COMP SCI 4TB3 / 6TB3 Compiler Construction

Emil Sekerinski
Department of Computing and Software
McMaster University

Emil Sekerinski 2016, revised March 2021

11. Generalized Parsing: Bottom-Up Parsing …

- Consider parsing the sentence x * (y + z) with the grammar:

  E → T I E "+" T.
  T → F I T "*" F.
  F → id I "(" E ")".

- We take two kinds of steps, S (shift) and R (reduce):

	stack	input
		x * (y + z)
S	x	* (y + z)
R	F	* (y + z)
R	T	* (y + z)
S	T *	(y + z)
S	T * (	y + z)
S	T * (y	+ z)
R	T * (F	+ z)
R	T * (T	+ z)
R	T * (E	+ z)

… Bottom-Up Parsing

S	T * (E +	z)
S	T * (E + z	)
R	T * (E + F	)
R	T * (E + T	)
R	T * (E	)
S	T * (E)	
R	T * F	
R	T	
R	E	

- Bottom-up parsing proceeds without a specific goal; the parse tree grows from bottom to top; the input is accepted if is reduced to the start symbol by two kinds to step:
  - Shift steps that shift the next input symbol on the stack.
  - Reduce steps that reduce a sequence of symbols on the stack according to a production.

LL(k) and LR(k) …

- In top-down parsing the leftmost symbol is expanded. In bottom-up parsing the rightmost symbol is reduced.

- In both methods the input is read from left to right. In both methods the decision for the next step is based on the stack and up to k symbols lookahead. The grammars suitable for op-down and bottom-up parsing methods are also known as:

  LL(k): left-to-right parse, leftmost derivation, k symbol lookahead
  LR(k): left-to-right parse, rightmost derivation, k symbol lookahead

  If k is omitted, it is assumed to be 1.

… LL(k) and LR(k)

- Consider following grammar for statements:

  S → I ":=" E I I "(" E ")".
  I → id ["." id].

  This grammar is not LL(1), LL(2), LL(3) but is LL(4) and LR(1). Consider now:

  S → I ":=" E I I "(" E ")".
  I → id { "." id }.

  This grammar is not LL(k) for any k, but is LR(1).

- The above grammar could be rewritten to be LL(1). In general this is not possible: More languages can be generated by LR(1) grammars than by LL(1) grammars.

LR Parsing – 1

- It is not possible to construct a program similar to recursive descent parsing. Rather, a deterministic finite state machine is applied to the input and the stack and determines the next step according to a transition table.

- The entries in the transition table are:

s n	shift into state n
r k	reduce by rule k
n	goto state n
a	accept
(blank)	error

- s n step: advance input one symbol; push n on stack
  r k step: assuming k: $A \rightarrow \chi$, pop $|\chi|$ symbols from stack,
  lookup A in goto with state on top of stack,
  push A and new state on stack.

LR Parsing – 2

Expression grammar:

(1) E → E + T

(2) E → T

(3) T → T * F

(4) T → F

(5) F → (E)

(6) F → id

The input is extended with an end-of-file symbol $.

state	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				a			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

LR Parsing – 3

- Consider parsing the sentence x * (y + z):

	stack	input
	${}_0$	x * (y + z)
s5	${}_0\ x_5$	* (y + z)
r6	${}_0\ F_3$	* (y + z)
r4	${}_0\ T_2$	* (y + z)
s7	${}_0\ T_2\ {*}_7$	(y + z)
s4	${}_0\ T_2\ {*}_7\ (_4$	y + z)
s5	${}_0\ T_2\ {*}_7\ (_4\ y_5$	+ z)
r6	${}_0\ T_2\ {*}_7\ (_4\ F_3$	+ z)
r4	${}_0\ T_2\ {*}_7\ (_4\ T_2$	+ z)
r2	${}_0\ T_2\ {*}_7\ (_4\ E_8$	+ z)

LR Parsing – 4

s6	${}_0\, T_2\ *_7\ (_4\ E_8\ +_6$	z)
s5	${}_0\, T_2\ *_7\ (_4\ E_8\ +_6\ z_5$	)
r6	${}_0\, T_2\ *_7\ (_4\ E_8\ +_6\ F_3$	)
r4	${}_0\, T_2\ *_7\ (_4\ E_8\ +_6\ T_9$	)
r1	${}_0\, T_2\ *_7\ (_4\ E_8$	)
s11	${}_0\, T_2\ *_7\ (_4\ E_8\ )_{11}$	
r5	${}_0\, T_2\ *_7\ F_{10}$	
r3	${}_0\, T_2$	
r2	${}_0\, E_1$	
a		

- The symbols on the stack are unnecessary, only the state, here written as an index, is relevant.

- The finite state machine only needs to inspect the top element of the stack as it contains all the information needed about the elements below.

LR Parsing – 5

- The key question is how to construct the transition table. Several ways of constructing it exist, each leading to a different set of languages accepted by the recognizer:
  - SLR (simple LR): easy to construct, but languages restrictive.
  - Canonical LR: difficult to construct, but least restrictive.
  - LALR (lookahead LR): in between above two, useful set of languages.

- In any case, the transition tables are too laborious to construct by hand; LR parsing can only be used with parser generators. Yacc and bison are commonly used LALR parser generators.

- Transition tables get large, compressing them is an issue.

LL vs. LR …

- Consider the leftmost and rightmost derivations of x * (y + z):

$$\begin{array}{lll@{\qquad\qquad}lll}
E & \Rightarrow & T & E & \Rightarrow & T \\
 & \Rightarrow & T * F & & \Rightarrow & T * F \\
 & \Rightarrow & F * F & & \Rightarrow & T * (E) \\
 & \Rightarrow & x * F & & \Rightarrow & T * (E + T) \\
 & \Rightarrow & x * (E) & & \Rightarrow & T * (E + F) \\
 & \Rightarrow & x * (E + T) & & \Rightarrow & T * (E + z) \\
 & \Rightarrow & x * (T + T) & & \Rightarrow & T * (T + z) \\
 & \Rightarrow & x * (F + T) & & \Rightarrow & T * (F + z) \\
 & \Rightarrow & x * (y + T) & & \Rightarrow & T * (y + z) \\
 & \Rightarrow & x * (y + F) & & \Rightarrow & F * (y + z) \\
 & \Rightarrow & x * (y + z) & & \Rightarrow & x * (y + z)
\end{array}$$

  We denote these by $\Rightarrow^L$ and $\Rightarrow^R$, respectively.

- LL parsing constructs the leftmost derivation, LR parsing constructs the rightmost derivation: the concatenation of stack and input is exactly the rightmost derivation in reverse order!

… LL vs. LR …

- Let $k{:}\omega$ be ω truncated to the first k symbols.

- G = (T, N, P, S) is LL(k) if for arbitrary derivations

$$\begin{tikzcd}[row sep=small]
 & & \mu\,\nu\,\chi \arrow[r, Rightarrow, "*"] & \mu\,\gamma \\
S \arrow[r, Rightarrow, "L"] & \mu\,A\,\chi \arrow[ur, Rightarrow] \arrow[dr, Rightarrow] & & \\
 & & \mu\,\nu'\,\chi \arrow[r, Rightarrow, "*"] & \mu\,\gamma'
\end{tikzcd}$$

$\mu, \gamma, \gamma' \in T^*, \nu, \nu', \chi \in V^*, A \in N$

$k{:}\gamma = k{:}\gamma'$ implies $\nu = \nu'$

- G = (T, N, P, S) is LR(k) if for arbitrary derivations

$$\begin{tikzcd}[row sep=small]
 & \mu\,A\,\omega \arrow[r, Rightarrow] & \mu\,\chi\,\omega \\
S \arrow[ur, Rightarrow, "R"] \arrow[dr, Rightarrow, "R"'] & & \\
 & \mu\,A'\,\omega' \arrow[r, Rightarrow] & \mu\,\chi\,\omega'
\end{tikzcd}$$

$\omega, \omega' \in T^*, \mu \in V^*$

$k{:}\omega = k{:}\omega'$ implies $A = A'$

… LL vs. LR

- For every LR(k) grammar there exists an equivalent LR(1) grammar.
- Every LL(k) grammar is also an LR(k) grammar
- There exists LR(k) grammars that are not LL(k'), for any k'.

- The LL and LR definitions involve all derivations of all sentences. They can equivalently be expressed in terms of the grammar.

- Theorem: Grammar G is LL(1) if and only if for any distinct productions $A \rightarrow \chi$ and $A \rightarrow \chi'$ we have first(χ follow (A)) $\cap$ first(χ' follow(A)) = $\emptyset$. (This is equivalent to Conditions 1 and 2.)

- For LR grammars no such condition exists; instead, the parser table is constructed and if that fails, the grammar is not LR (SLR, LALR).

SLR – 1

- An LR(0) item is a grammar rule with a dot on the right-hand side. For example, the rule E → E + T generates

  E → • E + T E → E • + T E → E + • T E → E + T •

  The rule A → ε generates only A → • .

- An LR(0) state is a set of (LR(0)) items. The items of a state indicate how much of the input has been recognized.

- We assume without loss of generality that each grammar contains a rule S' → S, where S' is the start symbol and S' does not occur anywhere else. For example, we add to the expression grammar:

  E' → E

SLR – 2

- If I contains $A \rightarrow \mu \cdot B\ \chi$, then closure(I) adds to I all items $B \rightarrow \cdot\ \nu$ for each rule for B, continuing recursively, i.e. adds all rules that may be needed in recognizing B. For example, for the expression grammar the closure of $\{E' \rightarrow \cdot\ E\}$ is:

  $E' \rightarrow \cdot\ E$ $\quad$ $E \rightarrow \cdot\ T$ $\quad$ $T \rightarrow \cdot\ F$ $\quad$ $F \rightarrow \cdot\ id$
  $E \rightarrow \cdot\ E + T$ $\quad$ $T \rightarrow \cdot\ T * F$ $\quad$ $F \rightarrow \cdot\ (E)$

- The result of goto(I, X) contains for every item $A \rightarrow \mu \cdot X\ \chi$ in I the closure of $A \rightarrow \mu\ X \cdot \chi$, i.e. the state after X is recognized, for $X \in V$. For example, if $I = \{E' \rightarrow \cdot\ E,\ E \rightarrow E \cdot + T\}$, then goto(I, +) is:

  $E \rightarrow E + \cdot\ T$ $\quad$ $T \rightarrow \cdot\ F$ $\quad$ $F \rightarrow \cdot\ id$
  $T \rightarrow \cdot\ T * F$ $\quad$ $F \rightarrow \cdot\ (E)$

SLR – 3

```
closure (I) =
    repeat
        for any item $A \to \mu \bullet B \chi$ in I
            add all $B \to \bullet \nu$ to I
    until I does not change
    return I
```

```
goto (I, X) =
    set J to the empty set
    for any item $A \to \mu \bullet X \chi$ in I
        add $A \to \mu X \bullet \chi$ to J
    return closure (J)
```

- The parsing tables action[i, a] and goto[i, A], with $a \in T$, $A \in N$ are obtained by taking state i to be I_i:

1. If $A \to \mu \bullet a \chi$ is in I_i and goto(I_i, a) = I_j, then action[i, a] := "s j"
2. If $A \to \omega \bullet$ is in I_i, then action[i, a] := "r $A \to \omega$ " for all $a \in$ follow(A)
3. If $S' \to S \bullet$ is in I_i, then action[i, \$] := "a"
4. If goto(I_i, A) = I_j, then goto[i, A] := j

- All entries not defined by rules 1 - 3 are made "error". The initial state is the one containing the item $S' \to \bullet S$.

SLR – 4

For the expression grammar we have following LR(0) sets …

I_0: $E' \rightarrow \bullet\, E \qquad T \rightarrow \bullet\, F$
$E \rightarrow \bullet\, E + T \qquad F \rightarrow \bullet\, (E)$
$E \rightarrow \bullet\, T \qquad F \rightarrow \bullet\, id$
$T \rightarrow \bullet\, T * F$

I_1: $E' \rightarrow E\, \bullet \qquad E \rightarrow E\, \bullet + T$

I_2: $E \rightarrow T\, \bullet \qquad T \rightarrow T\, \bullet * F$

I_3: $T \rightarrow F\, \bullet$

I_4: $F \rightarrow (\bullet\, E) \qquad T \rightarrow \bullet\, F$
$E \rightarrow \bullet\, E + T \qquad F \rightarrow \bullet\, (E)$
$E \rightarrow \bullet\, T \qquad F \rightarrow \bullet\, id$
$T \rightarrow \bullet\, T * F$

I_5: $F \rightarrow id\, \bullet$

I_6: $E \rightarrow E + \bullet\, T \qquad F \rightarrow \bullet\, (E)$
$T \rightarrow \bullet\, T * F \; F \rightarrow \bullet\, id$
$T \rightarrow \bullet\, F$

I_7: $T \rightarrow T * \bullet\, F \; F \rightarrow \bullet\, id$
$F \rightarrow \bullet\, (E)$

I_8: $F \rightarrow (E\, \bullet) \qquad E \rightarrow E\, \bullet + T$

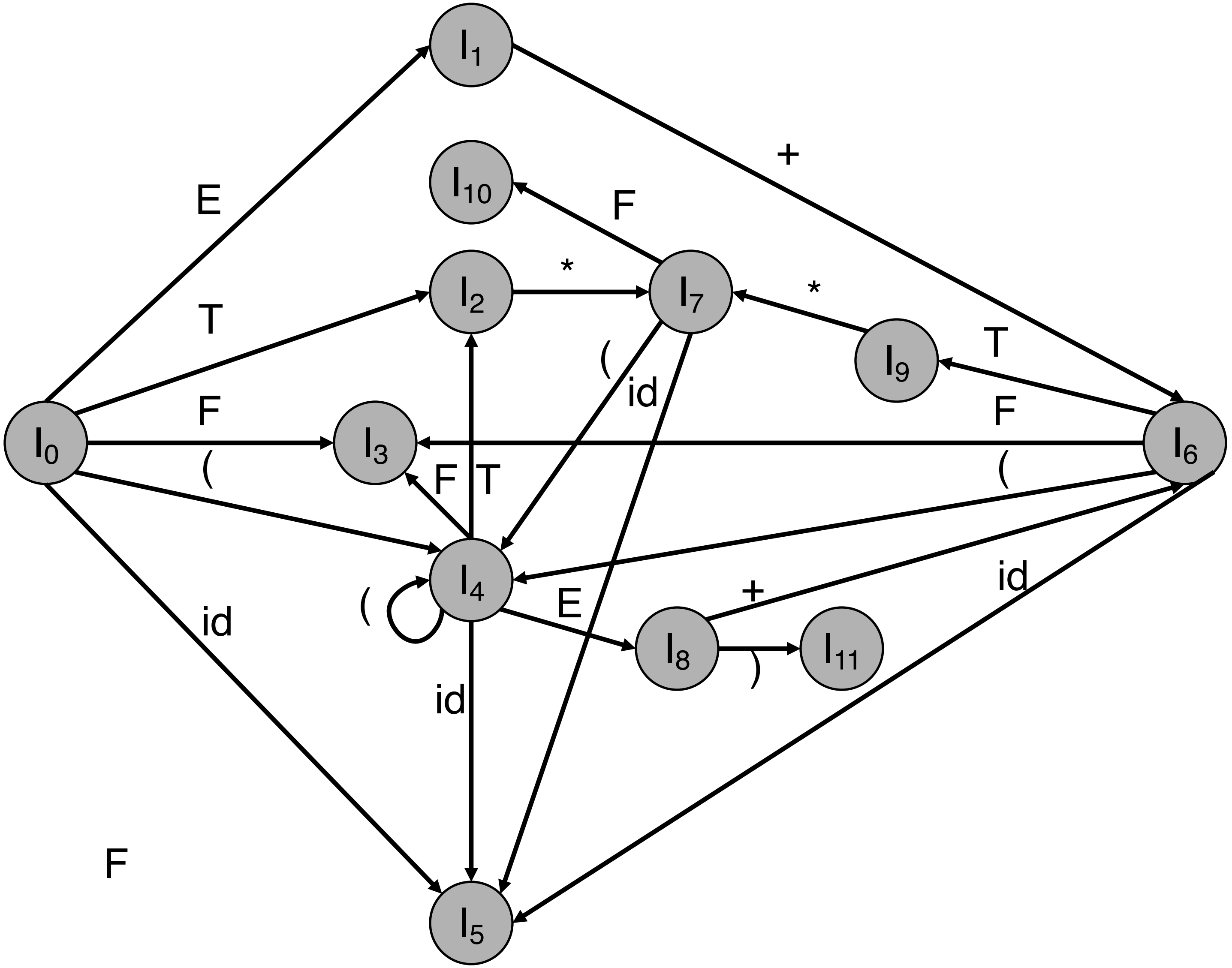
I_9: $E \rightarrow E + T\, \bullet \qquad T \rightarrow T\, \bullet * F$

I_{10}: $T \rightarrow T * F\, \bullet$

I_{11}: $F \rightarrow (E)\, \bullet$

SLR – 5

… and following transition diagram:



A grammar that is not SLR…

L and R stand for l-value and r-value, and * for "contents of":

(0) $S' \to S$
(1) $S \to L = R$
(2) $S \to R$
(3) $L \to * R$
(4) $L \to \mathrm{id}$
(5) $R \to L$

The grammar is unambiguous. The LR(0) items are:

I_0: $S' \to \cdot S$ $\quad L \to \cdot * R$
$S \to \cdot L = R$ $\quad L \to \cdot \mathrm{id}$
$S \to \cdot R$ $\quad R \to \cdot L$

I_1: $S' \to S \cdot$

I_2: $S \to L \cdot = R$ $\quad R \to L \cdot$

I_3: $S \to R \cdot$

I_4: $L \to * \cdot R$ $\quad L \to \cdot * R$
$R \to \cdot L$ $\quad L \to \cdot \mathrm{id}$

I_5: $L \to \mathrm{id} \cdot$

I_6: $S \to L = \cdot R$ $\quad L \to \cdot * R$
$R \to \cdot L$ $\quad L \to \cdot \mathrm{id}$

I_7: $L \to * R \cdot$

I_8: $R \to L \cdot$

I_9: $S \to L = R \cdot$

… A grammar that is not SLR

- Consider I_2:
  - $S \rightarrow L \bullet = R$ makes action[2,=] = "s 6"
  - $R \rightarrow L \bullet$ makes action[2,=] = "r $R \rightarrow L$", as = is in follow(R) (for example in * R = R)

- Hence this leads to a shift-reduce conflict; an SLR parser cannot be constructed.

- In above situation, a reduction would be incorrect as there is no sentence of the form R = … . The reason for failure is that not enough context is captured in the states.

- The solution is to split states by redefining items.

LR - 1

- An LR(1) item is a grammar rule with a dot on the right-hand side together with a lookahead symbol. An item ($A \to \sigma \bullet \tau$, a) indicates that σ is on top of the stack and the head of the input is derivable from τ a.

- An LR(1) state is a set of (LR(1)) items. The items of a state indicate how much of the input has been recognized.

- We assume without loss of generality that each grammar contains a rule $S' \to S$, where S' is the start symbol and S' does not occur anywhere else.

- The closure and goto functions are extended appropriately.

LR - 2

```
closure (I) =
    repeat
        for any $(A \to \mu \bullet B \chi, a)$ in I
            for any $b \in$ first($\chi$ a)
                add all $(B \to \bullet \nu, b)$ to I
    until I does not change
    return I
```

```
goto (I, X) =
    set J to the empty set
    for any $(A \to \mu \bullet X \chi, a)$ in I
        add $(A \to \mu X \bullet \chi, a)$ to J
    return closure (J)
```

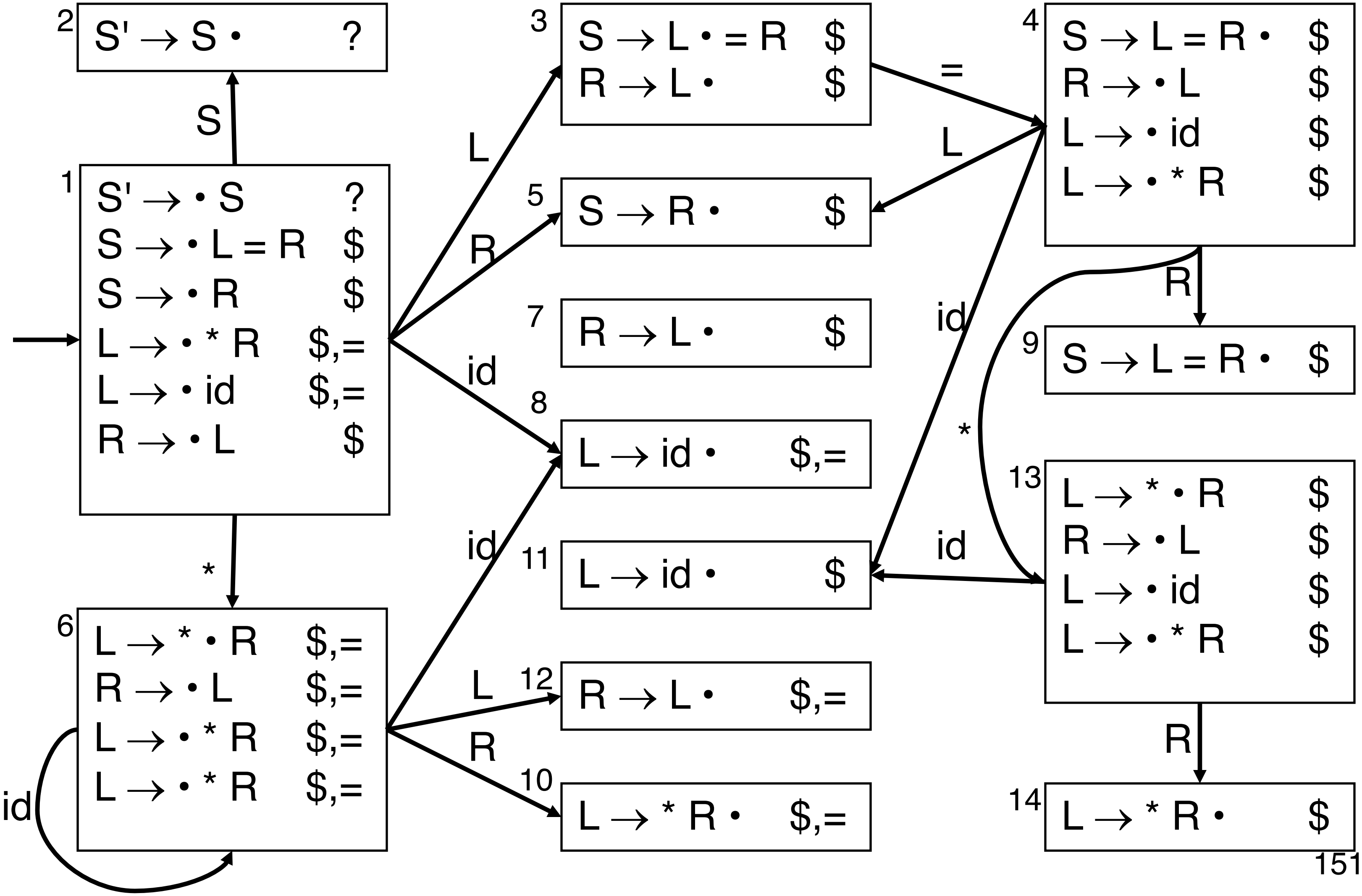
- The parsing tables action[i, a] and goto[i, A], with $a \in T$, $A \in N$ are obtained by taking state i to be I_i:

1. If $(A \to \mu \bullet a \chi, b)$ is in I_i and goto(I_i, a) = I_j, then action[i, a] := "s j"
2. If $(A \to \omega \bullet, a)$ is in I_i, then action[i, a] := "r $A \to \omega$ "
3. If $(S' \to S \bullet, a)$ is in I_i, then action[i, $] := "a"
4. If goto(I_i, A) = I_j, then goto[i, A] := j

- All entries not defined by rules 1 - 3 are made "error". The initial state is the one containing the item $S' \to \bullet S$.

LR - 3

LR(1) states for previous grammar, with items that differ only in their lookahead merged on one line …



1
S' → • S ?
S → • L = R $
S → • R $
L → • * R $,=
L → • id $,=
R → • L $

2
S' → S • ?

3
S → L • = R $
R → L • $

4
S → L = R • $
R → • L $
L → • id $
L → • * R $

5
S → R • $

6
L → * • R $,=
R → • L $,=
L → • * R $,=
L → • * R $,=

7
R → L • $

8
L → id • $,=

9
S → L = R • $

10
L → * R • $,=

11
L → id • $

12
R → L • $,=

13
L → * • R $
R → • L $
L → • id $
L → • * R $

14
L → * R • $

LR - 4

… the resulting LR(1) and LALR(1) parsing tables:

	id	*	=	$	S	R	L
1	s8	s6			2	5	3
2				a			
3			s4	r5			
4	s11	s13				9	7
5				r2			
6	s8	s6				10	12
7				r5			
8			r4	r4			
9				r1			
10			r3	r3			
11				r4			
12			r5	r5			
13	s11	s13				14	7
14				r3			

	id	*	=	$	S	R	L
1	s8	s6			2	5	3
2				a			
3			s4	r5			
4	s8	s6				9	7
5				r2			
6	s8	s6				10	7
7			r5	r5			
8			r4	r4			
9				r1			
10			r3	r3			

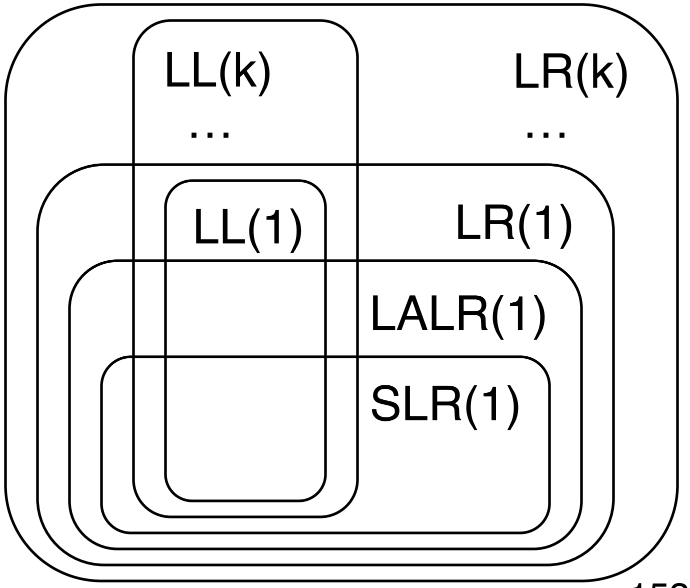
LR - 5

- The LALR table is constructed by merging states with identical items except for the lookaheads.

- In the sample grammar, states 6 and 13, states 7 and 12, states 8 and 11, states 10 and 14 can be merged.

- For a language like Pascal, doing so reduces the number of states from thousands to hundreds, and hence makes this practical.

- In some cases this may introduce a shift-reduce or a reduce-reduce conflict which is not present in the LR(1) table. Hence LALR(1) languages are a subset of LR(1) languages. For most practical purposes, this restriction is of no relevance.



… LL vs. LR …

- Let G_0 be:

  $S \rightarrow A \qquad A \rightarrow x\,A \qquad B \rightarrow x\,B$

  $S \rightarrow B \qquad A \rightarrow y \qquad B \rightarrow z$

  Then:

  $L(G_0) = \{x^i\,y \mid i \geq 0\} \cup \{x^i\,z \mid i \geq 0\}$

- G_0 is not LL(k) for any k. Consider derivations

$$\begin{tikzcd} & A \Rightarrow^* x^i\,y \\ S \arrow[ur, Rightarrow] \arrow[dr, Rightarrow] & \\ & B \Rightarrow^* x^i\,z \end{tikzcd}$$

$(\nu = A,\ \nu' = B,\ \mu = \chi = \varepsilon,\ \gamma = x^i\,y,\ \gamma' = x^i\,z)$

$k{:}x^i\,y = k{:}x^i\,z$ implies $A = B$ does not hold for $i \geq k$.

- G_0 is LR(1) and is SLR.

Construct an SLR parser for G_0!

… LL vs. LR …

- Let G_1 be:

  $S \rightarrow A\ y \qquad A \rightarrow A\ x \qquad B \rightarrow B\ x$

  $S \rightarrow B\ z \qquad A \rightarrow \varepsilon \qquad B \rightarrow \varepsilon$

  Then:

  $L(G_1) = L(G_0)$

Give the derivation of xxy with S and R steps!

- G_1 is not LR(k) for any k. Consider derivations

$$\begin{tikzcd} & A\ x^k\ y \arrow[r, Rightarrow] & x^k\ y \\ S \arrow[ur, Rightarrow, "R"] \arrow[dr, Rightarrow, "R"'] & & \\ & B\ x^k\ z \arrow[r, Rightarrow] & x^k\ z \end{tikzcd}$$

$(A' = B,\ \mu = \chi = \varepsilon,\ \omega = x^k\ y,\ \omega' = x^k\ z)$

$k{:}x^k\ y = k{:}x^k\ z$ implies A = B does not hold (reduce-reduce conflict).

- (Note that an equivalent LR(1) grammar, LL(1) grammar, and even regular grammar can be given).

- This issue occurs with any grammar that has productions with same right and sides such that if one is chosen, one can "get stuck" later on.

… LL vs. LR

- Let G_2 be:

  $S \rightarrow A\ B$ $A \rightarrow x\ y$ $C \rightarrow y\ y\ x$

  $S \rightarrow x\ C$ $B \rightarrow y\ y$

  Consider following two derivations:

			x y y x			x y y y
	S	x	y y x	S	x	y y y
	S	x y	y x	S	x y	y y
(*)	S	x y y	x	R	A	y y
	S	x y y x		S	A y	y
	R	x C		S	A y y	
	R	S		R	A B	
				R	S	

  G_2 is not LR(1) (shift-reduce conflict in (*)), but is LR(2).

- In general, for every context free grammar an equivalent LR(1) grammar exists. However, since the meaning is based on the parse tree, that would have to be redefined as well.

LALR

- Following grammar is LALR(1), but not SLR:
  S → A a | b A c | d c | b d a
  A → d

- Following grammar is LR(1), but not LALR(1):
  S → A a | b A c | B c | b B a
  A → d
  B → d

Functional Parsing - 1

- Functional language have recursion as the basic control structure. They allow particularly compact implementations of recursive descent parsing. We use the ocaml functional language.

- If T is the type of terminals, then a parser is a function of type

  T list -> T list

  that consumes a prefix of its input and returns the remainder.

- For each nonterminal B defined by B → E, we construct a function with the same name which parses those sentences which can be derived from E:

  B → E   let B = Pr(E)

  The whole parser becomes a set of recursive functions, with the function associated to the start symbol being the function which has to be called first.

Functional Parsing - 2

- The input to a parsing function is matched against the x::t (list with head x and tail t) to obtain the next input symbol x.

- The rules for constructing Pr(E) for recognizing expression E are:

E	Pr(E)
"a"	function "a" :: t -> t
B	B
(F)	Pr(F)
[F]	function s -> match s with first(F)::_ -> Pr(F) I any -> any
{F}	let rec h = function s -> match s with first(F)::_ -> h(Pr(F) s) I any -> any in h
F G	function s -> Pr(G) (Pr(F) s)

F I G	function s -> match s with
	    first(F)::_ -> Pr(F) s I
	    first(G)::_ -> Pr(G) s

Functional Parsing - 3

- The function construct combines definition and matching:

```
function p -> e l q -> f = function a -> match a with p -> e l q -> f
```

- For example consider the grammar:

  A → "a" A "c" l "b"

  Sentences can be recognized by following function:

```
let rec aa = function
    'a'::t -> (match aa t with 'c'::u -> u) l
    'b'::t -> t
;;
```

  For example:

```
#   aa ['a'; 'b'; 'c'];;
- : char list = []
# aa ['a'; 'c'];;
Exception: Match_failure ("", 155, 0).
```

Derive this parser!

Add more meaningful error messages!

Functional Parsing - 4

- The goal of parsing is not just to accept the input, but to construct its its (abstact) syntax tree. For the grammar

```
expression  → term { ("+" | "-") term }
term        → factor { ( "*" | "/") factor }
factor      → id | id "(" exprList ")" | "(" expression ")"
exprList→ expression { "," expression }
```

  where id is a lower case character, we define

```
type exp =
    | Sum of exp * exp
    | Diff of exp * exp
    | Prod of exp * exp
    | Quot of exp * exp
    | Id of char
    | Fun of char * exp list;;
```

- For example, the syntax tree of a * (b + c) is represented by:

```
Prod (Id 'a', Sum (Id 'b', Id 'c'))
```

Functional Parsing – 5

- If T is the type of terminals, then a parser is a function of type

  ```
  T list ->  P * T list
  ```

  that consumes a prefix of its input and returns the syntax tree of the consumed part (of type P) and the remainder of the input.

- The rules for constructing Pr(E) are extended accordingly:

- E F   Suppose the parse trees of E, F are of types e, f and the resulting parse tree of type f is to be constructed with function Cons: e * f -> g:

  ```
  functions s ->
      let (p, t) = Pr(E) s in
      let (q, u) = Pr(F) t
      in (Cons(p, q), u)
  ```

Functional Parsing - 6

- E {sep E} Suppose the parse tree of E is of type e and the tree for E sep … sep E is to be constructed as a flat list. The auxiliary function h is of type e * T list -> e list * T list:

```
functions s ->
    let rec h = function
        l (p, sep::t) -> let (q, u) = h (Pr(E) t) in (p::q, u)
        l (p, t) -> ([p], t)
    in h(Pr(E) s)
```

- E {lop E} Suppose the parse tree of E is of type e and the tree for E lop … lop E is to be constructed with Op: e * e -> e in a left associative way. The auxiliary function h is of type e * T list -> e * T list:

```
function s ->
    let rec h = function
        l (p, lop::t) -> let (q, u) = Pr(E) t in h(Op(p, q), u)
        l (p, t) -> (p, t)
    in h(Pr(E) s)
```

Functional Parsing - 7

- For the expression grammar:

```
let rec expression s =
  let rec moreterms = function
      | (p, '+'::t) -> let (q, u) = term t in moreterms (Sum (p, q), u)
      | (p, '-'::t) -> let (q, u) = term t in moreterms (Diff (p, q), u)
      | any -> any
  in moreterms (term s)

and term s =
  let rec morefactors = function
    | (p, '*'::t) -> let (q, u) = factor t in morefactors (Prod (p, q), u)
    | (p, '/'::t) -> let (q, u) = factor t in morefactors (Quot (p, q), u)
    | any -> any
  in morefactors (factor s)

and …
```

Functional Parsing – 8

```
and factor = function
  | '('::t ->
      (match expression t with
        | (p, ')'::u) -> (p, u)
        | _ -> raise (Failure "missing )"))
  | id::'('::t when id >= 'a' & id <= 'z' ->
      (match exprList t with
        | (p, ')'::u) -> (Fun(id, p), u)
        | _ -> raise (Failure "missing )"))
  | id::t when id >= 'a' & id <= 'z' -> (Id (id), t)
  | _::_ -> raise (Failure "id or ( expected")
  | [] -> raise (Failure "unexpected end")

and exprList s =
    match expression s with
      | (p, ','::t) -> let (q, n) = exprList t in (p::q, n)
      | (p, t) -> ([p], t)
;;
```

Functional Parsing – 9

- For example:

```
# expression ['a'; '*'; '('; 'b'; '+'; 'c'; ')'];;
- : exp * char list = (Prod (Id 'a', Sum (Id 'b', Id 'c')), [])
# expression ['a'; '+'; 'f'; '('; 'x'; ','; 'y'; ')'];;
- : exp * char list = (Sum (Id 'a', Fun ('f', [Id 'x'; Id 'y'])), [])
```

- Note that the grammar is not LL(1) because of

  factor → id I id "(" exprList ")" I …

  Here we use a lookahead of two symbols to resolve this conflict. The scheme can be generalized to a lookahead of arbitrarily many symbols!

Functional Parsing - 10

- For producing a meaningful error message, the current position has to be maintained. If T is the type of terminals, then a parser is a function of type

  ```
  position * T list -> position * T list
  ```

  that takes the input together with its starting position and returns a suffix together with its starting position.

- For separating scanning from parsing, we can scan the whole input first:

  ```
  scanner : char list -> symbol list
  parser : symbol list -> P * symbol list
  ```

  For the parser to report error positions, we add a position to each symbol:

  ```
  scanner : char list -> (position * symbol) list
  parser : (position * symbol) list -> P * (position * symbol list)
  ```

Functional Parsing - 11

- Alternatively we can scan "lazily" as needed. Scanning is then done by a function symbol that scans the first symbol of the input:

  ```
  symbol : T list -> symbol * T list
  parser : T list -> T list
  ```

  For reporting the error position, we extend:

  ```
  symbol : position * char list -> symbol * (position * char list)
  parser : position * char list -> position * char list
  ```

Combinator Parsing – 1

- Since parsing functions are just functions that can be passed as parameters, the idea is to construct new parsers using parsing combinator functions. The combinators are defined to reflect the structure of EBNF grammars.

- Function sym x tests whether the first symbol of the input is x; if so, it returns the remainder of the input, otherwise it raises an exception:

```
exception Noparse;;
let sym x = function
        | h::t when x = h -> t
        | _ -> raise Noparse;;
```

Combinator Parsing – 2

- The function seq pr1 pr2 s first applies parser pr1 to s and then parser pr2 to the remainder, and returns the remainder after applying both pr1 and pr2 to s. Its use is for parsing E F:

```
let seq pr1 pr2 s =
    pr2 (pr1 s);;
```

- The function option pr s tries to apply parser pr to s and returns the remainder in that case. If that fails, it returns the input instead. Its use it for parsing [E]:

```
let option pr s =
    try pr s
    with Noparse -> s;;
```

Combinator Parsing – 3

- The function repeat pr s keeps applying parser pr to s until it fails, and the returns the remainder of the input. Its use is for parsing {E}:

```
let rec repeat pr s =
    try repeat pr (pr s)
    with Noparse -> s;;
```

- The function choice pr1 pr2 s tries to apply parser pr1 to s. If that succeeds, the remainder of the input is returned, otherwise pr2 is applied to s. Its use is for parsing E | F:

```
let choice pr1 pr2 s =
    try pr1 s
    with Noparse -> pr2 s;;
```

Combinator Parsing – 4

- The parser for A → {a} b is:

```
let aa =  seq (repeat (sym 'a')) (sym 'b');;
```

- The parser for A → a A c | b is:

```
let rec aa s = choice (seq (sym 'a') (seq aa (sym 'c'))) (sym 'b') s;;
```

- The parser for S → A | B, A → x A | y, B → x B | z:

```
let rec ss s = choice aa bb s
and aa s = choice (seq (sym 'x') aa) (sym 'y') s
and bb s = choice (seq (sym 'x') bb) (sym 'z') s;;
```

- The last grammar is not LL(k), but can still be parsed: the choice combinator <u>backtracks</u> if one alternative fails. Ambiguity is resolved by giving preference to the first alternative. While backtracking may lead to an exponential complexity, grammars that are not LL(k) can be parsed. Left recursion is still not allowed.

Context-free Parsing – 1

- Earley's parser works with an arbitrary context-free grammar without backtracking. If the grammar is unambiguous, it produces a parse tree in quadratic time; if the grammar is ambiguous, it produces all parse trees in cubic time (in the length of the input). For most LR(k) grammars, it produces a parse tree in linear time.

- An Earley item is a pair with a grammar rule with a dot on the right-hand side and an index into the input. An item $(A \rightarrow \sigma \bullet \tau, i)$ means that we started recognizing A at input position i and have recognized σ so far.

- An Earley state is a set of (Earley) items.

- We assume without loss of generality that each grammar contains a rule $S' \rightarrow S$, where S' is the start symbol and S' does not occur anywhere else.

Context-free Parsing – 2

- Let the input be given by x(1), …, x(n) and assume x(n+1) = \$. Let s(0), …, s(n) be the Earley sets.

```
s(0) := {(S' → • S, 0)} ; for i = 1 to n do s(i) := {} ;
for i = 0 to n do
      repeat
            for any e ∈ s(i) choose:
            if e = (A → σ • a τ, j) and a = x(i + 1) then      -- scanner (S)
                  add (A → σ a • τ, j) to s(i + 1)
            if e = (A → σ • B τ, j) then                       -- predictor (P)
                  for all rules B → μ do
                        add (B → • μ, i) to s(i)
            if e = (A → σ •, j) then                           -- completer (C)
                  for all items (B → μ • A χ, k) ∈ s(j) do
                        add (B → μ A • χ, k) to s(i)
      until s(i) does not change
accept := (S' → S •, 0) ∈ s(n)
```

Context-free Parsing – 3

- Consider the grammar:

  E → T | E + T T → F | T * F F → a

  The input a + a * a is accepted as (S → E •, 0) ∈ s(5):

	item			index	
s(0):	S	→	• E	0	
(x(1) = a)	E	→	• T	0	P
	E	→	• E + T	0	P
	T	→	• F	0	P
	T	→	• T * F	0	P
	F	→	• a	0	P
s(1):	**F**	**→**	**a •**	**0**	**S (at 0)**
(x(2) = +)	**T**	**→**	**F •**	**0**	**C**
	E	**→**	**T •**	**0**	**C**
	T	→	T • * F	0	C
	S	→	E •	0	C
	E	→	E • + T	0	C

Context-free Parsing – 4

s(2):	E	→	E + • T	0	S (at 1)	
(x(3) = a)	T	→	• T * F	2	P	
	T	→	• F	2	P	
	F	→	• a	2	P	
s(3):	**F**	**→**	**a •**	**2**	**S (at 2)**	
(x(4) = *)	**T**	**→**	**F •**	**2**	**C**	
	E	→	E + T •	0	C	
	T	→	T • * F	2	C	
	S	→	E •	0	C	
	E	→	E • + T	0	C	
s(4):	T	→	T * • F	2	S (at 2)	
(x(5) = a)	F	→	• a	4	P	
s(5):	**F**	**→**	**a •**	**4**	**S (at 3)**	
(x(6) = $)	**T**	**→**	**T * F •**	**2**	**C**	
	E	**→**	**E + T •**	**0**	**C**	items in bold
	T	→	T • * F	2	C	correspond to
	S	**→**	**E •**	**0**	**C**	the derivation
	E	→	E • + T	0	C	

Context-free Parsing – 5

- Earley's parser is a top-down parser as it starts with S.

- The number of items in s(i) is proportional to i (worst case). Scanner and predictor need at most i steps for s(i), but the completer may need i^2 steps, as adding an item may cause a previous set to be revisited. Summing i^2 for i = 0 to n gives n^3 steps.

- The states can be implemented as lists with a marker of which items have been visited and which still need to be visited.

- The items can be implemented representing the rules as linked lists and having pointers into these lists. Alternatively the items can be treated as LR(0) items; the members of the LR(0) states can be precomputed (prediction is similar to closure), arriving at an implementation similar to SLR parsing! Such a parser is about 3 times slower than an LALR parser; combined with the scanner, it is only twice as slower.

Context-free Parsing – 6

- An application is allowing programmers to extend the syntax of programming languages or mathematical systems, for which one should not impose restrictions on the kind of extension. This allows domain-specific syntax. (Some theorem provers allow this already by using Earley. Languages like C#, Java, C have certain syntactic translations built in, e.g. for for-loops.)

- Other context-free parsers exist: the Cocke-Younger-Kasami algorithm constructs a Boolean array P of $n \times n \times k$ values, where k is the size of the vocabulary: P(i, j, k) is true if and only if x[i .. j] can be derived from symbol k. The array is filled "bottom-up". The algorithm always requires n^3 time and n^2 space (and the grammar has to be in a certain normal form), so is not practical, but used to argue about the complexity of context-free parsing.

- Generalized LR (GLR) parsing allows shift-reduce and reduce-reduce conflicts by keeping a "cactus stack".