Announcements

- You may optionally work with a partner on the project
- Must work with the partner for the entire project
- Let me know by next Thursday who you are working with (if anyone), and under which username you will be submitting
- I’m trying to get the lectures posted online at least a day before class (some days I’m more successful than others!)

From last time: Lex (Flex)

- Commonly used Unix scanner generator (superseded by Flex)
- Has character classes and regular expressions like ScanGen but some key differences:
  - After each token is matched, calls user-defined “filter” function, which processes identified token before returning it to parser
  - Hence, no “Toss” facility (why?)
- No exception list
  - Instead, supports matching multiple regexps.
  - Matches longest token (i.e., doesn’t think if is IF ID(a))
  - In case of tie, returns earliest-defined regexp
  - To treat if as a reserved word instead of an identifier, define token IF before defining identifiers.

Lex operation

Parser

Example of Lex input on page 67 of textbook

Parsers

Terminology

- Grammar $G = (V_t, V_n, S, P)$
- $V_t$ is the set of terminals
- $V_n$ is the set of non-terminals
- $S$ is the start symbol
- $P$ is the set of productions
- Each production takes the form: $V_t \rightarrow \lambda | (V_t | V_n)^+$
- Grammar is context-free (why?)
- A simple grammar:
  $G = \{(a, b), (S, A, B), (S \rightarrow A B, S A \rightarrow A, A \rightarrow a, B \rightarrow B b, B \rightarrow b), S\}$

Agenda

- Terminology
- LL(1) Parsers
- Overview of LR Parsing

Thursday, September 10, 2009
Terminology

- $V$ is the vocabulary of a grammar, consisting of terminal ($V_t$) and non-terminal ($V_n$) symbols
- For our sample grammar:
  - $V_n = \{S, A, B\}$
    - Non-terminals are symbols on the LHS of a production
    - Non-terminals are constructs in the language that are recognized during parsing
  - $V_t = \{a, b\}$
    - Terminals are the tokens recognized by the scanner
    - They correspond to symbols in the text of the program

Productions (rewrite rules) tell us how to derive strings in the language

- Apply productions to rewrite strings into other strings
- We will use the standard BNF form

$P = \{
S \rightarrow A \; B \; $  
A \rightarrow A \; a 
A \rightarrow a
B \rightarrow B \; b 
B \rightarrow b 
\}$

Generating strings

- Given a start rule, productions tell us how to rewrite a non-terminal into a different set of symbols
- By convention, first production applied has the start symbol on the left, and there is only one such production

To derive the string “a a b b b” we can do the following rewrites:

$S \Rightarrow A \; B \; $  
$A \Rightarrow A \; a 
A \Rightarrow a 
B \Rightarrow B \; b 
B \Rightarrow b$

Strings are composed of symbols

- A A a a B b b A a is a string
- We will use Greek letters to represent strings composed of both terminals and non-terminals
- $L(G)$ is the language produced by the grammar $G$
- All strings consisting of only terminals that can be produced by $G$
- In our example, $L(G) = a+b+$
- All regular expressions can be expressed as grammars for context-free languages, but not vice-versa
- Consider: $a*b*$ (what is the grammar for this?)

Parse trees

- Tree which shows how a string was produced by a language
- Interior nodes of tree: non-terminals
- Children: the terminals and non-terminals generated by applying a production rule
- Leaf nodes: terminals

Leftmost derivation

- Rewriting of a given string starts with the leftmost symbol
- Exercise: do a leftmost derivation of the input program $F(V + V)$

using the following grammar:

<table>
<thead>
<tr>
<th>Production</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>Prefix ($E$)</td>
</tr>
<tr>
<td>$E$</td>
<td>$V$ Tail</td>
</tr>
<tr>
<td>Prefix</td>
<td>$F$</td>
</tr>
<tr>
<td>Prefix</td>
<td>$\lambda$</td>
</tr>
<tr>
<td>Tail</td>
<td>$+ E$</td>
</tr>
<tr>
<td>Tail</td>
<td>$\lambda$</td>
</tr>
</tbody>
</table>

- What does the parse tree look like?
**Rightmost derivation**

- Rewrite using the rightmost non-terminal, instead of the left
- What is the rightmost derivation of this string?

\[ F(V + V) \]

\[
\begin{align*}
E & \rightarrow \text{Prefix } (E) \\
E & \rightarrow V \text{Tail} \\
\text{Prefix} & \rightarrow F \\
\text{Prefix} & \rightarrow \lambda \\
\text{Tail} & \rightarrow + E \\
\text{Tail} & \rightarrow \lambda
\end{align*}
\]

**Top-down vs. Bottom-up parsers**

- Top-down parsers use left-most derivation
- Bottom-up parsers use right-looking parse
- Notation:
  - LL(1): Leftmost derivation with 1 symbol lookahead
  - LL(k): Leftmost derivation with k symbols lookahead
  - LR(1): Right-looking derivation with 1 symbol lookahead

**Micro in standard BNF**

1. Program
2. Statement-list
3. Statement
4. Statement-list
5. Statement
6. Statement
7. Statement
8. Id-list
9. IdTail
10. IdTail
11. Expression
12. Expression
13. Expression
14. Expression
15. Primary
16. Primary
17. Primary
18. Primary
19. Primary
20. Add-op
21. Add-op
22. System-goal

**Micro in standard BNF**

1. Program
2. Statement-list
3. Statement
4. Statement-list
5. Statement
6. Statement
7. Statement
8. Id-list
9. IdTail
10. IdTail
11. Expression
12. Expression
13. Expression
14. Expression
15. Primary
16. Primary
17. Primary
18. Primary
19. Primary
20. Add-op
21. Add-op
22. System-goal

**What is parsing**

- Parsing is recognizing members in a language specified/defined/generated by a grammar
- When a construct (corresponding to a production in a grammar) is recognized, a typical parser will take some action
- In a compiler, this action generates an intermediate representation of the program construct
- In an interpreter, this action might be to perform the action specified by the construct. Thus, if \( a + b \) is recognized, the value of \( a \) and \( b \) would be added and placed in a temporary variable
Another simple grammar

PROGRAM  \rightarrow\ begin \ STMTLIST\$  
STMTLIST  \rightarrow\ STMT\ ;\ STMTLIST  
STMTLIST  \rightarrow\ end  
STMT  \rightarrow\ id  
STMT  \rightarrow\ if\ (\ id)\ STMTLIST  

• A sentence in the grammar:
begin if (id) id ; end; end; end; $  
• What are the terminals and non-terminals of this grammar?

Parsing this grammar

PROGRAM  \rightarrow\ begin \ STMTLIST\$  
STMTLIST  \rightarrow\ STMT\ ;\ STMTLIST  
STMTLIST  \rightarrow\ end  
STMT  \rightarrow\ id  
STMT  \rightarrow\ if\ (\ id)\ STMTLIST  

• Note  
  • To parse STMT in STMTLIST \rightarrow STMT; STMTLIST, it is necessary to  
  parse either STMT \rightarrow id or STMT \rightarrow if ...  
  • Choose the production to parse by finding out if next token is if or id  
    • i.e., which production the next input token matches  
    • This is the first set of the production

Another example

S  \rightarrow A B$  
A  \rightarrow x a A  
A  \rightarrow y A  
A  \rightarrow \lambda  
B  \rightarrow b  

• Consider S  \rightarrow A B$  
• When parsing x a B$ we know from the goal production we need to match an A. The  
  next token is x, so we apply A \rightarrow x a A  
• The parser matches x, matches a and now needs to parse A again  
• How do we know which A to use? We need to use A \rightarrow \lambda  
  • When matching the right hand side of A \rightarrow \lambda, the next token comes from a non-  
    terminal that follows A (i.e., it must be b)  
• Tokens that can follow A are called the follow set of A

First and follow sets

• First(\alpha): the set of terminals that begin all strings that can be derived  
  from \alpha  
  • First(A) = \{x, y\}  
  • First(x A A) = \{x\}  
  • First (A B) = \{x, y, b\}  

• Follow(A): the set of terminals that can appear immediately after A in some partial derivation  
  • Follow(A) = \{b\}

First and follow sets

• First(\alpha) = \{a\ in V_t | \alpha \Rightarrow^\ast a \beta\} \cup \{\lambda | \text{if } \alpha \Rightarrow^\ast \lambda\}  
• Follow(A) = \{a\ in V_t | S \Rightarrow^\ast ... A a ... \} \cup \{\$ | \text{if } S \Rightarrow^\ast ... A \$\}  

S: start symbol  
a: a terminal symbol  
A: a non-terminal symbol  
\alpha, \beta: a string composed of terminals and non-terminals (typically, \alpha is the  
RHS of a production  
\Rightarrow: derived in 1 step  
\Rightarrow^\ast: derived in 0 or more steps  
\Rightarrow^\ast^\ast: derived in 1 or more steps

Computing first sets

• Terminal: First(\alpha) = \{a\}  
• Non-terminal: First(A)  
  • Look at all productions for A  
  • First(A) \supseteq (First(X_1), \ldots, X_n)  
  • If \lambda \in First(X_i), First(A) \supseteq (First(X_i), \ldots, \lambda)  
  • If \lambda is in First(X_i) for all i, then \lambda \in First(A)  
• Computing First(\alpha): similar procedure to computing First(\alpha)
Exercise

- What are the first sets for all the non-terminals in following grammar:

  \[
  S \rightarrow A \ B \ \$ \\
  A \rightarrow x \ a \ A \\
  A \rightarrow y \ a \ A \\
  A \rightarrow \lambda \\
  B \rightarrow b
  \]

Computing follow sets

- Follow(S) = \{\$\}
- To compute Follow(A):
  - Find productions which have A on rhs. Three rules:
    1. \( X \rightarrow \alpha \ A \ \beta : \text{Follow(A)} \supset (\text{First(}\beta\text{)} - \lambda) \)
    2. \( X \rightarrow \alpha \ A \ \beta : \text{if } \lambda \in \text{First(}\beta\text{)}, \text{Follow(A)} \supset \text{Follow(X)} \)
    3. \( X \rightarrow \alpha \ A : \text{Follow(A)} \supset \text{Follow(X)} \)
- Note: Follow(X) never has \( \lambda \) in it.

Exercise

- What are the follow sets for

  \[
  S \rightarrow A \ B \ \$ \\
  A \rightarrow x \ a \ A \\
  A \rightarrow y \ a \ A \\
  A \rightarrow \lambda \\
  B \rightarrow b
  \]

Towards parser generators

- Key problem: as we read the source program, we need to decide what productions to use
- Step 1: find the tokens that can tell which production \( P \) (of the form \( A \rightarrow X_1 X_2 \ldots X_m \)) applies

\[
\text{Predict}(P) = \\
\begin{cases} 
\text{First}(X_1 \ldots X_m) & \text{if } \lambda \notin \text{First}(X_1 \ldots X_m) \\
(\text{First}(X_1 \ldots X_m) - \lambda) \cup \text{Follow}(A) & \text{otherwise}
\end{cases}
\]
- If next token is in Predict(P), then we should choose this production

Parse tables

- Step 2: build a parse table
  - Given some non-terminal \( V_n \) (the non-terminal we are currently processing) and a terminal \( V_t \) (the lookahead symbol), the parse table tells us which production \( P \) to use (or that we have an error
  - More formally:
    \( T : V_n \times V_t \rightarrow P \cup \{\text{Error}\} \)

Building the parse table

- Start: \( T[A][t] = \text{initialize all fields to “error”} \)
  - foreach A:
    - foreach P with A on its lhs:
      - foreach t in Predict(P):
        \( T[A][t] = P \)
    - Exercise: build parse table for our toy grammar
      1. \( S \rightarrow A \ B \ \$ \)
      2. \( A \rightarrow x \ a \ A \)
      3. \( A \rightarrow y \ a \ A \)
      4. \( A \rightarrow \lambda \)
      5. \( B \rightarrow b \)
Recursive-descent parsers

- Given the parse table, we can create a program which generates recursive descent parsers
- Remember the recursive descent parser we saw for MICRO
- If the choice of production is not unique, the parse table tells us which one to take
- However, there is an easier method!

Stack-based parser for LL(1)

- Given the parse table, a stack-based algorithm is much simpler to generate than a recursive descent parser
- Basic algorithm:
  1. Push the RHS of a production onto the stack
  2. Pop a symbol, if it is a terminal, match it
  3. If it is a non-terminal, take its production according to the parse table and go to 1

Note: always start with start state

An example

- How would a stack-based parser parse: \( x \ a \ y \ a \ b \)

<table>
<thead>
<tr>
<th>Parse stack</th>
<th>Remaining input</th>
<th>Parser action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>( x \ a \ y \ a \ b ) $</td>
<td>predict 1</td>
</tr>
<tr>
<td>( A \ B )</td>
<td>( x \ a \ y \ a \ b ) $</td>
<td>predict 2</td>
</tr>
<tr>
<td>( x \ a \ B )</td>
<td>( x \ a \ y \ a \ b ) $</td>
<td>match(a)</td>
</tr>
</tbody>
</table>

1. \( S \to A \ B \) $ 
2. \( A \to x \ a \ A \) 
3. \( A \to y \ a \ A \) 
4. \( A \to \lambda \) 
5. \( B \to b \) 

An example

- How would a stack-based parser parse: \( x \ a \ y \ a \ b \)

<table>
<thead>
<tr>
<th>Parse stack</th>
<th>Remaining input</th>
<th>Parser action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>( x \ a \ y \ a \ b ) $</td>
<td>predict 1</td>
</tr>
<tr>
<td>( A \ B )</td>
<td>( x \ a \ y \ a \ b ) $</td>
<td>predict 2</td>
</tr>
<tr>
<td>( x \ a \ A \ B )</td>
<td>( x \ a \ y \ a \ b ) $</td>
<td>match(a)</td>
</tr>
</tbody>
</table>

1. \( S \to A \ B \) $ 
2. \( A \to x \ a \ A \) 
3. \( A \to y \ a \ A \) 
4. \( A \to \lambda \) 
5. \( B \to b \) 

An example

- How would a stack-based parser parse: \( x \ a \ y \ a \ b \)

<table>
<thead>
<tr>
<th>Parse stack</th>
<th>Remaining input</th>
<th>Parser action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>( x \ a \ y \ a \ b ) $</td>
<td>predict 1</td>
</tr>
<tr>
<td>( A \ B )</td>
<td>( x \ a \ y \ a \ b ) $</td>
<td>predict 2</td>
</tr>
<tr>
<td>( x \ a \ A \ B )</td>
<td>( x \ a \ y \ a \ b ) $</td>
<td>match(a)</td>
</tr>
<tr>
<td>( * \ A \ B )</td>
<td>( a \ y \ a \ b ) $</td>
<td>match(a)</td>
</tr>
</tbody>
</table>

1. \( S \to A \ B \) $ 
2. \( A \to x \ a \ A \) 
3. \( A \to y \ a \ A \) 
4. \( A \to \lambda \) 
5. \( B \to b \)
### An example

- How would a stack-based parser parse: `x a y a b`

<table>
<thead>
<tr>
<th>Parse stack</th>
<th>Remaining input</th>
<th>Parser action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td><code>x a y b $</code></td>
<td>predict 1</td>
</tr>
<tr>
<td>A B $</td>
<td><code>x a y b $</code></td>
<td>predict 2</td>
</tr>
<tr>
<td><code>x a A B $</code></td>
<td><code>x a y b $</code></td>
<td>match(x)</td>
</tr>
<tr>
<td>A B $</td>
<td><code>y a b $</code></td>
<td>match(a)</td>
</tr>
<tr>
<td><code>A B $</code></td>
<td><code>y a b $</code></td>
<td>match(y)</td>
</tr>
<tr>
<td><code>B $</code></td>
<td><code>b $</code></td>
<td>predict 5</td>
</tr>
</tbody>
</table>

#### 1. \( S \rightarrow A B $ \)

#### 2. \( A \rightarrow x A \)

#### 3. \( A \rightarrow y A \)

#### 4. \( A \rightarrow \lambda \)

#### 5. \( B \rightarrow b \)
An example

- How would a stack-based parser parse:
  \[ x a y a b \]

<table>
<thead>
<tr>
<th>Parse stack</th>
<th>Remaining input</th>
<th>Parser action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>x a y a b $</td>
<td>predict 1</td>
</tr>
<tr>
<td>A B $</td>
<td>x a y a b $</td>
<td>predict 2</td>
</tr>
<tr>
<td>x A B $</td>
<td>x a y a b $</td>
<td>match(a)</td>
</tr>
<tr>
<td>A B $</td>
<td>y a b $</td>
<td>predict 3</td>
</tr>
<tr>
<td>y A B $</td>
<td>y a b $</td>
<td>match(y)</td>
</tr>
<tr>
<td>z A B $</td>
<td>a b $</td>
<td>match(a)</td>
</tr>
<tr>
<td>A B $</td>
<td>b $</td>
<td>predict 4</td>
</tr>
<tr>
<td>B $</td>
<td>b $</td>
<td>predict 5</td>
</tr>
<tr>
<td>b $</td>
<td>b $</td>
<td>match(b)</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>Done!</td>
</tr>
</tbody>
</table>

LL(k) parsers

- Can use similar techniques for LL(k) parsers
- Use more than one symbol of look-ahead to distinguish productions
- Why might this be bad?

Dealing with semantic actions

- Recall: we can annotate a grammar with action symbols
- Tell the parser to invoke a semantic action routine
- Can simply push action symbols onto stack as well
- When popped, the semantic action routine is called

Non-LL(1) grammars

- Not all grammars are LL(1)!
- Consider
  \[
  \text{<stmt>} \rightarrow \text{if <expr> then } \text{<stmt list> endif}
  \]
  \[
  \text{<stmt>} \rightarrow \text{if <expr> then } \text{<stmt list> else } \text{<stmt list> endif}
  \]
- This is not LL(1) (why?)
- We can turn this in to
  \[
  \text{<stmt>} \rightarrow \text{if <expr> then } \text{<stmt list> <if suffix>}
  \]
  \[
  \text{<if suffix>} \rightarrow \text{endif}
  \]
  \[
  \text{<if suffix>} \rightarrow \text{else } \text{<stmt list> endif}
  \]

Left recursion

- Left recursion is a problem for LL(1) parsers
- LHS is also the first symbol of the RHS
- Consider:
  \[
  E \rightarrow E + T
  \]
- What would happen with the stack-based algorithm?

Removing left recursion

- \[
  E \rightarrow E + T
  \]
- \[
  E \rightarrow T
  \]
- \[
  E \rightarrow E1 Etail
  \]
- \[
  E1 \rightarrow T
  \]
- \[
  Etail \rightarrow + T Etail
  \]
- \[
  Etail \rightarrow \lambda
  \]

Algorithm on page 125
Are all grammars LL(1)?

- No! Consider the if-then-else problem
- If x then y else z
- Problem: else is optional
- If a then if b then c else d
  - Which if does the else belong to?
- This is analogous to a "bracket language": \([\)] (i \geq j)

\[
\begin{align*}
S & \rightarrow [SC] \\
S & \rightarrow \lambda \\
C & \rightarrow ] \\
C & \rightarrow \lambda
\end{align*}
\]

This can be parsed: SS\lambda C or SS\lambda (it's ambiguous!)

Solving the if-then-else problem

- The ambiguity exists at the language level. To fix, we need to define the semantics properly
- \"[\) matches nearest unmatched \"[\"
- This is the rule C uses for if-then-else
- What if we try this?

\[
\begin{align*}
S & \rightarrow [S] \\
S & \rightarrow S1 \\
S1 & \rightarrow [S1] \\
S1 & \rightarrow \lambda
\end{align*}
\]

This grammar is still not LL(1) (or LL(k) for any k!)

Two possible fixes

- If there is an ambiguity, prioritize one production over another
  - e.g., if C is on the stack, always match \"[\) before matching \"[\"

\[
\begin{align*}
S & \rightarrow [SC] \\
S & \rightarrow \lambda \\
C & \rightarrow ] \\
C & \rightarrow \lambda
\end{align*}
\]

- Another option: change the language!
  - e.g., all if-statements need to be closed with an endif

\[
\begin{align*}
S & \rightarrow \text{if } S E \\
S & \rightarrow \text{other} \\
E & \rightarrow \text{else } S \text{ endif} \\
E & \rightarrow \text{endif}
\end{align*}
\]

Parsing if-then-else

- What if we don't want to change the language?
  - C does not require \{\} to delimit single-statement blocks
  - To parse if-then-else, we need to be able to look ahead at the entire rhs of a production before deciding which production to use
  - In other words, we need to determine how many \"[\) to match before we start matching \"[\"
  - \textbf{LR parsers} can do this!

LR Parsers

- Parser which does a Left-to-right, Right-most derivation
  - Rather than parse top-down, like LL parsers do, parse bottom-up, starting from leaves
  - Basic idea: put tokens on a stack until an entire production is found
  - Issues:
    - Recognizing the endpoint of a production
    - Finding the length of a production (RHS)
    - Finding the corresponding nonterminal (the LHS of the production)

Data structures

- At each state, given the next token,
  - A \textit{goto table} defines the successor state
  - An \textit{action table} defines whether to
    - \textit{shift} – put the next state and token on the stack
    - \textit{reduce} – an RHS is found; process the production
    - \textit{terminate} – parsing is complete
Example

- Consider the simple grammar:
  <program> → begin <stmts> end $
  <stmts> → SimpleStmt ; <stmts>
  <stmts> → begin <stmts> end ; <stmts>
  <stmts> → λ
- Shift-reduce driver algorithm on page 142

Action and goto tables

<table>
<thead>
<tr>
<th></th>
<th>begin</th>
<th>end</th>
<th>SimpleStmt</th>
<th>$</th>
<th>&lt;program&gt;</th>
<th>&lt;stmts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5 / 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5 / 4</td>
<td>R4</td>
<td>5 / 5</td>
<td>5 / 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5 / 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5 / 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5 / 4</td>
<td>R4</td>
<td>5 / 5</td>
<td>5 / 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5 / 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5 / 4</td>
<td>R4</td>
<td>5 / 5</td>
<td>5 / 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5 / 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5 / 9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>5 / 4</td>
<td>R4</td>
<td>5 / 6</td>
<td>5 / 11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>R3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example

- Parse: begin SimpleStmt ; SimpleStmt ; end $

<table>
<thead>
<tr>
<th>Step</th>
<th>Parse Stack</th>
<th>Remaining Input</th>
<th>Parser Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>begin ; ; end $</td>
<td>Shift 1</td>
</tr>
<tr>
<td>2</td>
<td>0 1</td>
<td>$ ; ; end $</td>
<td>Shift 5</td>
</tr>
<tr>
<td>3</td>
<td>0 1 5</td>
<td>$ ; ; end $</td>
<td>Shift 6</td>
</tr>
<tr>
<td>4</td>
<td>0 1 5 6</td>
<td>$ ; end $</td>
<td>Shift 5</td>
</tr>
<tr>
<td>5</td>
<td>0 1 5 6 5</td>
<td>; end $</td>
<td>Shift 6</td>
</tr>
<tr>
<td>6</td>
<td>0 1 5 6 5 6</td>
<td>end $</td>
<td>Reduce 4 (goto 10)</td>
</tr>
<tr>
<td>7</td>
<td>0 1 5 6 5 6 10</td>
<td>end $</td>
<td>Reduce 2 (goto 10)</td>
</tr>
<tr>
<td>8</td>
<td>0 1 5 6 10</td>
<td>end $</td>
<td>Reduce 2 (goto 2)</td>
</tr>
<tr>
<td>9</td>
<td>0 1 2</td>
<td>end $</td>
<td>Shift 3</td>
</tr>
<tr>
<td>10</td>
<td>0 1 2 3</td>
<td>$</td>
<td>Accept</td>
</tr>
</tbody>
</table>

Announcements

- I will be out of town on Tuesday (9/15)
- Class will be covered by Professor Midkiff

LR Parsers

- Basic idea:
  - **shift** tokens onto the stack. At any step, keep the set of productions that could generate the read-in tokens
  - **reduce** the RHS of recognized productions to the corresponding non-terminal on the LHS of the production. Replace the RHS tokens on the stack with the LHS non-terminal.

LR(k) parsers

- LR(0) parsers
  - No lookahead
  - Predict which action to take by looking only at the symbols currently on the stack
- LR(k) parsers
  - Can look ahead k symbols
  - Most powerful class of deterministic bottom-up parsers
- LR(1) and variants are the most common parsers
**Terminology for LR parsers**

- Configuration: a production augmented with a “•”
  
  \[ A \rightarrow X_1 \ldots X_i • X_{i+1} \ldots X_j \]

- The “•” marks the point to which the production has been recognized. In this case, we have recognized \(X_1 \ldots X_i\).

- Configuration set: all the configurations that can apply at a given point during the parse:
  
  \[ A \rightarrow B • CD \]
  
  \[ A \rightarrow B • GH \]
  
  \[ T \rightarrow B • Z \]

- Idea: every configuration in a configuration set is a production that can possibly be matched

**Configuration closure set**

- Include all the configurations necessary to recognize the next symbol after the “•”
  
  \[ \text{closure}_0(\text{configuration set}) \] defined on page 146

- Example:
  
  \[ \begin{align*}
  S & \rightarrow E \$
  E & \rightarrow E + T | T \\
  T & \rightarrow \text{ID} | (E)
  \end{align*} \]

  \[ \text{closure}_0(\{S \rightarrow E \$\}) = \{
  S \rightarrow E \$
  E \rightarrow E + T \\
  E \rightarrow T \\
  T \rightarrow \text{ID} \\
  T \rightarrow (E)
  \} \]

**Successor configuration set**

- Starting with the initial configuration set
  
  \[ s_0 = \text{closure}_0(\{S \rightarrow • \alpha \$\}) \]

  an LR(0) parser will find the successor given the next symbol \(X\).

- \(X\) can be either a terminal (the next token from the scanner) or a non-terminal (the result of applying a reduction).

- Determining the successor \(s' = \text{go}_0(s, X)\):
  
  - For each configuration in \(s\) of the form \(A \rightarrow \beta • X \gamma\) add \(A \rightarrow \beta • X \gamma \) to \(t\)
  
  - \(s' = \text{closure}_0(t)\)

**CFSM**

- CFSM = Characteristic Finite State Machine

- Nodes are configuration sets (starting from \(s_0\))

- Arcs are \(\text{go}_0\) relationships

**Building the goto table**

- We can just read this off from the CFSM

<table>
<thead>
<tr>
<th>Symbol</th>
<th>ID</th>
<th>$</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Building the action table**

- Given the configuration set \(s\):
  
  - **We shift** if the next token matches a terminal after the “•” in some configuration
    
    \[ A \rightarrow \alpha • \beta \in s \text{ and } \alpha \in V_0, \text{ else error} \]
  
  - **We reduce** production \(P\) if the “•” is at the end of a production
    
    \[ B \rightarrow \alpha • \in s \text{ where production } P \text{ is } B \rightarrow \alpha \]

- Extra actions:
  
  - **shift** if goto table transitions between states on a non-terminal
  
  - **accept** if we are about to shift $
**Action table**

<table>
<thead>
<tr>
<th>State</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$</td>
</tr>
<tr>
<td>1</td>
<td>R2</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

**Alternate representation**

- Some books represent goto and action tables differently.
  - Action table only has columns for terminals, and consists of two kinds of actions:
    - $shift + state$ shift and move to a $state$
    - $reduce + rule$ reduce according to $rule$
  - Goto table only has columns for non-terminals
  - Specifies which state to go to after reducing

**Conflicts in action table**

- For LR(0) grammars, the action table entries are unique: from each state, can only shift or reduce
- But other grammars may have conflicts
  - Reduce/reduce conflicts: multiple reductions possible from the given configuration
  - Shift/reduce conflicts: we can either shift or reduce from the given configuration

**Shift/reduce example**

- Consider the following grammar:
  
  \[
  S \rightarrow A \ y \\
  A \rightarrow \lambda | x
  \]

  This leads to the following initial configuration set:

  \[
  S \rightarrow \bullet A \ y \\
  A \rightarrow \bullet x \\
  A \rightarrow \lambda \bullet
  \]

- Can shift or reduce here

**Lookahead**

- Can resolve reduce/reduce conflicts and shift/reduce conflicts by employing lookahead
- Looking ahead one (or more) tokens allows us to determine whether to shift or reduce
  - (cf how we resolved ambiguity in LL(1) parsers by looking ahead one token)
- Note that it is possible to create an LR(0) grammar for any LR(k) grammar (as long as we can determine the end of a program), but it may be very complex!

**LR(1) parsing**

- Configurations in LR(1) look similar to LR(0), but they are extended to include a lookahead symbol
  
  \[
  A \rightarrow X_1 \ \cdots \ X_i \ \bullet \ X_{i+1} \ \cdots \ X_j \ \lambda \ (where \ \lambda \in \ V_t \cup \lambda)
  \]

  If two configurations differ only in their lookahead component, we combine them

  \[
  A \rightarrow X_1 \ \cdots \ X_i \ \bullet \ X_{i+1} \ \cdots \ X_j \ \{l_1 \ \cdots \ l_m\}
  \]
Building configuration sets

- To close a configuration
  \( B \rightarrow \alpha \cdot A \beta / \)
- Add all configurations of the form \( A \rightarrow \gamma \cdot u \) where \( u \in \text{First}(\beta) \)
- Intuition: the parse could apply the production for \( A \), and the lookahead after we apply the production should match the next token that would be produced by \( B \)

\[
\text{Example}
\]
\[
\text{closure}_1(\{S \rightarrow E \cdot (\lambda)\}) =
\]
\[
\begin{align*}
S &\rightarrow E \cdot (\lambda) \\
E &\rightarrow E + T | T \\
T &\rightarrow \text{ID} | (E)
\end{align*}
\]

\[
\text{Example}
\]
\[
\text{closure}_1(\{S \rightarrow E \cdot (\lambda)\}) =
\]
\[
\begin{align*}
S &\rightarrow E \cdot (\lambda) \\
E &\rightarrow E + T | T \\
T &\rightarrow \text{ID} | (E)
\end{align*}
\]
Example

\[
\text{closure1}(S \rightarrow E \cdot S, (\lambda)) =
\]

<table>
<thead>
<tr>
<th>Production</th>
<th>S \rightarrow E \cdot S, (\lambda)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E \rightarrow E + T, (\lambda)</td>
<td></td>
</tr>
<tr>
<td>E \rightarrow T, (\lambda)</td>
<td></td>
</tr>
<tr>
<td>T \rightarrow ID, (\lambda)</td>
<td></td>
</tr>
<tr>
<td>T \rightarrow \ast (\lambda), (\lambda)</td>
<td></td>
</tr>
</tbody>
</table>

Example

\[
\text{closure1}(S \rightarrow E \cdot S, (\lambda)) =
\]

<table>
<thead>
<tr>
<th>Production</th>
<th>S \rightarrow E \cdot S, (\lambda)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E \rightarrow E + T, (\lambda)</td>
<td></td>
</tr>
<tr>
<td>E \rightarrow T, (\lambda)</td>
<td></td>
</tr>
<tr>
<td>T \rightarrow ID, (\lambda)</td>
<td></td>
</tr>
<tr>
<td>T \rightarrow \ast (\lambda), (\lambda)</td>
<td></td>
</tr>
</tbody>
</table>

Example

\[
\text{closure1}(S \rightarrow E \cdot S, (\lambda)) =
\]

<table>
<thead>
<tr>
<th>Production</th>
<th>S \rightarrow E \cdot S, (\lambda)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E \rightarrow E + T, (\lambda)</td>
<td></td>
</tr>
<tr>
<td>E \rightarrow T, (\lambda)</td>
<td></td>
</tr>
<tr>
<td>T \rightarrow ID, (\lambda)</td>
<td></td>
</tr>
<tr>
<td>T \rightarrow \ast (\lambda), (\lambda)</td>
<td></td>
</tr>
</tbody>
</table>

Example

\[
\text{closure1}(S \rightarrow E \cdot S, (\lambda)) =
\]

<table>
<thead>
<tr>
<th>Production</th>
<th>S \rightarrow E \cdot S, (\lambda)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E \rightarrow E + T, (\lambda)</td>
<td></td>
</tr>
<tr>
<td>E \rightarrow T, (\lambda)</td>
<td></td>
</tr>
<tr>
<td>T \rightarrow ID, (\lambda)</td>
<td></td>
</tr>
<tr>
<td>T \rightarrow \ast (\lambda), (\lambda)</td>
<td></td>
</tr>
</tbody>
</table>

Example

\[
\text{closure1}(S \rightarrow E \cdot S, (\lambda)) =
\]

<table>
<thead>
<tr>
<th>Production</th>
<th>S \rightarrow E \cdot S, (\lambda)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E \rightarrow E + T, (\lambda)</td>
<td></td>
</tr>
<tr>
<td>E \rightarrow T, (\lambda)</td>
<td></td>
</tr>
<tr>
<td>T \rightarrow ID, (\lambda)</td>
<td></td>
</tr>
<tr>
<td>T \rightarrow \ast (\lambda), (\lambda)</td>
<td></td>
</tr>
</tbody>
</table>

Example

\[
\text{closure1}(S \rightarrow E \cdot S, (\lambda)) =
\]

<table>
<thead>
<tr>
<th>Production</th>
<th>S \rightarrow E \cdot S, (\lambda)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E \rightarrow E + T, (\lambda)</td>
<td></td>
</tr>
<tr>
<td>E \rightarrow T, (\lambda)</td>
<td></td>
</tr>
<tr>
<td>T \rightarrow ID, (\lambda)</td>
<td></td>
</tr>
<tr>
<td>T \rightarrow \ast (\lambda), (\lambda)</td>
<td></td>
</tr>
</tbody>
</table>

Building goto and action tables

- The function `goto1` (configuration-set, symbol) is analogous to `goto0` (configuration-set, symbol) for LR(0).
- Build goto table in the same way as for LR(0).
- Key difference: the action table.

\[
\text{action[s][x]} =
\]

- `reduce` when • is at end of configuration and x \in lookahead set of configuration

\[
A \rightarrow \alpha \cdot \{ \_ \_ \_ \_ \} \in s
\]

- `shift` when • is before x

\[
A \rightarrow \beta \cdot x \ y \in s
\]
**Problems with LR(1) parsers**

- LR(1) parsers are very powerful ...
- But the table size is much larger than LR(0) — as much as a factor of $|V_t|$ (why?)
- Example: Algol 60 (a simple language) includes several thousand states!
- Storage efficient representations of tables are an important issue

**Solutions to the size problem**

- Different parser schemes
- SLR (simple LR): build an CFSM for a language, then add lookahead wherever necessary (i.e., add lookahead to resolve shift/reduce conflicts)
- What should the lookahead symbol be?
- To decide whether to reduce using production $A \rightarrow \alpha$, use Follow($A$)
- LALR: merge LR states in certain cases (we won’t discuss this)

**Semantic actions**

- Recall: in LL parsers, we could integrate the semantic actions with the parser
- Why? Because the parser was **predictive**
- Why doesn’t that work for LR parsers?
  - Don’t know which production is matched until parser reduces
  - For LR parsers, we put semantic actions at the end of productions
  - May have to rewrite grammar to support all necessary semantic actions