrow ( ) \quad (3) \]
DFA States Based on Items

- We need to capture how much of a given production we have scanned so far

\[ X \rightarrow (\ X \ ) \]

Are we here? Or here? Or here? Or here?
Items

- We need to capture how much of a given production we have scanned so far

\[ X \rightarrow (X) \]

- Production Generates 4 items
  - \( X \rightarrow (X) \)
  - \( X \rightarrow (\cdot X) \)
  - \( X \rightarrow (X\cdot) \)
  - \( X \rightarrow (X)\cdot \)
Example of Items

- The grammar
  
  \[
  S \rightarrow X \ \$ \\
  X \rightarrow (X) \\
  X \rightarrow (\ ) \\
  \]

- Items
  
  \[
  S \rightarrow \cdot \ X\$ \\
  S \rightarrow X\cdot \ \$ \\
  X \rightarrow \cdot \ (X) \\
  X \rightarrow (\cdot \ X) \\
  X \rightarrow (X\cdot) \\
  X \rightarrow (X) \cdot \\
  X \rightarrow \cdot \ (\ ) \\
  X \rightarrow (\cdot ) \\
  X \rightarrow (\ ) \cdot 
  \]
Notation

• If write production as $A \rightarrow \alpha c \beta$
  • $\alpha$ is sequence of grammar symbols, can be terminals and nonterminals in sequence
  • $c$ is terminal
  • $\beta$ is sequence of grammar symbols, can be terminals and nonterminals in sequence
• If write production as $A \rightarrow \alpha \cdot B \beta$
  • $\alpha, \beta$ as above
  • $B$ is a single grammar symbol, either terminal or nonterminal
Key idea behind items

- States correspond to sets of items
- If the state contains the item \( A \rightarrow \alpha \cdot c \beta \)
  
  Parser is expecting to eventually reduce using the production \( A \rightarrow \alpha \cdot c \beta \)
  
  - Parser has already parsed an \( \alpha \)
  - It expects the input may contain \( c \), then \( \beta \)
- If the state contains the item \( A \rightarrow \alpha \cdot \)
  
  - Parser has already parsed an \( \alpha \)
  - Will reduce using \( A \rightarrow \alpha \)
- If the state contains the item \( S \rightarrow \alpha \cdot $ \)
  
  and the input buffer is empty
  - Parser accepts input
Correlating Items and Actions

- If the current state contains the item $A \rightarrow \alpha \cdot c \beta$ and the current symbol in the input buffer is $c$:
  - Parser shifts $c$ onto stack
  - Next state will contain $A \rightarrow \alpha c \cdot \beta$

- If the current state contains the item $A \rightarrow \alpha$:
  - Parser reduces using $A \rightarrow \alpha$

- If the current state contains the item $S \rightarrow \alpha \cdot \$ and the input buffer is empty:
  - Parser accepts input
Closure() of a set of items

- Closure finds all the items in the same “state”
- Fixed Point Algorithm for Closure(I)
  - Every item in I is also an item in Closure(I)
    - If $A \rightarrow \alpha B \beta$ is in Closure(I) and $B \rightarrow \gamma$ is an item, then add $B \rightarrow \bullet \gamma$ to Closure(I)
  - Repeat until no more new items can be added to Closure(I)
Example of Closure

- Closure( \{X \rightarrow ( \cdot X)\})

\[
\begin{align*}
X & \rightarrow ( \cdot X) \\
X & \rightarrow \cdot (X) \\
X & \rightarrow \cdot ( )
\end{align*}
\]

- Items

\[
\begin{align*}
S & \rightarrow \cdot X$ \\
S & \rightarrow X\cdot $ \\
X & \rightarrow \cdot (X) \\
X & \rightarrow (\cdot X) \\
X & \rightarrow (X\cdot) \\
X & \rightarrow (X)\cdot \\
X & \rightarrow \cdot ( ) \\
X & \rightarrow ( . ) \\
X & \rightarrow ( )\cdot
\end{align*}
\]
Another Example

- closure(\{S \rightarrow \cdot X\}$\})

\[
\begin{align*}
S & \rightarrow \cdot X\$ \\
X & \rightarrow \cdot (X) \\
X & \rightarrow \cdot ( )
\end{align*}
\]

- Items

\[
\begin{align*}
S & \rightarrow \cdot X\$ \\
S & \rightarrow X\cdot \$ \\
X & \rightarrow \cdot (X) \\
X & \rightarrow (\cdot X) \\
X & \rightarrow (X\cdot) \\
X & \rightarrow (X) \cdot \\
X & \rightarrow \cdot ( ) \\
X & \rightarrow ( . ) \\
X & \rightarrow ( ) \cdot 
\end{align*}
\]
Goto() of a set of items

- Goto finds the new state after consuming a grammar symbol while at the current state

- Algorithm for Goto(I, X)
  where I is a set of items
  and X is a grammar symbol

\[
\text{Goto}(I, X) = \text{Closure}( \{ A \rightarrow \alpha X \beta \mid A \rightarrow \alpha \beta \text{ in } I \} )
\]

- goto is the new set obtained by “moving the dot” over X
Example of Goto

- Goto ( \{ \{X \rightarrow ( \cdot X)\}, X\} )

\hspace{1cm}
\begin{align*}
X & \rightarrow (X \cdot) \\
\end{align*}

- Items

\begin{align*}
S & \rightarrow \cdot X \$
S & \rightarrow X \cdot \$
X & \rightarrow \cdot (X)
X & \rightarrow (\cdot X)
X & \rightarrow (X \cdot)
X & \rightarrow (X) \cdot
X & \rightarrow \cdot ( )
X & \rightarrow ( )
X & \rightarrow ( ) \cdot
\end{align*}
## Another Example of Goto

- **Goto** (`{X → •(X)}`, ( ))

- **Items**
  
  \[
  \begin{align*}
  S & \rightarrow \cdot X$ \\
  S & \rightarrow X\cdot $ \\
  X & \rightarrow \cdot (X) \\
  X & \rightarrow (\cdot X) \\
  X & \rightarrow (X\cdot) \\
  X & \rightarrow (X)\cdot \\
  X & \rightarrow \cdot ( ) \\
  X & \rightarrow ( \cdot ) \\
  X & \rightarrow ( )\cdot \\
  \end{align*}
  \]
Building the DFA states

- Start with the item $S \rightarrow \cdot \beta \$ 
- Create the first state to be $\text{Closure}(\{ S \rightarrow \cdot \beta \$\})$
- Pick a state $I$
  - for each item $A \rightarrow \alpha \cdot X \beta$ in $I$
    - find $\text{Goto}(I, X)$
    - if $\text{Goto}(I, X)$ is not already a state, make one
    - Add an edge $X$ from state $I$ to $\text{Goto}(I, X)$ state
- Repeat until no more additions possible
DFA Example

\[ S \rightarrow X \cdot $ \]
\[ X \rightarrow \cdot (X) \]
\[ X \rightarrow \cdot ( ) \]

\[ S \rightarrow X \] $ \]
\[ X \rightarrow (X) \]
\[ X \rightarrow ( ) \]
Constructing A Parse Engine

• Build a DFA - DONE

• Construct a parse table using the DFA
Creating the parse tables

• For each state

  • Transition to another state using a terminal symbol is a shift to that state (shift to sn)
  • Transition to another state using a non-terminal is a goto to that state (goto sn)
  • If there is an item A → α in the state do a reduction with that production for all terminals (reduce k)
Building Parse Table Example

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>shift to s2</td>
<td>error goto s1</td>
</tr>
<tr>
<td>s1</td>
<td>error</td>
<td>accept</td>
</tr>
<tr>
<td>s2</td>
<td>shift to s2</td>
<td>shift to s5</td>
</tr>
<tr>
<td>s3</td>
<td>error</td>
<td>shift to s4</td>
</tr>
<tr>
<td>s4</td>
<td>reduce (2)</td>
<td>reduce (2)</td>
</tr>
<tr>
<td>s5</td>
<td>reduce (3)</td>
<td>reduce (3)</td>
</tr>
</tbody>
</table>

Production Rules:

- $S \rightarrow X \cdot$
- $X \rightarrow \cdot (\cdot)$
- $X \rightarrow \cdot (\cdot)$
- $X \rightarrow \cdot (X)$
- $X \rightarrow \cdot (\cdot)$
- $S \rightarrow X \cdot$
- $X \rightarrow (X)$
- $X \rightarrow (\cdot)$
- $X \rightarrow (X)$
- $X \rightarrow (\cdot)$
Potential Problem

- No lookahead
- Vulnerable to unnecessary conflicts
  - Shift/Reduce Conflicts (may reduce too soon in some cases)
  - Reduce/Reduce Conflicts
- Solution: Lookahead
  - Only for reductions - reduce only when next symbol can occur after nonterminal from production
  - Systematic lookahead, split states based on next symbol, action is always a function of next symbol
  - Can generalize to look ahead multiple symbols
Reduction-Only Lookahead Parsing

- If a state contains $A \rightarrow \beta$.
- Reduce by $A \rightarrow \beta$ only if next input symbol can follow $A$ in some derivation.
- Example Grammar
  
  $$S \rightarrow X \$ $$
  $$X \rightarrow a $$
  $$X \rightarrow a \ b$$
### Parser Without Lookahead

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>shift to s1</td>
<td>s3</td>
</tr>
<tr>
<td>s1</td>
<td>reduce(2)</td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>reduce(3)</td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td>error</td>
<td></td>
</tr>
</tbody>
</table>

### Grammar

- **S → \cdot X \cdot \$**
- **X → \cdot a**
- **X → \cdot a \cdot b**
- **S → X \cdot \$**
- **X → a \cdot**
- **X → a \cdot b**
- **X → a b**
- **S → X \$**
- **X → a**
- **X → a b**
Creating parse tables with reduction-only lookahead

• For each state
  • Transition to another state using a terminal symbol is a shift to that state (*shift to sn*) (same as before)
  • Transition to another state using a non-terminal is a goto that state (*goto sn*) (same as before)
  • If there is an item \( X \rightarrow \alpha \) in the state do a reduction with that production whenever the current input symbol \( T \) may follow \( X \) in some derivation (more precise than before)

• Eliminates useless reduce actions
New Parse Table

b never follows X in any derivation
resolve shift/reduce conflict to shift

<table>
<thead>
<tr>
<th>State</th>
<th>a ACTION</th>
<th>b ACTION</th>
<th>$ ACTION</th>
<th>Goto ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>shift to s1</td>
<td>error</td>
<td>error</td>
<td>goto s3</td>
</tr>
<tr>
<td>s1</td>
<td>reduce(2)</td>
<td>shift to s2</td>
<td>reduce(2)</td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>reduce(3)</td>
<td>reduce(3)</td>
<td>reduce(3)</td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td>error</td>
<td>error</td>
<td>accept</td>
<td></td>
</tr>
</tbody>
</table>

```
S → X \cdot S
X → \cdot a
X → \cdot a \cdot b

S \rightarrow X \cdot S
\]

S \rightarrow X \cdot S
\]

S \rightarrow X \cdot S
\]

X \rightarrow a \cdot b
S \rightarrow X \cdot S
```
More General Lookahead

- Items contain potential lookahead information, resulting in more states in finite state control.
- Item of the form \([A \rightarrow \alpha \cdot \beta \cdot T]\) says:
  - The parser has parsed an \(\alpha\).
  - If it parses a \(\beta\) and the next symbol is \(T\).
  - Then parser should reduce by \(A \rightarrow \alpha \beta\).

- In addition to current parser state, all parser actions are function of lookahead symbols.
Terminology

• Many different parsing techniques
  • Each can handle some set of CFGs
  • Categorization of techniques
Terminology

- Many different parsing techniques
  - Each can handle some set of CFGs
  - Categorization of techniques
Terminology

- Many different parsing techniques
  - Each can handle some set of CFGs
  - Categorization of techniques

- $L$ - parse from left to right
- $R$ - parse from right to left
Terminology

- Many different parsing techniques
  - Each can handle some set of CFGs
  - Categorization of techniques

- $L$ - leftmost derivation
- $R$ - rightmost derivation
Terminology

• Many different parsing techniques
  • Each can handle some set of CFGs
  • Categorization of techniques

• Number of lookahead characters
Terminology

• Many different parsing techniques
  • Each can handle some set of CFGs
  • Categorization of techniques

• Examples: LL(0), LR(1)

• This lecture
  • LR(0) parser
  • SLR parser – LR(0) parser augmented with follow information
Summary

- Parser generators – given a grammar, produce a parser
- Standard technique
  - Automatically build a pushdown automaton
  - Obtain a shift-reduce parser
    - Finite state control plus push down stack
    - Table driven implementation
- Conflicts: Shift/Reduce, Reduce/Reduce
- Use of lookahead to eliminate conflicts
  - SLR parsing (eliminates useless reduce actions)
  - LR(k) parsing (lookahead throughout parser)
Follow() sets in SLR Parsing

For each non terminal $A$, Follow($A$) is the set of terminals that can come after $A$ in some derivation.
Constraints for Follow()

• $ \in \text{Follow}(S)$, where $S$ is the start symbol
• If $A \rightarrow \alpha B \beta$ is a production then $\text{First}(\beta) \subseteq \text{Follow}(B)$
• If $A \rightarrow \alpha B$ is a production then $\text{Follow}(A) \subseteq \text{Follow}(B)$
• If $A \rightarrow \alpha B \beta$ is a production and $\beta$ derives $\epsilon$ then $\text{Follow}(A) \subseteq \text{Follow}(B)$
Algorithm for Follow

for all nonterminals $NT$
  
  $\text{Follow}(NT) = {}$

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

while Follow sets keep changing
  
  for all productions $A \rightarrow \alpha B \beta$
    
    $\text{Follow}(B) = \text{Follow}(B) \cup \text{First}(\beta)$

    if ($\beta$ derives $\varepsilon$) $\text{Follow}(B) = \text{Follow}(B) \cup \text{Follow}(A)$

for all productions $A \rightarrow \alpha B$
  
  $\text{Follow}(B) = \text{Follow}(B) \cup \text{Follow}(A)$
Augmenting Example with Follow

- Example Grammar for Follow

\[ S \rightarrow X \$
\[ X \rightarrow a
\[ X \rightarrow a \ b
\]

Follow(\( S \)) = \{ \$, \}

Follow(\( X \)) = \{ \$, \}
SLR Eliminates Shift/Reduce Conflict

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>a shift to s1, b error, $ error</td>
<td>goto s3</td>
</tr>
<tr>
<td>s1</td>
<td>reduce(2) shift to s2, reduce(2)</td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>reduce(3), reduce(3), reduce(3)</td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td>error, error, accept</td>
<td></td>
</tr>
</tbody>
</table>

S → • X $  
X → • a  
X → • a b

s0

s1

s2

s3

X → a b •

b \notin \text{Follow}(X)
Basic Idea Behind LR(1)

- Split states in LR(0) DFA based on lookahead
- Reduce based on item and lookahead
LR(1) Items

- Items will keep info on
  - production
  - right-hand-side position (the dot)
  - look ahead symbol
- LR(1) item is of the form \([A \rightarrow \alpha \cdot \beta \ T]\)
  - \(A \rightarrow \alpha \beta\) is a production
  - The dot in \(A \rightarrow \alpha \cdot \beta\) denotes the position
  - \(T\) is a terminal or the end marker ($$\$$)
Meaning of LR(1) Items

- Item \([A \rightarrow \alpha \cdot \beta \cdot T]\) means
  - The parser has parsed an \(\alpha\)
  - If it parses a \(\beta\) and the next symbol is \(T\)
  - Then parser should reduce by \(A \rightarrow \alpha \beta\)
• The grammar
  \[ S \rightarrow X$ \]
  \[ X \rightarrow (X) \]
  \[ X \rightarrow \varepsilon \]

• Terminal symbols
  • ‘(‘ ’)’
  • ‘$’

• End of input symbol

LR(1) Items

\[ [S \rightarrow \cdot X$ ] \]
\[ [S \rightarrow \cdot X$ ( ] \]
\[ [S \rightarrow \cdot X$ $ ] \]
\[ [S \rightarrow X\cdot$ $ ] \]
\[ [S \rightarrow X\cdot$ ( ] \]
\[ [S \rightarrow X\cdot$ $ ] \]
\[ [X \rightarrow \cdot (X) ] \]
\[ [X \rightarrow \cdot (X) ] \]
\[ [X \rightarrow \cdot (X) $ ] \]
\[ [X \rightarrow (\cdot X) ] \]
\[ [X \rightarrow (\cdot X) ] \]
\[ [X \rightarrow (\cdot X) $ ] \]
\[ [X \rightarrow (\cdot X) $ ] \]
\[ [X \rightarrow (\cdot X) ] \]
Creating a LR(1) Parser Engine

- Need to define Closure() and Goto() functions for LR(1) items

- Need to provide an algorithm to create the DFA

- Need to provide an algorithm to create the parse table
Closure algorithm

Closure(I)
  repeat
    for all items [A → α • X β c] in I
    for any production X → γ
      for any d ∈ First(βc)
        I = I ∪ { [X → • γ d] }
  until I does not change
Goto algorithm

Goto(I, X)

J = {} for any item [A → \alpha \cdot X \beta \ c] in I

J = J \cup \{[A → \alpha X \cdot \beta \ c]\}

return Closure(J)
Building the LR(1) DFA

• Start with the item \([S’] \rightarrow \cdot <S> \cdot \] $ I$
  • $I$ irrelevant because we will never shift $
• Find the closure of the item and make an state
• Pick a state $I$
  • for each item \([A \rightarrow \alpha \cdot X \beta \cdot c] \in I$
    • find $\text{Goto}(I, X)$
    • if $\text{Goto}(I, X)$ is not already a state, make one
    • Add an edge $X$ from state $I$ to $\text{Goto}(I, X)$ state
• Repeat until no more additions possible
Creating the parse tables

• For each LR(1) DFA state
  • Transition to another state using a terminal symbol is a shift to that state (*shift to sn*)
  • Transition to another state using a non-terminal symbol is a goto that state (*goto sn*)
  • If there is an item \([A \rightarrow \alpha \cdot a]\) in the state, action for input symbol a is a reduction via the production \(A \rightarrow \alpha\) (*reduce k*)
LALR(1) Parser

- Motivation
  - LR(1) parse engine has a large number of states
  - Simple method to eliminate states
- If two LR(1) states are identical except for the look ahead symbol of the items
  Then Merge the states
- Result is LALR(1) DFA
- Typically has many fewer states than LR(1)
- May also have more reduce/reduce conflicts
6.035 Computer Language Engineering
Spring 2010

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.