Concepts of Programming Languages

9th Week

LR-Parsing

Prof. Dr. Peter Heusch

Overview

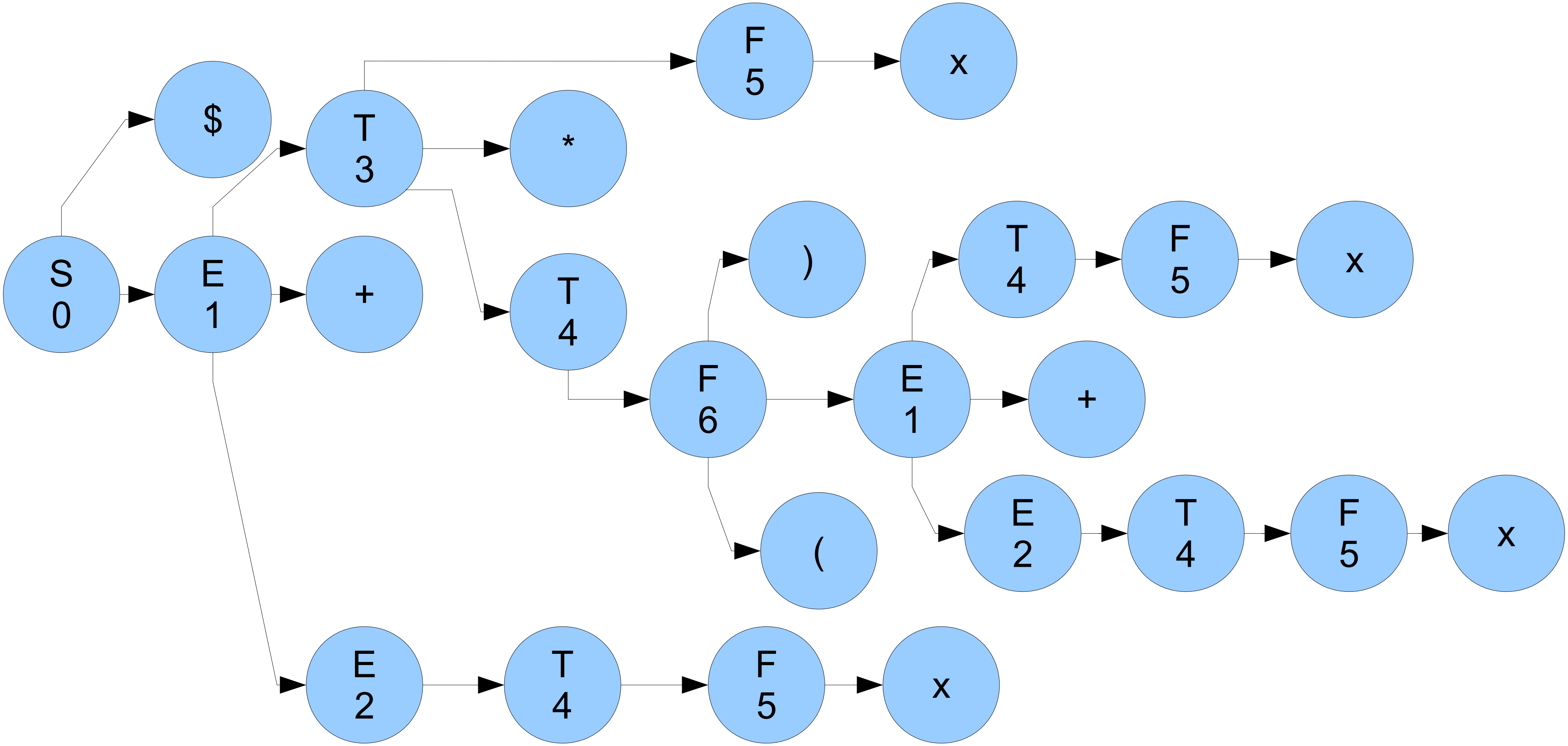
- LL-Parsing suffers from serious drawbacks, it is neccessary to adapt the grammar
- The main problem is the need to choose the correct rule by only looking at the first k tokens of the expansion
- It would be better to choose the correct rule after having seen the complete expansion
- This is accomplished by using a different parser strategy, LR-parsing (and some of its simplifications, SLR and LALR)

The Sample Grammar

- Unless otherwise stated, our sample grammar is as follows:
  - S –› E '$' 0
  - E –› E '+' T | T 1 2
  - T –› T '*' F | F 3 4
  - F –› 'x' | '(' E ')' 5 6
- The token '$' is assumed to be the end-of-input sign
- Furthermore we assume, that the parser will apply E –› T only, if E –› E '+' T is not applicable

Derivation Tree

- The ultimate result of a parser run is a derivation tree
- For the input x + (x + x) * x the tree looks as follows:



The Earley Parsing Algorithm

- We start with an parser algorithm suitable for every context-free grammar (even ambiguous)
- This algorithm uses sets Q_j of configuration entries having the form E –› E '+' • T
- If set Q_3 contains the above entry, after reading the first symbol, a derivation might have continued by trying to expand an E to symbols 2 and 3 and potentially more symbols derivable from T

Computing the Configuration Sets

- Three operations are needed to compute the configuration sets
- Prediction: If A–›… • B …, i is in Q_j, then for every rule B –› α add a new entries B –› • α, j to Q_j
- Completion: If A–›… •, i is in Q_j, then move every entry X –› … • A …, k from Q_i to Q_j while shifting the dot: X –› … A • …, k
- Scanning: If a configuration X –› … • b …, k where the • directly precedes terminal j+1 is contained in Q_j, move this configuration to Q_j+1 while shifting the dot: X –› … b • …, k

The algorithm

- If S is the start symbol of the grammar, add the configuration –› • S, 0 to Q_0
- Compute sets Q_i for all i from 0 to number of terminals as follows:
  - Use prediction and completion on Q_i until Q_i is stable
  - Use scanning to move from Q_i to Q_i+1
- The input is derivable from the grammar if and only if –› S •, 0 is contained in the last set Q_k

A Simple Example

- We use the standard grammar to derive x+x+x:

  Q_0: –›•S,0 S–›•E,0 E–›•E+T,0 E–›•T,0 T–›•T*F,0 T–›•F,0 F–›•(E),0

  F–›•x, 0

  Q_1: F–›x•,0 T–›F•,0 E–›T•,0 T–›T•*F,0 S–›E•,0 E–›E•+T,0 –›S•, 0

  Q_2: E–›E+•T,0 T–›•T*F,2 T–›•F,2 F–›•(E),2 F–›•x, 2

  Q_3: F–›x•,2 T–›F•,2 E–›E+T•,0 T–›T•*F,2 S–›E•,0 E–›E•+T,0 –›S•, 0

  Q_4: E–›E+•T,0 T–›•T*F,4 T–›•F,4 F–›•(E),4 F–›•x, 4

  Q_5: F–›x•,4 T–›F•,4 E–›E+T•,0 T–›T•*F,4 S–›E•,0 E–›E•+T,0 –›S•, 0

A More Complex Example

- We use the standard grammar to derive the word x+(x+x)*x: –›
  Q_0: –›•S, 0 S–›•E, 0 E–›•E+T, 0 E–›•T, 0 T–›•T*F, 0 T–›•F, 0 F–›•(E), 0 F–›•X, 0
  Q_1: F–›X•, 0 T–›F•, 0 E–›T•, 0 T–›T•*F, 0 S–›E•, 0 E–›E•+T, 0 –›S•, 0
  Q_2: E–›E+•T, 0 T–›•T*F, 2 T–›•F, 2 F–›•(E), 2 F–›•X, 2
  Q_3: F–›(•E), 2 E–›•E+T, 3 E–›•T, 3 T–›•T*F, 3 T–›•F, 3 F–›•(E), 3 F–›•X, 3
  Q_4: F–›X•, 3 T–›F•, 3 E–›T•, 3 T–›T•*F, 3 F–›(E•), 2 E–›E•+T, 3
  Q_5: E–›E+•T, 3 T–›•T*F, 5 T–›•F, 5 F–›•(E), 5 F–›•X, 5
  Q_6: F–›X•, 5 T–›F•, 5 E–›E+T•, 3 T–›T•*F, 5 F–›(E•), 2 E–›E•+T, 3
  Q_7: F–›(E)•, 2 T–›F•, 2 E–›E+T•, 0 T–›T•*F, 2 S–›E•, 0 E–›E•+T, 0 –›S•, 0
  Q_8: T–›T*•F, 2 F–›•(E), 8 F–›•X, 8
  Q_9: F–›X•, 8 T–›T*F•, 2 E–›E+T•, 0 T–›T•*F, 2 S–›E•, 0 E–›E•+T, 0 –› S•, 0

Basics of Shift / Reduce Parser

- For technical reasons, an end-of-input symbol \$ is added to the terminals and another new start symbol S‘ with a rule S‘ –› S \$ is added
- Construct the action tables and the goto tables
- While input is neither refuted nor accepted:

  if action table for state / symbol is shift x:
    shift symbol, goto state x
  if action table for state / symbol is reduce y:
    take twice the number of symbols on the right side of rule y from stack, push left side of this rule y onto the stack, compute new state from two topmost symbols on stack, push this state onto stack

Determining the Correct Rule

- In order to determine a correct rule for the derivation, the following items are usable:
  - Symbols that can start whatever is derived from a variable V (FIRST)
  - Symbols that can follow whatever is derived from a variable V (FOLLOW)
  - Symbols that can follow whatever is derived from a variable V after V was found on the right side of a rule (LOOKAHEAD)
- In theory, it is possible to look after arbitrary long sequences, but most parsers use only 0 or 1 symbol

FIRST Sets

- For every nonterminal V, the set FIRST_k(V) denotes the set of all sequences of k tokens that can possibly start a word derived from V
- If we only look at FIRST(V) = FIRST_1(V), the computation is easy:

  For any terminal x set FIRST(x) = { x }

  If V –› … –› ε is possible, add ε to FIRST(V)

  For any rule V –› x y z, add FIRST(x) to FIRST(V), if ε is in FIRST(x), then add FIRST(y) to FIRST(V), if ε is also in FIRST(y), etc.
- FIRST is easily extended to symbol sequences

FOLLOW Sets

- For every nonterminal V, the set FOLLOW_k(V) denotes the set of all sequences of k tokens that can possibly follow a word derived from V
- If we only look at FOLLOW(V) = FOLLOW_1(V), the computation is easy:
  - Add $ to FOLLOW(S) ($=end of-file, S start symbol)
  - For any production (V –›   B), add FIRST( ) \ { } to FOLLOW(B)
  - For productions (A –›   B) or (A –›   B) mit   –› .. –› , add FOLLOW(A) to FOLLOW(B)

LOOKAHEAD Sets

- LOOKAHEAD sets describe a refinement of FOLLOW sets
- While FOLLOW sets contain all terminals that can follow a certain variable, LOOKAHEAD sets consider how the variable was created:
  $S \rightarrow aTa \mid bTb \mid aUb \mid bUa$, $T \rightarrow c$, $U \rightarrow c$
- Both T and U have FOLLOW sets {a, b}, but if T was created by $S \rightarrow aTa$, it has LOOKAHEAD {a}, otherwise has LOOKAHEAD {b}

LOOKAHEAD Sets

- LOOKAHEAD sets describe a refinement of FOLLOW sets
- While FOLLOW sets contain all terminals that can follow a certain variable, LOOKAHEAD sets consider how the variable was created:
  S –› aTa | bTb | aUb | bUa, T –› c, U–›c
- Both T and U have FOLLOW sets {a, b}, but if T was created by S–›aTa, it has LOOKAHEAD {a}, otherwise has LOOKAHEAD {b}

Variants of Shift / Reduce Parsers

- There are multiple variants of shift reduce parsers that differ w.r.t their power:
  - LR(1), the most general one, but very large because LOOKAHEAD is used to control the reduce process
  - LALR(1) is somewhat easier because it compresses the LOOKAHEAD data
  - SLR(1) is the easiest technique because it only uses FOLLOW sets instead of LOOKAHEAD sets
  - LR(0) does not consider FOLLOW or LOOKAHEAD
- We will look at LR(0) and SLR(1) in detail

From Earley to LR

- To construct action tables, we observe that during the Earley algorithm, some situations repeat constantly:
  - When the • precedes an E, the configurations E–› • E+T,.. E–› •T,.. T–› •T*F,.. T–› •F,.. F–› •(E),.. F–› •x,.. are generated
  - When the configuration F –› x•,.. was scanned, completion generates the configurations T–›F•,.. T–› T•*F,.. E–› E+T•,.. E–› T•,.. and E–› E•+T,..
- The central observation is that there are only a finite sets of different configuration sets
- So the recognizer works with a DFA and a stack

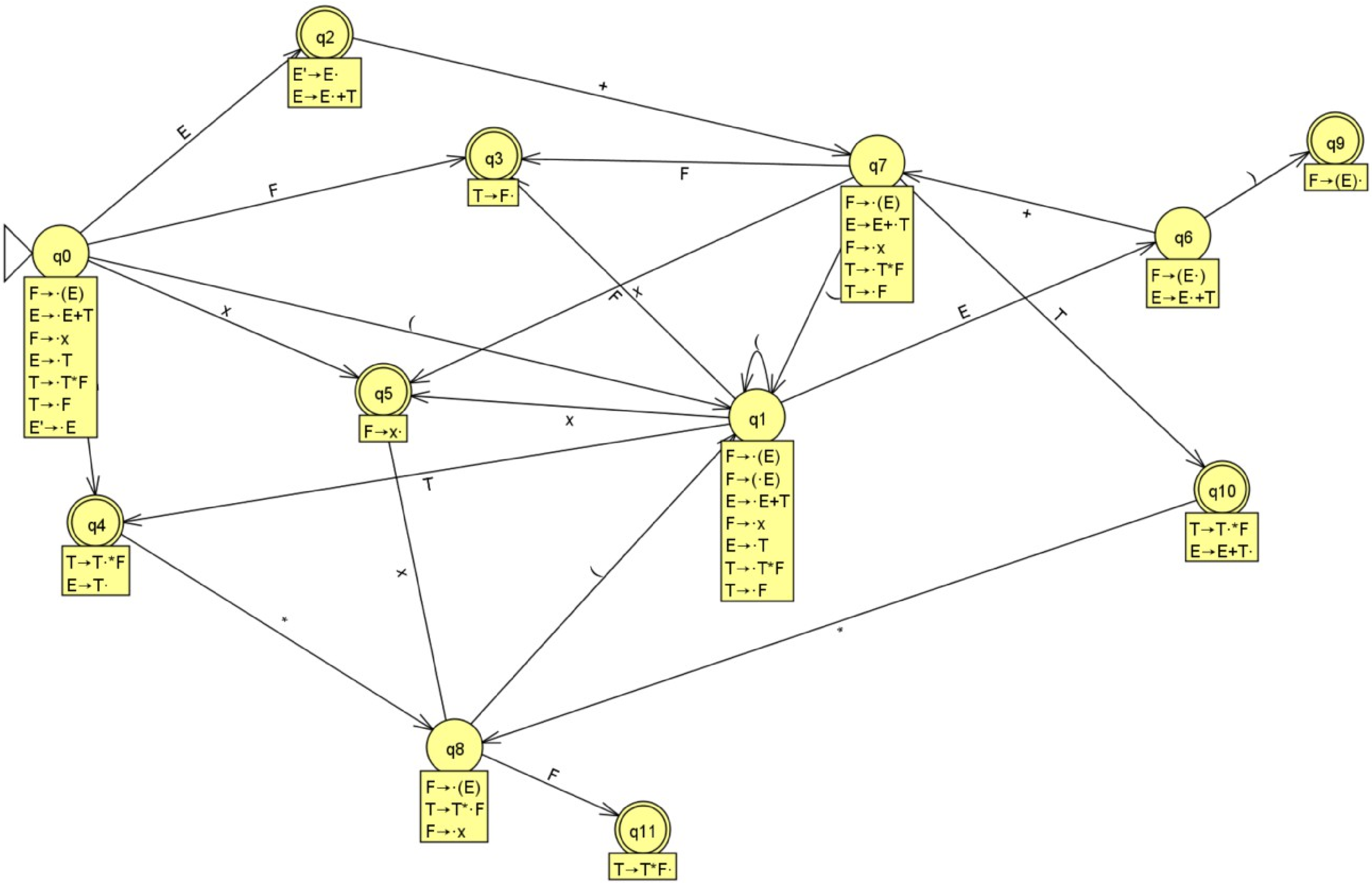
Constructing the States

- To construct the states, we use an algorithm that is comparable to Earley prediction:
  - Start with the production S –› •E $
  - Predict like in Earley by adding all productions for E with a • at the left margin, if now the • is to the left of some other variable, proceed recursively
  - All these configurations form state 0
- To form the other configuration sets, we add new states corresponding to configuration sets created by reading a variable or a terminal
- Proceed until no new states can be added

DFA for the Example Grammar

- Start with S–›•E$, predict E–›•E+T, E–›•T, T–›•T*F, T–›•F,0 F–›•(E), and F–›•x, giving q_0
- From q_0 and "(" create F–›(•E), predict E–›•E+T, E–›•T, T–›•T*F, T–›•F,0 F–›•(E), and F–›•x, giving q_1
- From q_0 and "E", create S–›E•$ and E–›E•+T, giving q_2
- From q_0 and "F", create T–›F•, giving q_3
- From q_0 and "T", create E–›T• and T–›T•*F, giving q_4
- From q_0 and "x", create F–›x•, giving q_5
- From q_1 and "E", create F–›(E•) and E–›E•+T, giving q_6
- Create all other states, if necessary remove copies, e.g. all states lead to q_5 with "x"
- The resulting DFA has 11 states

The DFA


q0
F→·(E)
E→·E+T
F→·x
E→·T
T→·T*F
T→·F
E'→·E
q1
F→·(E)
F→(·E)
E→·E+T
F→·x
E→·T
T→·T*F
T→·F
q2
E'→E·
E→E·+T
q3
T→F·
q4
T→T·*F
E→T·
q5
F→x·
q6
F→(E·)
E→E·+T
q7
F→·(E)
E→E+·T
F→·x
T→·T*F
T→·F
q8
F→·(E)
T→T*·F
F→·x
q9
F→(E)·
q10
T→T·*F
E→E+T·
q11
T→T*F·

Bulding the action tables

- To construct the parser one must build two types of tables:
  - The action table tells us what to do if some symbol is found in a certain state (shift or reduce?)
  - The goto table tells us where to go if a reduction has taken place
  - Both can be combined (see below)
- The construction of the action table depends on the type of parser being contstructed:
  - LR(0): Action depends on state only
  - SLR(1): Action depends on state and next symbol

Build the Action & Goto Table

- Build a Table with n (state count) x m (element count) elements, first the terminals, then the nonterminals
- If in some state the • is at the end of a rule, reduce using this rule
- If in some state the • is before a terminal, shift the terminal, goto the next state
- If in some state the • is before a non-terminal goto the next state

The LR(0) table

State	(	)	*	+	x	$	E	F	T
0	s 1				s 5		2	3	4
1	s 1				s 5		6	3	4
2				s 7		acc			
3	r 4	r 4	r 4	r 4	r 4	r 4			
4	r 2	r 2	r 2 / s 8	r 2	r 2	r 2			
5	r 6	r 6	r 6	r 6	r 6	r 6			
6		s 9		s 7					
7	s 1				s 5			3	10
8	s 1				s 5			11	
9	r 5	r 5	r 5	r 5	r 5	r 5			
10	r 1	r 1	r 1 / s 8	r 1	r 1	r 1			
11	r 3	r 3	r 3	r 3	r 3	r 3			

Problems

- As one can see, there is a problem with the LR(0)-table, since in states 4 and 10 there is a shift/reduce conflict if the next token is *
- To resolve this conflict, one can take into account the FOLLOW-sets of the left side of rule 2 (which causes the conflict)
- Rule 2 is E –› T, FOLLOW(E) is { +,), $ } , hence in this case (*) the reduction makes no sense
- By taking into account this information the number of shift/reduce conflicts decreases

Modification for SLR(1) Action Table

- If in some state the • is at the end of a rule, reduce using this rule, but only if the next token (lookahead) is in the FOLLOW set of the variable on the left side of the rule
- This also decreases the number of wrong error messages, because the error condition is detected before the reduction rather than after

The SLR(1) table

State	(	)	*	+	x	$	E	F	T
0	s 1				s 5		2	3	4
1	s 1				s 5		6	3	4
2				s 7		acc			
3		r 4	r 4	r 4		r 4			
4		r 2	s 8	r 2		r 2			
5		r 5	r 5	r 5		r 5			
6		s 9		s 7					
7	s 1				s 5			3	10
8	s 1				s 5			11	
9		r 6	r 6	r 6		r 6			
10		r 1	s 8	r 1		r 1			
11		r 3	r 3	r 3		r 3			

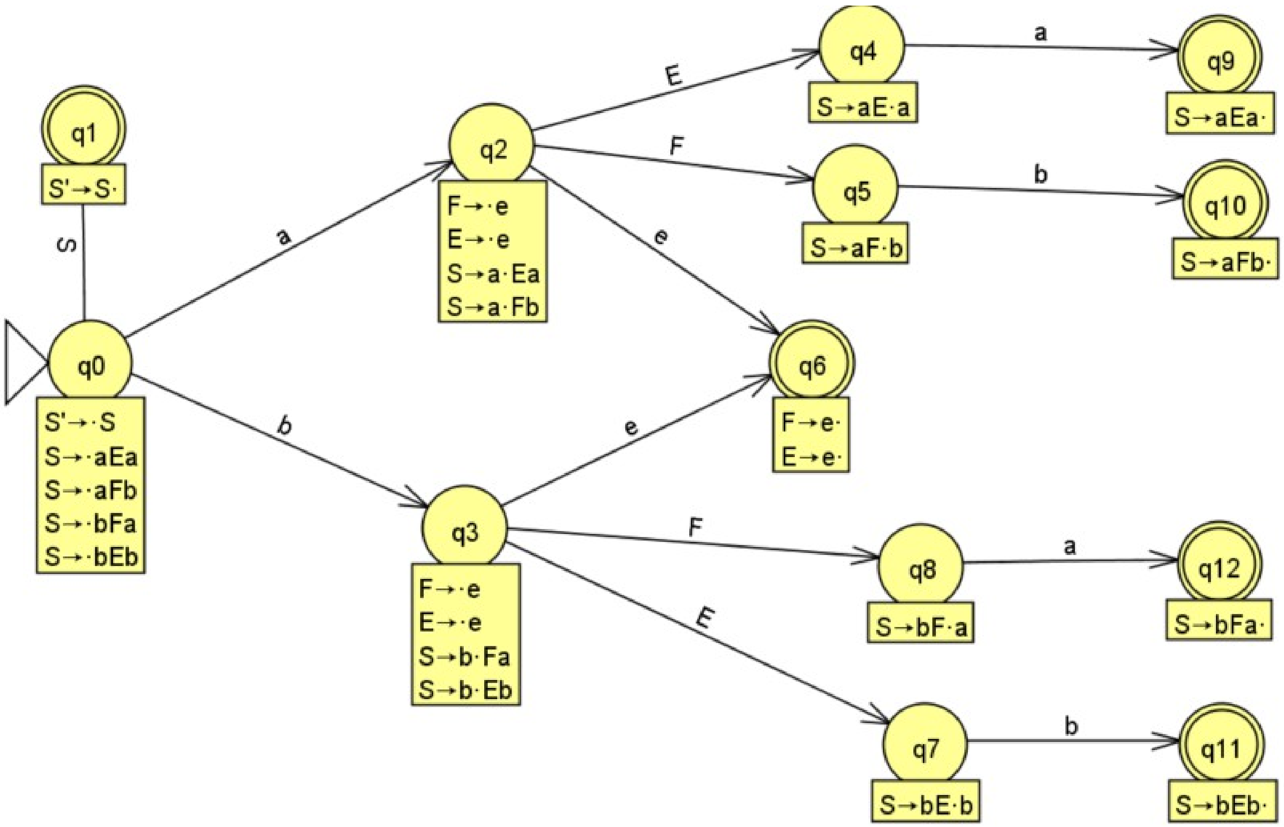
Limits of SLR Parsing

- SLR parsing can reduce the number of conflicts but it can not reduce all conflicts
- A simple example is:
  S –› aEa | bEb | aFb | bFa
  E –› e
  F –› e
- The problem will be visible on the next slide: after an a is shifted, the parser expects either an E or an F, but both variables have the same FOLLOW-set

Limits of SLR Parsing

- SLR parsing can reduce the number of conflicts but it can not reduce all conflicts
- A simple example is:
  S –› aEa | bEb | aFb | bFa
  E –› e
  F –› e
- The problem will be visible on the next slide: after an a is shifted, the parser expects either an E or an F, but both variables have the same FOLLOW-set

The DFA



q0
S'→·S
S→·aEa
S→·aFb
S→·bFa
S→·bEb
q1
S'→S·
q2
F→·e
E→·e
S→a·Ea
S→a·Fb
q3
F→·e
E→·e
S→b·Fa
S→b·Eb
q4
S→aE·a
q5
S→aF·b
q6
F→e·
E→e·
q7
S→bE·b
q8
S→bF·a
q9
S→aEa·
q10
S→aFb·
q11
S→bEb·
q12
S→bFa·
S
a
b
E
F
e
e
F
E
a
b
a
b

Extending LR Parsing

- The canonical LR-Parser further subdivides the lookahead token by checking the context where a rule was initialized:
  - If an E was introduced by rule 0, it's followed by a $
  - If an E was introduced by rule 1, it's followed by a +
  - If an E was introduced by rule 6, it's followed by a)
- However, these parsers tend to be very large
- A somewhat compacted form of an LR parser is the LALR-parser which unifies states if they only differ in the LOOKAHEAD-set, and which is the basis of most modern compilers

Conclusion

- Shift / Reduce parsing can parse all modern programming languages
- There are a variety of Shift / Reduce parser variants that only differ w.r.t. the selection of Shift / Reduce actions
- Most modern compilers use either a SLR or an LALR parser
- Nevertheless, it is almost impossible to write a parser without a parser generator

Exercise

- Take the sample grammar on slide 3 and the SLR(1)-table from slide 26 and evaluate how the grammar rules are applied to parse **x+(x+x)*x**:
  - If you run a LL-parse with my newly patented ILL© (infinite LL) parser
  - If you run a LR-parse
- Take the following grammar and construct the LR(0) automaton. Does it contain conflicts?
  - S'–› S
  - S –› [B
  - A –› int | [B
  - B –›] | C
  - C –› A] | A , C