Bottom-Up Parsing

- Recall that top-down parsing reflected a leftmost derivation of the grammar
- Bottom-Up Parsing reflects a rightmost derivation
  - The parse tree produced is the result of a rightmost derivation
  - The order of the reductions performed reflects this

- Top-down parsers have to make production decisions based on a small lookahead
- In contrast, bottom-up parsers push incoming tokens onto a stack and only make reduction decisions after they gain enough information
  - As a result, they are more powerful (can parse tougher grammars

Bottom-Up Parsing

- Compare the order the parse trees are created for leftmost versus rightmost derivations
  - For top-down parsers, productions are done corresponding to a leftmost derivation
  - For bottom-up parsers, reductions are done in reverse of the rightmost derivation
- Example: a + b + c * d

```
expr -> term expr2
expr2 -> addop term expr2 | ε
term -> factor  term2
term2 -> mulop factor term2 | ε
factor -> ( expr ) | IDENT | NUM
 addop -> + | -
 mulop ->  * | /
```

```
expr -> expr addop term  | term
term -> term mulop factor | factor
factor -> ( expr ) | IDENT | NUM
 addop -> + | -
 mulop ->  * | /
```

Bottom-Up Parsing

- Like an LL(1) parser, a bottom-up parser uses a stack
  - Holds both terminals and non-terminals
  - Also holds state information (like a DFA)
- The stack starts out empty, and ends up after a successful parse with just the Start symbol; i.e., the code reduces to just the Start symbol
- The parser uses the state information, plus possibly the next token(s) in the incoming string, to make one of 4 decisions
  1. Accept – just the Start symbol remains, and the code is correct
  2. Shift – move a token from the incoming string onto the stack
  3. Reduce – take a set of symbols from the stack (a production RHS) and reduce it to a single non-terminal (the LHS)
     - Symbols on stack replaced by non-terminal
  4. Error
- Bottom-up parsers sometimes called shift-reduce parsers

Bottom-Up Parsing

- The grammar used by a bottom-up parser is always augmented to contain a new Start symbol
  - New production from new Start Symbol to old start symbol
  - Prevents us from ever having a start symbol on stack until the incoming token stream is empty

- Common bottom-up parsers include LR(0) and LR(1)
  - Left-to-right scan, rightmost derivation, 0 or 1 character lookahead
  - Because a character which has already been shifted on the stack is not considered lookahead, it is feasible to build a bottom-up parser which does not need a lookahead symbol
    - Uses just the state information
    - Not as powerful as LR(1), and can't handle some language constructs well

Bottom-Up Parsing

- Example: matching parens
  - Note new start symbol
  - Note this is rightmost derivation

S' -> S
S -> (S) S | ε

Stack	Input	Action
	()$	shift
(	)$	reduce S-> ε
(S	)$	shift
(S)	$	reduce S-> ε
(S)S	$	reduce S->(S)S
S	$	reduce S' -> S
S'	$	accept

Bottom-Up Parsing

- Example: rudimentary expressions

S' -> E
E -> E + n | n

Stack	Input	Action
	n+n$	shift
n	+n$	reduce E -> n
E	+n$	shift
E+	n$	shift
E+n	$	reduce E -> E + n
E	$	reduce S' -> E
S'	$	accept

Note: 3rd and 6th line identical except for lookahead – This grammar is not an LR(0) grammar, since it cannot be parsed without lookahead

Bottom-Up Parsing

- The derivation used on previous slide was

  S' => E => E + n => n + n
- Each set of terminals/non-terminals formed during the rightmost derivation is a right sentential form
  - Part of this right sentential form will be on stack, part in string
  - The set of symbols on stack is known as a viable prefix of the right sentential form
  - A shift-reduce parser will continue to shift tokens until the entire RHS of a production is on the stack
    - At this point, a reduction may change the present right sentential form to another one
      - The complete viable prefix, plus the production which can be used to reduce it, is known as the handle of the right sentential form
    - Determining when a **handle** exists on the stack (and thus recognizing if it is time to make a reduction) is the primary task of a parser

Bottom-Up Parsing

- In our example of a simple expression grammar, we had E on top of the stack, which was the rhs of S' -> E
  - But we chose to shift instead of reduce at that point
  - So, just because the symbols on top of the stack happen to match a production rhs doesn't mean that we have a handle on the stack and it is time for a reduction
    - The lookahead symbols seem to be important
    - We will see that the state we are in is also important
- In the matching parens example, we had S-> ε as a production
  - Certainly ε is always on the top of the stack, but we don't always choose this reduction

Bottom-Up Parsing

- We said that a right sentential form can be split between the stack and the input string
  - Consider the production E -> E + n
  - There are 4 possible ways this could be split
    - Nothing on stack, just E on stack, etc.
  - We call each of these possibilities an LR(0) item or just item
    - We show the dividing point (stack vs string) with a period (.) – a metasymbol
    - The items for this production are:

      E -> . E + n

      E -> E . + n

      E -> E + . n

      E -> E + n .

Bottom-Up Parsing

- For the grammar we saw before, the items are:

S’ -> . S

S’ -> S .

S -> . (S) S

S -> (. S) S

S -> (S .) S

S -> (S) . S

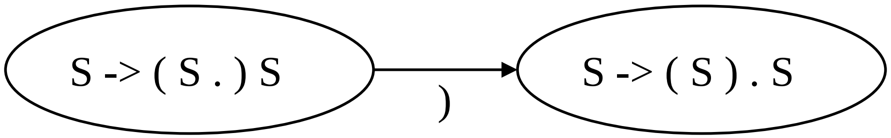
S -> (S) S .

S -> .

S’ -> S
S -> (S) S | ε

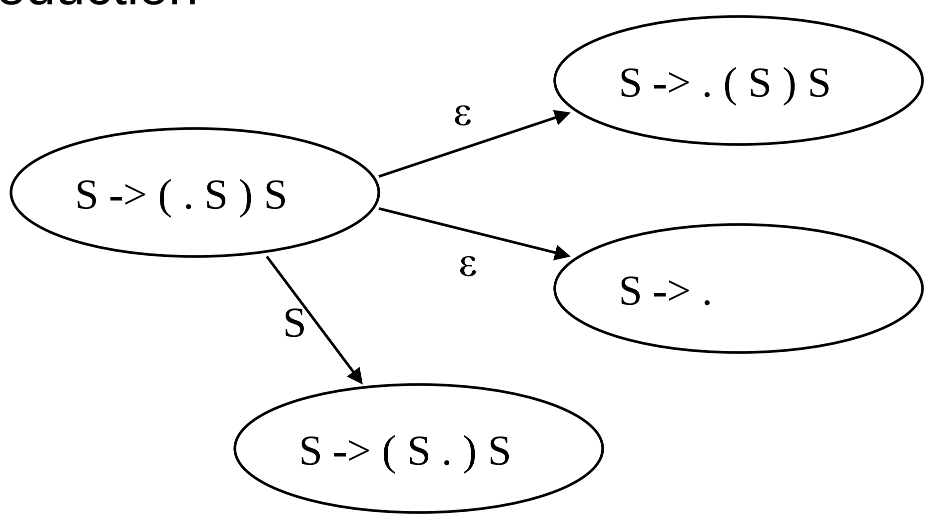
Bottom-Up Parsing

- Each of the items possible for a grammar could be thought of as a state the parser is currently in
  - In fact, this is what we do
    - The items are states of a finite automata, specifically an NFA
  - If I am in state S -> (S .) S, and I decide to shift, bringing a) from the input string, I transition to S -> (S) . S
  - It's a bit more complicated when the symbol following the period is a non-terminal, as in S -> (. S) S
    - There aren't any S tokens in the incoming string
    - The only way we're going to get an S on stack is as the result of a reduction



Bottom-Up Parsing

- So, in the case of S -> (. S) S, we will have to go do some reduction to S, and then we can move to S -> (S .) S
  - We show this in NFA by showing that state S -> (. S) S makes e-transition to states from which we can build an S
    - States with S on lhs, and period as first symbol on rhs
  - We can then show the transition on S in the NFA, realizing we can only make this transition immediately following a reduction

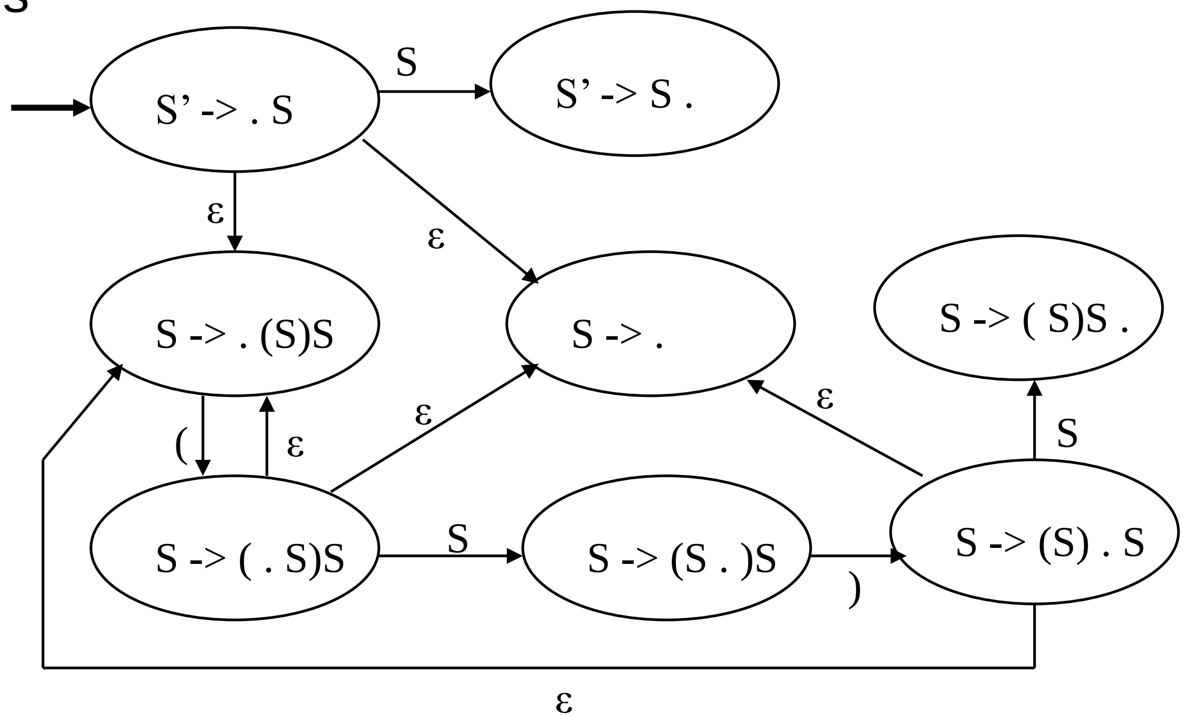


Bottom-Up Parsing

- Here is NFA for the grammar:
  - No accepting state
    - Purpose isn't to accept string, but to define state transitions

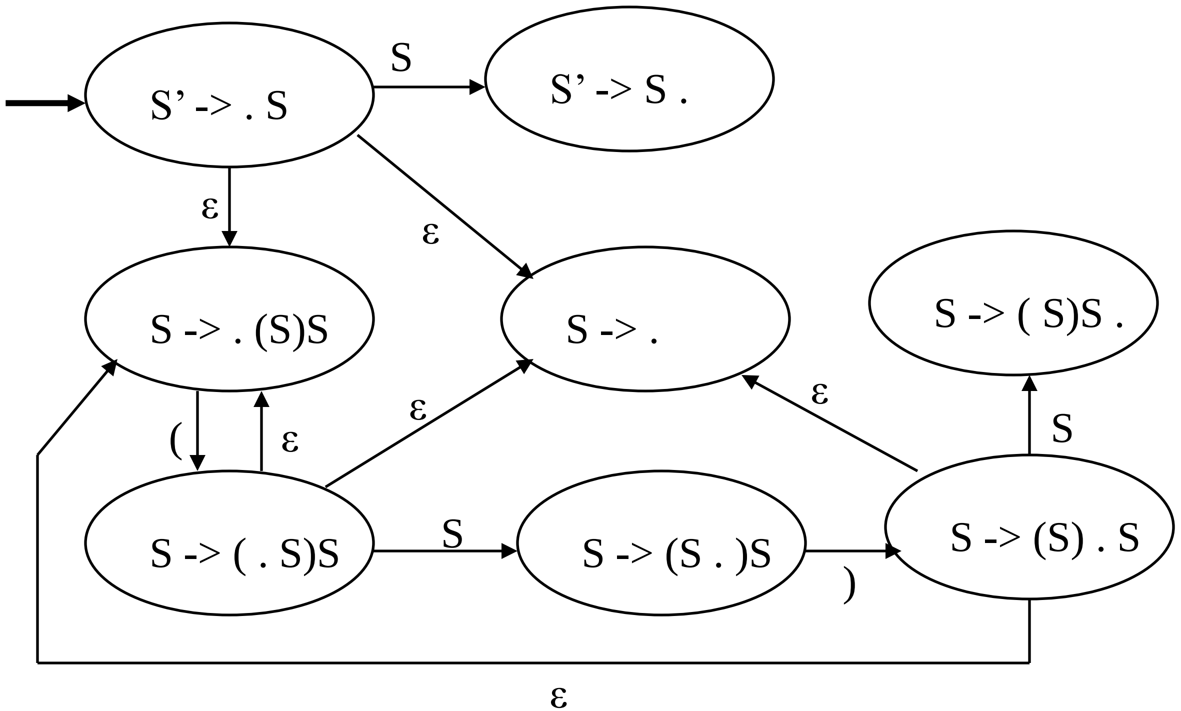
S' -> S
S -> (S) S | ε

S' -> . S
S' -> S .
S -> . (S) S
S -> (. S) S
S -> (S .) S
S -> (S) . S
S -> (S) S .
S -> .



Bottom-Up Parsing

- We can now construct a DFA, using subset construction
  - Just as before, the DFA states consist of sets of states of the NFA; i.e., sets of items
  - What would the DFA look like for this NFA?



Bottom-Up Parsing

- For the following grammar:
  - Construct list of items
  - Construct the NFA
  - Construct the DFA

```
S' -> E
E -> E + n | n
```

LR(0) Parsing

- The basic LR(0) parsing algorithm uses a stack, an input string of tokens, and a transition table (formed from DFA)
  - The stack contains not just symbols, but also current state
    - We show it as alternating symbols and states
    - In reality, the state captures the symbol information and only the states have to be stored
  - Input string terminated by $
  - Transition table shows states vs symbols
    - Typically, columns in table containing non-terminals are in separate section of table called goto section
    - Columns containing terminals reflect action on a shift
    - Columns in goto section reflect transition following a reduction

LR(0) Parsing

- Basic LR(0) algorithm
  - If your current state contains a production like A -> B . b C, such that the symbol after the period is a terminal, then the required action is to shift
    - Shift next character onto stack
      - This character must be b, or an error has occurred
    - Change to state containing A -> B b . C
  - If your current state contains A -> B . such that the period is after the last symbol (called a complete item), then you should do a reduction by the rule
    - Entire rhs of production must be on the stack
    - Remove all of rhs from stack (including associated states), push A onto the stack
      - If A is Start symbol, then accept (if input empty)
    - The current stack state must contain D -> E . A F
      - Transition to state containing D -> E A . F

LR(0) Parsing

- Basic LR(0) algorithm (cont)
  - If A -> B . b C, shift
  - If A -> B . , reduce
  - If neither, then your parser isn't working correctly

- If the above rules can be executed unambiguously for a particular grammar, the grammar is said to be an LR(0) grammar
  - If a particular state contains both a shift form and a reduce form, then the parser has a shift-reduce conflict cause by an ambiguity in the grammar
  - If a particular state contains 2 different complete items, then a reduce-reduce conflict has occurred due to ambiguity

- Thus, a grammar is LR(0) iff every state either contains only shift items, or contains a single complete item

LR(0) Parsing

- Look at DFA on page 205
  - Is the grammar represented an LR(0) grammar?

- Look at DFA on page 206
  - Is this an LR(0) grammar?

LR(0) Parsing

- Example: grammar A -> (A) | a
  - What are the items?
  - What does the NFA look like?
  - We can build DFA directly from items, without needing NFA
    - This is what most parser generators do
    - State 0 includes A' -> . A
      - Any time a period comes before non-terminal, all items which are productions on that terminal (with period before all symbols) are added to the state
      - So, state 0 also includes A -> . (A) and A -> . a
    - State including A -> (. A) also must contain above 2
    - Each of other items must also be in the DFA
    - We will look at an algorithm for building DFA without even having to enumerate items following this example

LR(0) Parsing

- Example (cont): grammar A -> (A) | a
  – Given DFA, what does the parsing table look like?
  – What does a parse of the string ((a)) look like?

State	Action	Rule	Input			GoTo
			(	a	)	A
0	Shift		3	2		1
1	Accept	A' -> A				
2	Reduce	A -> a				
3	Shift		3	2		4
4	Shift				5	
5	Reduce	A -> (A)				

Sets of Items Construction

- Next we will look at the technique for determining the DFA states directly from the grammar, and thus building the parse table directly
  - Called sets of items construction, which is logical name

- Before we look at algorithm, need to define 2 functions:
  - closure (I) – (where I is a set of items) – very similar to ε-closure; closure (I) includes:
    - All items in I
    - If a item in I has period before non-terminal A, include all items of form A -> . B
      - Apply this recursively to B if it is a non-terminal

Sets of Items Construction

- Another definition – goto (I, X) where I is a set of items and X is a grammar symbol (either terminal or non-terminal)
  - If I contains A -> B . a C , then goto (I,a) contains closure(A -> B a . C)
  - Thus, we move period past the symbol, and take closure

- Set of Items Construction
  - Create start state containing closure (S' -> . S)
  - For each grammar symbol X, if goto (start state, X) is non-empty and is not identical to an existing state, add a new state containing goto (start state, X)
  - Repeat above for all newly created states

Sets of Items Construction

- Example: matching parens

S' -> S
S -> (S) S | ε

- Start state contains closure (S'-> . S)

S' ->. S
S -> . (S)S
S -> .

- Only goto which makes sense are on S and '('
  - goto (s_0, '(')

S' -> . S
S -> . (S)S
S -> .
s_0

$\xrightarrow{(}$

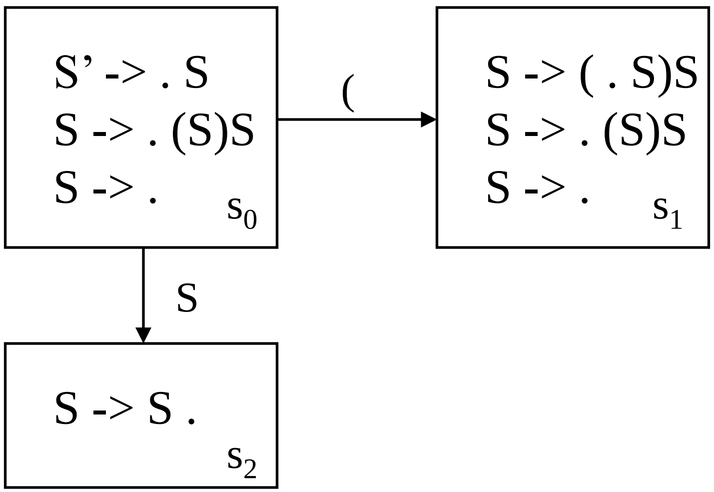
S -> (. S)S
S -> . (S)S
S -> .
s_1

Sets of Items Construction

- Example (cont)

S' -> S
S -> (S) S | ε

- Goto (s_0, S) is:



- Done with s_0, work on s_1 … continue iterating until no more new states created

Sets of Items Construction

- OK, a marathon example
  - Find DFA

```
E' -> E
E -> E + T | T
T -> T * F | F
F -> (E) | ID
```

  - I'll get you started
    - Start state is closure (E' -> . E)

```
E' -> . E
E -> . E + T
E -> . T
T -> . T * F
T -> . F
F -> . (E)
F -> . ID
```
s_0

- Can we create an LR(0) parse table from this grammar?

SLR(1) Parsing

- We saw that there were some very trivial grammars we couldn't parse using the LR(0) technique
  - Hey, bottom-up was supposed to be a more powerful technique
- Well, there are some obvious things we could have done to improve the technique
  - We made the decision on whether or not to shift based solely on what state we are in
    - We have tokens from the input string handy, but just ignored them
  - If we decided to reduce, we tried to make the decision based solely on what state we are in
    - Only allowed one complete item per state
    - Lookahead might help distinguish which reduction is appropriate

SLR(1) Parsing

- The Simple LR(1) Parsing or SLR(1) Parsing method uses lookahead to eliminate some of the shift-reduce and reduce-reduce conflicts
  - SLR(1) Parsing uses a more powerful parse table
    - Associates shift/reduce decisions with lookahead token

State	Action	Rule	Input			GoTo
			(	a	)	A
0	Shift		3	2		1
1	Reduce	A’ -> A				
2	Reduce	A -> a				
3	Shift		3	2		4
4	Shift				5	
5	Reduce	A -> (A)				

State	Input				GoTo
	(	a	)	$	A
0	s3	s2			1
1				acc	
2			A -> a	A -> a	
3	s3	s2			4
4			s5		
5			A ->(A)	A->(A)	

SLR(1) Parsing

- SLR(1) Algorithm
  – If your current state contains a production like A -> B . b C, and the nextToken is b, then the required action is to shift and go to state containing A -> B b . C
  – If your current state contains A -> B . and the nextToken is in Follow (A), then reduce by this production (popping rhs of production off stack) and Goto appropriate state for lhs
    - The current stack state must contain D -> E . A F
    - Transition to state containing D -> E A . F
  – If the two above rules can be followed unambiguously, I.e., no shift-reduce or reduce-reduce conflicts, then the grammar is an SLR(1) grammar
- Note that from same state you could either shift or reduce, based on nextToken
- Note that from same state you could do 2 different reductions, based on nextToken

SLR(1) Parsing

- Other than parse table changes, everything else stays same
- Example: matching parens
  - Not LR(0) grammar
  - Look at pg 205

S’ -> S
S -> (S) S | ε

State	Input			GoTo
	(	)	$	S
0	s2	S -> ε	S -> ε	1
1			accept	
2	s2	S-> ε	S-> ε	3
3		s4		
4	s2	S-> ε	S-> ε	5
5		S -> (S)S	S -> (S)S	

SLR(1) Parsing

- Example: rudimentary expressions
  - Not LR(0)
  - See pg 206
  - Follow (E) = ?
  - Parse n + n + n

E’ -> E
E -> E + n | n

State	Input			GoTo
	n	+	$	E
0				
1				
2				
3				
4				

SLR(1) Parsing

- SLR(1) parsing is powerful, but not perfect
  - Still has problems with shift-reduce and reduce-reduce conflicts
    - Typically most parser generators will default to performing the shift rather than the reduce
      - Fixes dangling else
    - Reduce-reduce occur infrequently in programming languages, and can probably be avoided
- An example of a problem grammar
  - Follow set of both S and V contains $

```
S' -> S
S -> id | V := E
V -> id
E -> V | n
```

Grammar

```
S' -> .S                 S -> id .
S -> .id       --id-->   V -> id .
S -> .V := E
V -> .id
```

Start state – only partial DFA shown

LR(1) Parsing

- How could SLR(1) Parsing be improved?
  - Once it has a parse table, it used lookahead as effectively as possible
  - However, it doesn't really use lookahead while building the parse table
    - Essentially it uses an LR(0) DFA, and applies lookahead
- LR(1) parsing builds a more advanced DFA,and then uses the same parse table and lookahead techniques as SLR(1) to conduct the parse
  - LR(1) DFA states (and thus parse table) are built keeping track of legal lookahead chars
    - If I am parsing if (expr) stmt else stmt, and am starting to parse expr, I know that the follow-on token when the expr is done has to be a)
      - Maybe I can benefit from this knowledge

LR(1) Parsing

- Consider the problem grammar below
  - Both the follow set of V and S contain $, but we can see that the V→. id item was only added to the Start State because of the S->. V := E item
    - Thus, in this state, the follow-on character to V can only be :=, not $
    - So, when I hit state where it's time to make a reduction, I could reduce S→id. if lookahead is $ and reduce V→id. if the lookahead is :=
    - But SLR(1) doesn't track this info, LR(1) does

```
S' -> S
S -> id | V := E
V -> id
E -> V | n
```

Grammar

```
S' -> .S                    S -> id .
S -> .id        --id-->     V -> id .
S -> .V := E
V -> .id
```

Start state – only partial DFA shown

LR(1) Parsing

- Essentially what LR(1) Parsing does is eliminate some of the reduce-reduce conflicts which we would find in an SLR(1) parser
  – In practice, almost all language constructs can be expressed by an LR(1) grammar
  – You can certainly design a non-ambiguous language which would not be LR(1), but it would likely be contrived

- Approach
  – We define a new type of item, an LR(1) item
    - Consists of an LR(0) item and a lookahead char/set
  – Build a DFA, or do Sets of LR(1) Items Construction, using slightly modified rules
  – From this DFA or Sets of Items, we can build the parse table
    - Format is the same as SLR(1), just has more states
  – Use parse table as we did in SLR(1)

LR(1) Parsing

- Building Sets of Items
  - Start with S' → . S, $
  - Take closure similar to before
    - For every item of form S→B.CD, x, we create a closure containing all productions on C, with each terminal in First(D) - if First(D) contains ϵ, then add x as lookahead
      - If C->E|F and First (D) = { +, (}, then we add following LR(1) items
        - C→.E, +
        - C → .E, (
        - C → .F, +
        - C → .F, (
  - Compute goto and shift same as before
    - S → B.CD, x transitions to S → BC.D, x

LR(1) Parsing

- Below is the non-SLR(1) grammar

```
S' -> S
S -> id | V := E
V -> id
E -> V | n
```

Grammar

```
S' -> .S                  id      S -> id .
S -> .id                 ----->   V -> id .
S -> .V := E
V -> .id
```

Old start state – only partial DFA shown

- With LR(1) Sets of Items
  - Now I can properly choose between reductions

```
S' -> .S, $               id      S -> id ., $
S -> .id, $              ----->   V -> id ., :=
S -> .V := E, $
V -> .id, :=
```

LR(1) start state – only partial DFA shown

LR(1) Parsing

- Again, if we can build a non-ambiguous parse table from our Sets of Items, we say that the grammar was an LR(1) grammar

- So, LR(1) is the most powerful parser of the 6 we will study
  - However, it has a problem … state explosion
  - Remember, we have to have epsilon-transitions (or closure) for every terminal character in the First set of the next symbol
    - In a full-size programming language, this could be a large set of characters and a huge number of states
  - Not practical to build a compiler this way

- LALR(1) Parsing (the last one, I promise) solves this problem
  - This is the one YACC and most other parser generators use

LALR(1) Parsing

- We are only going to take a brief look at LALR(1)
  - More powerful than SLR(1)
  - Almost as powerful as LR(1)

- Key concept is to look for multiple states that are identical in their LR(0) core, differing only in the lookahead portion of their items
  - If two states have same core, their nature is such that they must transition to states which also have same core
- If we combine states with the same core, and have a set of lookahead chars, we dramatically reduce # of states
  - In fact, will be identical to # of LR(0) states
- In practice, rarely lose any of the power of LR(1)
- Compare Fig 5-7 and Fig 5-9 (pgs 220, 225)

YACC

- Next we will look briefly at YACC, as an example of a parser generator or compiler-compiler
  - Stands for "Yet Another Compiler-Compiler"
  - Due to time, we will only look quickly at YACC

- YACC is a LALR(1) parser generator, created for the Unix environment
  - There are numerous implementations of YACC, and those included with the latest GNU release are called Bison

- Similar to Lex, you specify your grammar in an input file in YACC language, and then run YACC
  - It creates a .c file which can be compiled into your compiler
  - Typically called y.tab.c or ytab.c

YACC

- Typically, a YACC source file uses a .y suffix
- It is divided into 3 sections, with a %% (like Lex) dividing the sections
  - 1st section is used for preprocessor directives and definitions
    - % token LPAREN_TOKEN 258 defines a token used later and assigns a numeric value
    - % start stmt says that the start symbol for grammar is stmt – defaults to first production listed in section 2
  - 2nd section contains grammar rules
    - A → B | C would be:

```
A    : B    { code for B }
     | C    { code for C }
     ;
```

    - Inside the braces, you put C code that we want executed when the production is selected

YACC

- The C code allowed inside braces allows a special set of pseudo-variables
  - $$ refers to the lhs of the production
  - $1 refers to the 1st symbol on rhs of production
  - A : B addop C { $$ = $1 + $3 } [for calculator]
  - A : B addop C { $$ = new BinaryExpression (PLUS);
    $$->lChild = $1;
    $$->rChild = $3; } [for compiler]

- 3rd section contains top-level C routines
  - Need to specify a yyerror() routine
  - YACC will create yyparse() which is entry point for parser
  - YACC assumes a routine yylex() will be available

YACC

- When you run YACC, it tries to do LALR(1) parsing on the grammar you specify
  - It will produce an output file which shows the states it creates
  - It will show results of whether it found shift-reduce or reduce-reduce conflicts
  - If it found conflicts, it lets you know, but takes its best guess
    - Shift-reduce conflicts resolved in favor of shift
      - Fixes dangling else problem
    - Reduce-reduce resolved in favor of production listed first
      - Likely there is a problem with the grammar

- Look at YACC input file for Tiny on pg 539

Error Recovery in LR Parsers

- So, your LALR(1) parser is cooking away at the tokens in your program, and it hits a state where there is no shift or reduction specified in the table entry corresponding to the current lookahead
  - We don't want to just abort the parse
  - Need a method to get synced back up

- Options for things we could do (will do some combo)
  - Add some new state onto the stack
  - Start deleting states from the stack until you could press on
  - Start removing tokens from input stream until you can press on

- Must ensure that method will not infinite loop, even if it means consuming tokens and never recover

Error Recovery in LR Parsers

- YACC approach is pretty good
  - They add error productions at locations where they want to recover
    - Define an error token
    - Look at line 4047 on pg 539
    - Don't have to recover for all possible non-terminals
      - If you find an error somewhere in a statement, report it, and can press on parsing the next statement, that's pretty good recovery
  - When an error occurs, start popping states until get to state which contains error as a valid lookahead
    - Basically throwing away tokens we have already seen until we hit a sync point
    - Since parser is in error recovery mode, consider error to be the next token

Error Recovery in LR Parsers

- YACC approach (cont)
  - We are in state now where error is a valid lookahead, and error is the nextToken
    - Can just press on
      - Shift error onto stack
      - Do reduction of error to lhs non-terminal
  - Stack is in a stable state, but now the input string may contain residue from the production we reduced from error
    - Parser stays in error mode, and starts examining input tokens
    - If input makes sense, continue parsing
    - If input doesn't make sense, silently discard input
  - Once 3 input tokens have been shifted without another error occurring, parser exits error mode

Error Recovery in LR Parsers

- YACC approach isn't perfect, and may result in quite a bit input being discarded
  - Consider if [a == b) …
    - The entire if_stmt will be discarded, which may be hundreds of lines of code
- You would prefer to get error checking on the discarded code
- No magic solutions
  - You can tweak your grammar to add more of the error productions at critical locations
  - If you set up certain scenarios, you would probably be able to cause Netbeans or Visual C++ to miss errors
    - Of course, Visual has so many bugs of its own that it has trouble distinguishing your errors