Reprise on Languages and Grammars

- A language is a set (usually infinite) of strings, also known as sentences.
- Each string consists of a sequence of symbols taken from some alphabet.
- You cannot define a language by listing the strings.
- You can define a language by a grammar — a finite set of rules for generating the strings.
- A grammar, G, is a quadruple \((V_N, V_T, P, S)\) where:
  - \(V_N\) is a set of symbols called nonterminals.
  - \(V_T\) is a set of symbols called terminals, the "words" of the language.
  - \(P\) is a set of Productions — the rules for the formation of sentences.
  - \(S \in V_N\) is the Start symbol — the starting point.

Different kinds of Grammars

- Grammars have a set of productions.
- Most general form is \(\alpha \rightarrow \beta\).
- Where \(\alpha\) is a string including a non-terminal and \(\beta\) is any string (of terminals and non-terminals).
- Different kinds of grammars constrain the productions:
  - TYPE 1: \(|\alpha| \leq |\beta|\)
  - TYPE 2: \(\alpha\) consists only of a non-terminal.
  - TYPE 3: \(\alpha\) consists only of a non-terminal and \(\beta\) consists only of a terminal or a terminal followed by a non-terminal.

Reprise on Languages and Machines

- We can also define a language by specifying a machine that will recognise sentences of the language.
- The general structure of these machines is:
  - Input tape with string to be recognised.
  - Finite state control i.e. a "CPU".
  - Memory.
  - Read head.

Different kinds of machines

- The kind of auxiliary memory determines the kinds of languages that can be recognised:

<table>
<thead>
<tr>
<th>Aux memory</th>
<th>Kind of machine</th>
<th>Kind of grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Finite state aut.</td>
<td>TYPE3</td>
</tr>
<tr>
<td>Stack</td>
<td>Push down aut.</td>
<td>TYPE2</td>
</tr>
<tr>
<td>Tape bounded</td>
<td>Linear bounded aut.</td>
<td>TYPE1</td>
</tr>
<tr>
<td>by input length</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unbounded tape</td>
<td>Turing machine</td>
<td>TYPE0</td>
</tr>
</tbody>
</table>
Key result for Push Down Automata

- The class of languages that can be recognised by push down automata is the same as the class of languages that can be generated by TYPE2 grammars.

- This can be proved/demonstrated by showing how to map between the two kinds of specifications.

Consider a TYPE2 grammar with start symbol S:
- introduce an initial state $q_0$, a processing state $q_1$, and a final state $q_2$.
- for start symbol S, add a transition:
- for productions $A \rightarrow \alpha$, add a transition:
- for terminals $x$, add a transition:
- add a final transition:

Example

- Consider the earlier example of the language $a^nb^n$ where $n \geq 1$.
- Consider the grammar:
  $\{\{S,A\}, \{a,b\}, P, S\}$
  where $P$ consists of the productions:
  $S \rightarrow aAb$  \hspace{1cm} $A \rightarrow \varepsilon$  \hspace{1cm} $A \rightarrow aAb$
- The push down automaton would be:

Example

- Consider the earlier example of the language $a^nb^n$ where $n \geq 1$ and different kind of stack machine.
- Consider the grammar:
  $\{\{S,A\}, \{a,b\}, P, S\}$
  where $P$ consists of the productions:
  $S \rightarrow aAb$  \hspace{1cm} $A \rightarrow \varepsilon$  \hspace{1cm} $A \rightarrow aAb$
- The push down automaton would be:

Extended Pushdown Automata - EPDA

- The above stack machine is different in having the top of stack on the right (as opposed to the left).
- It is also different in removing multiple symbols off the stack at once — hence the name “Extended”.
- It essentially parses in a bottom up fashion — the right hand sides of productions are accumulated on the stack — a reduction leads to the start symbol.
Example of EPDA

- We can demonstrate acceptance of the string aaabbb with a sequence of configurations, each of which indicates the current state, unprocessed input and stack contents (top of stack on the right!!!):

  \[
  \begin{align*}
  (q_0, \text{aaabbb}, Z) & \rightarrow (q_0, \text{aabbb}, Za) & \# \text{used } A \rightarrow \varepsilon \\
  & \rightarrow (q_0, \text{abbb}, Zaa) & \# \text{used } A \rightarrow aAb \\
  & \rightarrow (q_0, bbb, ZaaaA) & \# \text{used } S \rightarrow aAb \\
  & \rightarrow (q_0, bb, ZaaaAb) & \# \text{used } A \rightarrow aAb \\
  & \rightarrow (q_0, b, ZaaAb) & \# \text{used } A \rightarrow aAb \\
  & \rightarrow (q_0, -, ZaAb) & \# \text{used } S \rightarrow aAb \\
  & \rightarrow (q_1, -, -) \\
  \end{align*}
  \]

Accept!

Key result for Extended PDA

- The class of languages that can be recognised by extended push down automata is the same as the class of languages that can be generated by TYPE2 grammars.

- This can be proved/demonstrated by showing how to map between the two kinds of specifications.

- Consider a TYPE2 grammar with start symbol S:
  - introduce an initial state \(q_0\), a final state \(q_1\)
  - for start symbol S, add a transition
  - for productions \(A \rightarrow \alpha\), add a transition:
  - for terminals \(x\), add a transition:

Non-determinism

- In the PDA model, the non-deterministic choice is to choose a possible expansion for a non-terminal
  - with a non-terminal on top of stack
  - which alternative expansion to choose?

- In the EPDA model, the non-deterministic choice is whether to shift or reduce, and which reduction to use
  - do you push a terminal onto the stack (always possible)
  - if the top of stack string matches the right hand side of a production, do we reduce by that production?
  - what if there is more than one alternative?
Deterministic bottom-up parsing

- In order to achieve deterministic parsing decisions, we intersperse the terminals and non-terminals on the stack with symbols that indicate a more precise context information.
- Then the parsing action tables are given in the form:
  
  \[ \text{Action}(X,a) = \begin{cases} \text{shift} n & / \text{reduce} n & / \text{accept} / \text{error} \\ \text{Goto}(X,Y) = \text{symbol to push onto stack after reduction} & & & & & & & & \end{cases} \]
  
  - \( X = \text{TOS symbol, a = input terminal} \)
  - \( \text{shift} n = \text{shift terminal a + symbol n onto the stack} \)
  - \( \text{reduce} n = \text{reduce by production n} \)
  - \( \text{accept} / \text{error} = \text{accept string / report a parsing error} \)
  - \( Y = \text{non-terminal on LHS of applied production} \)

Example: Expressions

- Consider parsing: \( x + x \times x \)
  
  \[
  \begin{array}{c}
  0, x+x\times x$ \\
  0x5, +x\times x$ & # action was s5 \\
  0F3, +x\times x$ & # action was r6 \\
  0T2, +x\times x$ & # action was r4 \\
  0E1, +x\times x$ & # action was r2 \\
  0E1+6, x\times x$ & # action was s6 \\
  0E1+6x5, \times x$ & # action was s5 \\
  (O1+6F3, \times x$ & # action was r6 \\
  0E1+6T9, \times x$ & # action was r4 \\
  0E1+6T9*7, x$ & # action was s7 \\
  0E1+6T9*7x5, $ & # action was s5 \\
  (O1+6T9*7F10, $ & # action was r6 \\
  0E1+6T9, $ & # action was r3 \\
  0E1, $ & # action was 1 \\
  1 & accept \\
  \end{array}
  \]

Note that applying a reduction is a complex operation:

- For: \( (0E1+6T9*7F10, $) \) action is r3
- Production 3 is \( T \rightarrow T \times F \)
- This production has 3 symbols on the RHS
- We need to remove 6 symbols from the stack
  - the grammar symbols and the corresponding stack symbols
- This leaves \( 0E1+6 \) on the stack with symbol 6 on top
- The LHS of the production is T
- The entry goto(6,T) indicates 9
- So, the stack becomes \( 0E1+6T9 \)
Observations

- The parse was deterministic without changing the grammar
  - bottom-up parsing can cope with more grammars (and languages) without change than top-down parsing
- The table is quite large by comparison
  - attempts to reduce the table size differentiate the various flavours of bottom up parsing algorithms
- The question is what do the stack symbols represent and how do we generate them?

Bottom-up parsing (1)

- The stack symbols keep track of the possible positions in productions

  $E \rightarrow E + T \mid T$

  $(1,0) \rightarrow (1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (2,0) \rightarrow (2,1)$

  - A symbol on the stack indicating position $(1,1)$ tells us that ‘+’ is a valid symbol and can be pushed onto the stack which takes us to position $(1,2)$

Bottom-up parsing (2)

- The stack symbols keep track of the possible sets of positions in productions

  $E \rightarrow E + T \mid T$

  $(1,0) \rightarrow (1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (2,0) \rightarrow (2,1)$

- Position $(1,0)$ cannot be considered in isolation
  - we need to consider position $(2,0)$ as well
  - the start of an $E$ could equally well be the start of a $T$
- This is known as a **closure** operation

Bottom-up parsing (3)

- The stack symbols keep track of the possible sets of positions in productions **together with the possible follower symbols**

  $E \rightarrow E + T \mid T$

  $(1,0) \rightarrow (1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (2,0) \rightarrow (2,1)$

  - If the goal symbol is $E$, which is followed by $\$\$ (EOF), then we start with position $(0,0)$ with $\$$, i.e. $(0,0,\$$)$
  - Closure will add items $(1,0,\$), (2,0,\$)$
  - Closure will add items $(1,0,+)\$, (2,0,+) — expansions of $E$ with + following — giving us $(0,0,\$), (1,0,\$), (2,0,\$+)$
Bottom-up parsing (4)

- The stack symbols keep track of the possible sets of positions in productions and the follower symbols

\[ E \rightarrow E + T | T \]

- Positions at end of productions will have associated reduce actions for each of the follower symbols
  - e.g. \((1,3,\$+)\) will have “r1” actions for \$ and +

---

Example: Expressions

<table>
<thead>
<tr>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0 = (0,0,$), (1,0,$+), (2,0,$+), (3,0,$+), (4,0,$+), (5,0,$+), (6,0,$+)</td>
<td>s5 s4 E T F 1 2 3</td>
</tr>
<tr>
<td>(1 = (0,1,$), (1,1,$)</td>
<td>s6 acc</td>
</tr>
<tr>
<td>(2 = (2,1,$+), (3,1,$+)</td>
<td>r2 s7 s2</td>
</tr>
<tr>
<td>(3 = (4,1,$+)</td>
<td>r4 r4 r4</td>
</tr>
<tr>
<td>(4 = (5,1,$+), (1,0,$+), (2,0,$+), (3,0,$+), (4,0,$+), (5,0,$+), (6,0,$+)</td>
<td>s10 s9 E T F 8 2 3</td>
</tr>
<tr>
<td>(5 = (6,1,$+)</td>
<td>r6 r6 r6</td>
</tr>
<tr>
<td>(6 = (1,2,$+), (3,0,$+), (4,0,$+), (5,0,$+), (6,0,$+)</td>
<td>s5 s4 F E T F 11 3</td>
</tr>
<tr>
<td>(7 = (3,2,$+), (5,0,$+), (6,0,$+)</td>
<td>s5 s4</td>
</tr>
<tr>
<td>(8 = (5,2,$+), (1,1,$)</td>
<td>s14 s13</td>
</tr>
<tr>
<td>(9 = (5,1,$+), (1,0,$+), (2,0,$+), (3,0,$+), (4,0,$+), (5,0,$+), (6,0,$+)</td>
<td>s10 s9 E T F 15 16 17</td>
</tr>
</tbody>
</table>

---

Example: Expressions

<table>
<thead>
<tr>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10 = (6,1,$)</td>
<td>r6 r6 r6</td>
</tr>
<tr>
<td>(11 = (1,3,$+), (3,1,$+)</td>
<td>r1 s7 r1</td>
</tr>
<tr>
<td>(12 = (3,3,$+)</td>
<td>r3 r3 r3</td>
</tr>
<tr>
<td>(13 = (5,3,$+)</td>
<td>r5 r5 r5</td>
</tr>
<tr>
<td>(14 = (1,2,$+), (3,0,$+), (4,0,$+), (5,0,$+), (6,0,$+)</td>
<td>s10 s9 E T F F E T F 18 17</td>
</tr>
<tr>
<td>(15 = (5,2,$+), (1,1,$)</td>
<td>s14 s19</td>
</tr>
<tr>
<td>(16 = (2,1,$), (3,1,$+)</td>
<td>r2 s20 r2</td>
</tr>
<tr>
<td>(17 = (4,1,$+)</td>
<td>r4 r4 r4</td>
</tr>
<tr>
<td>(18 = (1,3,$+), (3,1,$+)</td>
<td>r1 s20 r1</td>
</tr>
<tr>
<td>(19 = (5,3,$+)</td>
<td>r5 r5 r5</td>
</tr>
<tr>
<td>(20 = (3,2,$+), (5,0,$+), (6,0,$+)</td>
<td>s10 s9 E T F F E T F 21</td>
</tr>
<tr>
<td>(21 = (3,3,$+)</td>
<td>r3 r3 r3</td>
</tr>
</tbody>
</table>
LR parsing table construction (1)

```pascal
type s_items = set of items;
function closure(I : s_items) : s_items;
var
  temp, res : s_items;
begin
  res := I;
  repeat
    temp := res;
    for all (A → α.Bβ, a) ∈ temp do # '.' indicates position in production
      for all productions B → γ do # consider expansions of B
        for all terminals b ∈ FIRST(βa) do # what can follow B
          res := res + (B → γ, b); # possibly new items added to set
    until temp = res;
  return res;
end;
```

LR parsing table construction (2)

```pascal
var
  X_1(=S), X_2, ... X_n : non-terminals;
  X_{n+1}, X_{n+2}, ... X_m : terminals;
  temp, I_1, I_2, ... : s_items;
begin
  i := 0; p := 1; I_1 := closure({(S' → S,)}); # start here
  while i < p do # while sets to process
    begin
      i := i+1;
      for all (A → β, X_j) ∈ I_i where A → β is production k do
        if (A → β, X_j) is (S' → S, $) then
          action[i,j] := accept else
          action[i,j] := reduce k
      end
    end
end
```

LR parsing table construction (3)

```pascal
while i < p do # while sets to process
  for all terminal/non-terminal X_j do
    begin
      temp := closure({all (A → α.X_j β, a)}) where (A → α.X_j β, a) ∈ I_i
      if not empty(temp) then
        begin
          I_{p+1} := temp; # set up sentinel
          k := minimum r such that I_r = temp; # find set of items - or sentinel
          if k = p+1 then p := p+1; # increment p for new set of items
          if X_j is terminal then
            goto[i,j] := shift k else
            action[i,j] := k;
        end
    end
end
```

LR parsers

- The above techniques are generically referred to as LR(1) parser
LR parsers

- Specifically, there are differences in LR(1) parsers:
  - LR(1) parsers differentiate stack symbols by both the positions and the follower symbols
  - LALR(1) parsers merge stack symbols if they represent the same set of positions but different follower symbols (the sets of follower symbols are merged)
  - SLR(1) only differentiates sets of positions and determines follower symbols using the algorithms presented earlier in the course

LALR(1) parsers

- LALR(1) parsers merge stack symbols if they represent the same set of positions but different follower symbols.

In the expression parsing table, we merge the following symbols:
- 2 and 16, 3 and 17, 4 and 9, 5 and 10, 6 and 14, 7 and 20, 8 and 15, 11 and 18, 12 and 21, 13 and 19

This gives us the table overleaf which is identical to the one first considered

Example: Expressions

<table>
<thead>
<tr>
<th></th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0,0,5), (1,0,5)+, (2,0,5)+, (3,0,5)+<em>, (4,0,5)+</em>, (5,0,5)+<em>, (6,0,5)+</em></td>
<td>s5, s4</td>
</tr>
<tr>
<td>1</td>
<td>(0,1,5), (1,1,5)+</td>
<td>s6</td>
</tr>
<tr>
<td>2</td>
<td>(2,1,5)+, (3,1,5)+*</td>
<td>r2, s7, r2, r2</td>
</tr>
<tr>
<td>3</td>
<td>(4,1,5)+*</td>
<td>r4, r4, r4, r4</td>
</tr>
<tr>
<td>4</td>
<td>(5,1,5)+<em>, (1,0,5)+, (2,0,5)+, (3,0,5)+</em>, (4,0,5)+<em>, (5,0,5)+</em>, (6,0,5)+*</td>
<td>s5, s4</td>
</tr>
<tr>
<td>5</td>
<td>(6,1,5)+*</td>
<td>r6, r6, r6, r6</td>
</tr>
<tr>
<td>6</td>
<td>(1,2,5)+, (3,0,5)+<em>, (4,0,5)+</em>, (5,0,5)+<em>, (6,0,5)+</em></td>
<td>s5, s4</td>
</tr>
<tr>
<td>7</td>
<td>(3,2,5)+<em>, (5,0,5)+</em>, (6,0,5)+*</td>
<td>s5, s4</td>
</tr>
<tr>
<td>8</td>
<td>(5,2,5)+*, (1,1)+</td>
<td>s6, s11</td>
</tr>
<tr>
<td>9</td>
<td>(1,3,5)+, (3,1,5)+*</td>
<td>r1, s7, r1, r1</td>
</tr>
<tr>
<td>10</td>
<td>(3,3,5)+*</td>
<td>r3, r3, r3, r3</td>
</tr>
<tr>
<td>11</td>
<td>(5,3,5)+*</td>
<td>r5, r5, r5, r5</td>
</tr>
</tbody>
</table>

LR parsers

- LR(1) parsers are able to handle the largest class of grammars (without modification) but produce very large tables
- LALR(1) parsers are able to handle most common language grammars with a significant reduction in table size compared to LR(1)
- SLR(1) parsers have the smallest tables but may have problems with common language grammars
- Hence, LALR(1) is a common choice in parser-generators
Tricky grammars
- The following grammar is LALR(1) but not SLR(1)
  \[ S \rightarrow L = R \mid R \]  # assignment or expression
  \[ L \rightarrow * R \mid \text{id} \]  # indirection or identifier
  \[ R \rightarrow L \]  # R-value can be an L-value

- The following grammar is LR(1) but not LALR(1)
  \[ A \rightarrow aA \mid bB \mid aBe \mid bAe \]
  \[ A \rightarrow c \]
  \[ B \rightarrow c \]

Properties of LR parsing
- The parse time is linear relative to the length of the input
- Grammars generally don’t require modification, or at least only minor modification!!
- LR parsing has good error detection — no shift is performed if a symbol is incompatible with the grammar — erroneous input is reported as soon as possible
- Error recovery is not so simple — cf. LL parsing where the stack contains the unmatched part of the sentential form, and hence the context

Using an LR parser generator
- You define your grammar in the appropriate notation and feed it to the parser generator
- It may report “shift-reduce conflicts” — the parser has certain situations when it cannot decide between a shift and a reduce action — default response is to prefer a shift over a reduce
- It may report “reduce-reduce conflicts” — the parser has certain situations where it cannot decide which production to use for a reduction — commonly occurs when you have empty productions — need to resolve the problem, possibly by factoring the grammar
- Typically, you encounter a handful of such situations

Parseable languages
- We can compare the languages that can be parsed by various techniques:
Parseable languages

- The previous diagram is in terms of parseability of languages
- Just because you have a grammar which is not LR(1) doesn’t mean that you can’t find a grammar for the language which is LR(1)

- For all k, there are languages which are LL(k+1)-parseable but not LL(k) parseable
- For all k, if a language is LR(k)-parseable, then it is also LR(1)-parseable