Chapter 4

Syntactic Analysis II

1. Introduction to Bottom-Up parsing

- Grammar: E--> E+E | E*E | i
- Expression: i+i*i
- Rightmost derivation:
  - E =>E+E
  - E=> E+E*E
  - E=>E+E*i
  - E=>E+i*i
  - E=>i+i*i

1.1 Parsing with a Stack

- We will push tokens onto the stack until we see something to reduce. This something is called a "handle"
  - This is known as shifting and reducing.
- **Def:** a handle is a right hand side of a production that we can reduce to get to the preceding step in the derivation.

- We carry out the reduction by popping the right hand side off of the stack and pushing the left hand side on in its place.
- **Notice:** a handle is not just any right-hand side; it has to be the correct one -- the one that takes us one step back in the derivation.

1.2 More about Handles

- The bottom up parser's problem is to find a way of detecting when there is a handle at the top of the stack. If there is, reduce it; otherwise shift.
- For this reason bottom up parsers are often called shift-reduce parsers
- When selecting handles, some things may be the right hand side, but may not be handles.

2. The Operator-Precedence Parser

- This is the simplest bottom-up parser (and the least powerful parser for CFG's)
- It is generally simpler to construct
- table entities consist of <·, =, and ·>
- handles look like <·====·>

2.1 A Simple Operator-Precedence Parser

- Grammar:
  - E -> E + E
  - | E * E
  - | (E)
  - | i
- Table:

		In input →					
		+	*	(	)	i	$
On	+	·>	<·	<·	·>	<·	·>
stack	*	·>	·>	<·	<·	<·	·>
↓	(	<·	<·	<·	≐	<·	
	)	·>	·>		·>		·>
	i	·>	·>		·>		·>
	$	<·	<·	<·		<·	

- Algorithm:
  - Push a $ on stack and append $ to end of input
  - repeat
    - x=top teminal on stack, y is incoming
    - Find table relationship (x,y)
    - if x<·y or x=y, then shift.
    - if x·>y there is a handle on stack (<· to ·>)
    - Reduce & push LHS of production on stack.
    - If the table entry is blank, or handle is not a RHS, there is an error.
  - until x = $ and y = $ or an error is found

- Parse: i+i*i

		In input →					
		+	*	(	)	i	$
On	+	·>	<·	<·	·>	<·	·>
stack	*	·>	·>	<·	<·	<·	·>
↓	(	<·	<·	<·	≐	<·	
	)	·>	·>		·>		·>
	i	·>	·>		·>		·>
	$	<·	<·	<·		<·	

Line	Stack		Input	Production
1	$	<·	i + i * i$	
2	$<·i	·>	+ i * i$	$E \to$ i
3	$E	<·	+ i * i$	
4	$<·E +	<·	i * i$	
5	$<·E+<·i	·>	* i$	$E \to$ i
6	$<·E + E	<·	* i$	
7	$<·E +<· E *	<·	i$	
8	$<·E +<· E *<·i	·>	$	$E \to$ i
9	$<·E +<· E * E	·>	$	$E \to E * E$
10	$<·E + E	·>	$	$E \to E + E$
11	$E		$	

- Parse: (i+i)i

		In input →					
		+	*	(	)	i	$
On	+	·>	<·	<·	·>	<·	·>
stack	*	·>	·>	<·	<·	<·	·>
↓	(	<·	<·	<·	≐	<·	
	)	·>	·>		·>		·>
	i	·>	·>		·>		·>
	$	<·	<·	<·		<·	

Stack		Input	Production
$	<·	(i + i)i$	
$<·(	<·	i + i)i$	
$<·(<·i	·>	+ i)i$	$E \to$ i
$<·(E	<·	+ i)i$	
$<·(<·E +	<·	i)i$	
$<·(<·E + i	·>	)i$	$E \to$ i
$<·(<·E + E	·>	)i$	$E \to E + E$
$<·(E	≐	)i$	
$<·(≐E)		i$	

- Parse: ( )

		+	*	(	)	i	$
On stack ↓	+	·>	<·	<·	·>	<·	·>
	*	·>	·>	<·	<·	<·	·>
	(	<·	<·	<·	≐	<·	
	)	·>	·>		·>		·>
	i	·>	·>		·>		·>
	$	<·	<·	<·		<·	

(In input →)

Stack		Input	Production
$	<·	( )$	
$<(	≐	)$	
$<(≐)	·>	$	$E \rightarrow (E)$

2.2 Forming the Table

- Grammar Restrictions
  - 1. There must never be 2 or more consecutive non-terminals on the right hand side.
  - 2. No 2 distinct non-terminals may have the same right hand side.
  - 3. For any 2 terminals at most 1 of <·, =, or ·> may hold
  - 4.No ε productions

- Rules for building the table:
  - If a has higher precedence than b, then a ·>b and b<·a, regardless of the associativity.
  - If a and b have equal precedence, then relations depend upon associativity.
    - If left associative a·>b and b·>a
    - If right associative a<·b and b<·a

- Rules for building the table (cont.):
  - Paired operators like ( ) or [ ] are related by =
    - We force the parser to reduce expressions inside these operators by having (<·a and a·>)
    - Similarly we force the parser to reduce (E) before shifting any other terminals by having a <· (and) ·>a,
    - where a is any terminal that may legally precede (or follow)

- Rules for building the table (cont.):
  - For identifiers, i·>a and a<·i,
    - where a is any terminal that may legally precede or follow an identifier.
  - End markers have lower precedence than any other terminal.

3. The LR Parser

- The most powerful of all parsers that we will consider (Knuth, 1965)
- They can handle the widest variety of CFG's (including everything that predictive parsers and precedence parsers can handle)
- They work fast, and can detect errors as soon as possible. (as soon as the first incorrect token is encountered)

- It is also easy to extend LR parsers to incorporate intermediate code generation.
- **Def:** LR(k) -- Left to right scan of the tokens, Rightmost derivation, k-character lookahead.
- The larger the lookahead (k) the larger the table.
- Hopcroft and Ullman (1979) have shown that any deterministic CFL can be handled by an LR(1) parser, so that is the kind we will learn.

- Table Layout:
  - states | <- Terminals -> | <- Non-Terminals ->
  - ---
  - 0 | |
  - 1 | Action | go-to
  - 2 | part | part
  - ... | |
- LR Parser tables tend to be large. For economy, we don't place the productions themselves in the table; instead, we number them and refer to them in the table by number.

- place $ at end of input, state 0 on stack.
- Repeat Until input is accepted or an error
  - Let q_m be the current state (at the top of the stack) and let a_i be the incoming token.
  - Enter the action part of the table; X=Table[q_m,a_i]
  - Case X of
    - Shift q_n: Shift (that is, push) a_i onto the stack and enter State q_n. (We mark the fact that we have entered that state by pushing it onto the stack along with a_i
    - Reduce n: Reduce by means of production #n. (We do the reduction in essentially the same was as in the operator-precedence parser, except for managing the states.) When the left-hand side has been pushed, we must also place a new state on the stack using the go-to part of the table.
    - Accept: parse is complete
    - Error: Indicate input error

- The only complicated thing is reducing.
  - 1. If the right hand side of the indicated production has k symbols, pop the top k things off the stack (that is, k state-symbol pairs). This is the handle. If the right hand side is epsilon, nothing is popped.)
  - 2. Next, note the state on the top of the stack (after the handle has been popped). Suppose it is q_j.
  - 3. Suppose the left-hand side is X. Enter the go-to part of the table at [q_j, X] and note the entry. It will be a state; suppose it is q_k
  - 4. Push X and the new state q_k onto the stack.

- We will use our familiar grammar for expressions: (with productions numbered)
  - (1) E -> E + T
  - (2) E -> E - T
  - (3) E -> T
  - (4) T -> T * F
  - (5) T -> T / F
  - (6) T -> F
  - (7) F -> (E)
  - (8) F -> I

- Parse (i+i)/i

State	i	+	−	*	/	(	)	$	E	T	F
	Terminals								Nonterminals		
0	s5					s4			1	2	3
1		s6	s7					acc			
2		r3	r3	s8	s9		r3	r3			
3		r6	r6	r6	r6		r6	r6			
4	s5					s4			10	2	3
5		r8	r8	r8	r8		r8	r8			
6	s5					s4				11	3
7	s5					s4				12	3
8	s5					s4					13
9	s5					s4					14
10		s6	s7				s15				
11		r1	r1	s8	s9		r1	r1			
12		r2	r2	s8	s9		r2	r2			
13		r4	r4	r4	r4		r4	r4			
14		r5	r5	r5	r5		r5	r5			
15		r7	r7	r7	r7		r7	r7			

Line	Stack	Input	Entry	Action/Production
1	0	(i + i)/i$	s4	Shift, enter State 4
2	0(4	i + i)/i$	s5	Shift, enter State 5
3	0(4i5	+ i)/i$	r8	$F \to i$
4	0(4$F$3	+ i)/i$	r6	$T \to F$
5	0(4$T$2	+ i)/i$	r3	$E \to T$
6	0(4$E$10	+ i)/i$	s6	Shift, enter State 6
7	0(4$E$10+6	i)/i$	s5	Shift, enter State 5
8	0(4$E$10+6i5	)/i$	r8	$F \to i$
9	0(4$E$10+6$F$3	)/i$	r6	$T \to F$
10	0(4$E$10+6$T$11	)/i$	r1	$E \to E + T$
11	0(4$E$10	)/i$	s15	Shift, enter State 15
12	0(4$E$10)15	/i$	r7	$F \to (E)$
13	0$F$3	/i$	r6	$T \to F$
14	0$T$2	/i$	s9	Shift, enter State 9
15	0$T$2/9	i$	s5	Shift, enter State 5
16	0$T$2/9i5	$	r8	$F \to i$
17	0$T$2/9$F$14	$	r4	$T \to T / F$
18	0$T$2	$	r3	$E \to T$
19	0$E$1	$	acc	

- Parse i*(i-i

State	i	+	−	*	/	(	)	$	E	T	F
	Terminals								Nonterminals		
0	s5					s4			1	2	3
1		s6	s7					acc			
2		r3	r3	s8	s9		r3	r3			
3		r6	r6	r6	r6		r6	r6			
4	s5					s4			10	2	3
5		r8	r8	r8	r8		r8	r8			
6	s5					s4				11	3
7	s5					s4				12	3
8	s5					s4					13
9	s5					s4					14
10		s6	s7				s15				
11		r1	r1	s8	s9		r1	r1			
12		r2	r2	s8	s9		r2	r2			
13		r4	r4	r4	r4		r4	r4			
14		r5	r5	r5	r5		r5	r5			
15		r7	r7	r7	r7		r7	r7			

Line	Stack	Input	Entry	Action/Production
1	0	i*(i − i$	s5	
2	0i5	*(i − i$	r8	$F \to i$
3	0$F$3	*(i − i$	r6	$T \to F$
4	0$T$2	*(i − i$	s8	
5	0$T$2 * 8	(i − i$	s4	
6	0$T$2 * 8(4	i − i$	s5	
7	0$T$2 * 8(4i5	− i$	r8	$F \to i$
8	0$T$2 * 8(4$F$3	− i$	r6	$T \to F$
9	0$T$2 * 8(4$T$2	− i$	r3	$E \to T$
10	0$T$2 * 8(4$E$10	− i$	s7	
11	0$T$2 * 8(4$E$10−7	i$	s5	
12	0$T$2 * 8(4$E$10−7i	$	r8	$F \to i$
13	0$T$2 * 8(4$E$10−7$F$3	$	r6	$T \to F$
14	0$T$2 * 8(4$E$10−7$T$12	$	r2	$E \to E - T$
15	0$T$2 * 8(4$E$10	$	*** Error: no table entry for [10, $]	

3.1 Construction of LR Parsing Tables

- It is customary to cover the generation of LR parsing tables in a series of stages, showing three levels of LR parsers of increasing complexity.
  - (1) Simple LR (SLR)
  - (2) the canonical LR parser, and
  - (3) the lookahead LR (LALR) parser.

- This approach leads us into the subject by gradual stages, each building on the previous one, until we reach the LALR parser, the most practical one, which is impossibly complicated if presented without the background provided by the other 2.

- Let’s begin by introducing items that will be common for all three parsers.

Parser States

- In the LR parsers, each current state corresponds to a particular sequence of symbols at the top of the stack
- States in a FSA do two things. They reflect what has happened in the recent past, and they control how the FSA will respond to the next input.
- Hence, in the *design* of an LR parser, we must relate the state transition to what goes onto the stack.

Items

- An item is a production with a placeholder (.) telling how far we have gotten.
- The production E-> E+T gives rise to the following items.
  - [E->.E+T]
  - [E->E.+T]
  - [E->E+.T]
  - [E->E+T.]
- symbols to the left of the . are already on the stack; the rest is yet to come.

- So, putting it all together:
  - An item is a summary of the recent history of the parse.
  - An LR parser is controlled by a finite-state machine.
  - The recent history of a finite-state machine is contained in its state...So an item must correspond to a state in a LR parser

- Almost, If we have a state for each item we basically have an NDFA. Getting the LR states parallels getting the DFA from the NDFA.
- We need to tell the parser when to accept. For this we add a new "dummy" Non-Terminal Z -> E instead of reducing this production, we accept.

State Transitions

- Transitions are determined by the grammar and the item sets obtained from it.
- If we have 2 items P=[F->.(E)] and Q=[F->(.E)], then the structure of the items dictates that we have a transition on the symbol (from P to Q.

Constructing the State Table

- State 0
  - 1. put Z -> .E into the set
  - 2. for all items in the set, if there is a . before a Non-Terminal include all their initial items. (initial items are where the N-T --> .stuff (note the . is first)
  - 3. Apply 2 until nothing can be added.
- for every item in a state C->a.Xb
  - move the . past X
  - perform closure

- Our Language Example. -- by hand
  - (0) Z -> E
  - (1) E -> E + T
  - (2) E -> E - T
  - (3) E -> T
  - (4) T -> T * F
  - (5) T -> T / F
  - (6) T -> F
  - (7) F -> (E)
  - (8) F -> I

- Our Language Example. -- by hand
  - 0:
    - Z -> .E
    - E -> .E+T
    - E -> .E-T
    - E -> .T
    - T -> .T*F
    - T -> .T/F
    - T -> .F
    - F -> .(E)
    - F -> .i

- 1: (move over E from 0)
  - Z -> E.
  - E -> E.+T
  - E -> E.-T
- 2: (move over T from 0)
  - E -> T.
  - T -> T.*F
  - T -> T./F
- 3: (move over F from 0)
  - T -> F.
- 4: (move over '(' from 0)
  - F -> (.E)
  - E -> .E+T
  - E -> .E-T
  - E -> .T
  - T -> .T*F
  - T -> .T/F
  - T -> .F
  - F -> .(E)
  - F -> .i

- 5: (move over i from 0)
  - F -> i.
- 6: (move over '+' from 1)
  - E -> E+.T
  - T -> .T*F
  - T -> .T/F
  - T -> .F
  - F -> .(E)
  - F -> .i
- 7: (move over '-' from 1)
  - E -> E-.T
  - T -> .T*F
  - T -> .T/F
  - T -> .F
  - F -> .(E)
  - F -> .i
- 8: (move over '*' from 2)
  - T -> T*.F
  - F -> .(E)
  - F -> .i

- 9: (move over '/' from 2)
  - T -> T/.F
  - F -> .(E)
  - F -> .i
- 10: (move over E from 4)
  - F -> (E.)
  - E -> E.+T
  - E -> E.-T
- (over T from 4) -- same as 2
- (over F from 4) -- same as 3
- 11: (over T from 6)
  - E -> E+T.
  - T -> T.*F
  - T -> T./F
- 12: (over T from 7)
  - E -> E-T.
  - T -> T.*F
  - T -> T./F
- 13: (over F from 8)
  - T -> T*F.
- 14: (over F from 9)
  - T -> T/F.
- 15: (over ')' from 10)
  - F -> (E).

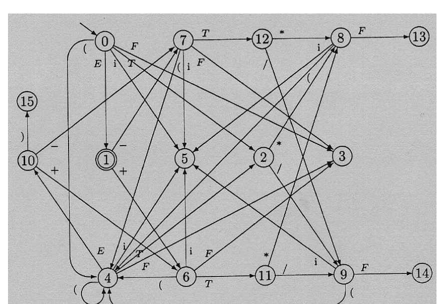

Figure 4.8

Our Language Example: Yacc grammar

- E
- : E PlusTok T
- | E MinusTok T
- | T
- ;
- T
- : T TimesTok F
- | T DivideTok F
- | F
- ;
- F
- : LParenTok E RParenTok
- | IDTok
- ;

Yacc output with states

- state 0
  - $accept : _E $end
  - IDTok shift 5
  - LParenTok shift 4
  - . error
  - E goto 1
  - T goto 2
  - F goto 3
- state 1
  - $accept : E_$end
  - E : E_PlusTok T
  - E : E_MinusTok T
  - $end accept
  - PlusTok shift 6
  - MinusTok shift 7
  - . error

- state 2
  - E : T_ (3)
  - T : T_TimesTok F
  - T : T_DivideTok F
  - TimesTok shift 8
  - DivideTok shift 9
  - . reduce 3
- state 3
  - T : F_ (6)
  - . reduce 6
- state 4
  - F : LParenTok_E RParenTok
  - IDTok shift 5
  - LParenTok shift 4
  - . error
  - E goto 10
  - T goto 2
  - F goto 3
- state 5
  - F : IDTok_ (8)
  - . reduce 8
- state ...

Filling the Rows of the State Table

- State 0:

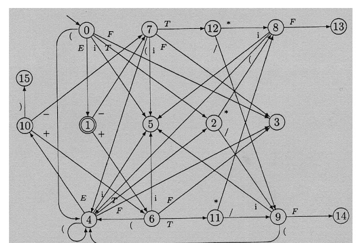

Figure 4.8

Creating Action Table Entries

- The shift entries are taken from the state table entries we just created. (terminals we moved across to get the next state).
- If a state, q, contains a completed item n:[C->β.] then for all inputs, x, in the Follow(C) reduce n is in [q,x]
- If State q contains [Z -> E.] then the action for [q,$] is "accept"

	Terminals								Nonterminals		
State	i	+	−	*	/	(	)	$	E	T	F
0	s5					s4			1	2	3
1		s6	s7					acc			
2		r3	r3	s8	s9		r3	r3			
3		r6	r6	r6	r6		r6	r6			
4	s5					s4			10	2	3
5		r8	r8	r8	r8		r8	r8			
6	s5					s4				11	3
7	s5					s4				12	3
8	s5					s4					13
9	s5					s4					14
10		s6	s7				s15				
11		r1	r1	s8	s9		r1	r1			
12		r2	r2	s8	s9		r2	r2			
13		r4	r4	r4	r4		r4	r4			
14		r5	r5	r5	r5		r5	r5			
15		r7	r7	r7	r7		r7	r7			

3.2 Error Handling

- For each empty slot in your table, you can have a unique error message.
- You could also try to guess what they left out.
- Panic Mode -- ignore everything until a ;

3.3 Conflicts

- If a grammar is not LR it will show up in the creation of the table.
- No ambiguous grammar is LR.

3.4 Canonical LR Parsers

- The SLR parser breaks down with a conflict.
- LR(1) item sets
  - standard item sets plus a lookahead symbol
  - this creates a lot more states. It is possible to have |LR(0) item set| * |terminals|

3.5 Lookahead LR (LALR) Parsers

- Why not just use the LR(0) item sets and only add a look ahead when we need one?
- They are as powerful as Canonical LR parsers.
- They are slower to detect errors (but will detect one before the next token is shifted onto the stack)

- Two ways to construct these:
  - 1. Brute Force LALR Parser Construction
    - Start with the LR(1) item sets and merge states.
  - 2. Efficient LALR Parser Construction
    - Start with the LR(0) item sets and add lookaheads as needed.

3.6 Compiler-Compilers

- YACC generates LALR(1) parser code
  - When it runs into conflicts it notifies the user
  - shift/reduce conflicts are resolved in favor of the shift.
  - operators are right associative by default

4. Summary: Which Parser Should I Use?

- We have seen several different parsing techniques, of which the most realistic are probably the table driven parsers. (predictive, operator precedence, and LR)
- Which is best? -- it seems to be personal taste.
- Now that Yacc-like parser generators are available, the LR parser seems to be the inevitable choice, but, a lot of people still write predictive, recursive-descent parsers.