CS143 Handout #9
Autumn 2000

Notes on bottom-up parsing

Handout written by Maggie Johnson and revised by me.

Bottom-up parsing

As the name suggests, bottom-up parsing begins with a string and tries to backwards to the start symbol by applying the productions in reverse. Along the way, we look for substrings in the sentential forms that match the right side of some production. When we find these substrings, we *reduce* them, i.e., we substitute the left side non-terminal of the matching right-side for the substring. The goal is to reduce our way up to the start symbol to indicate a successful parse.

Bottom-up parsing algorithms are in general more powerful than top-down methods, but not surprisingly, the constructions required in these algorithms are also more complex. It is difficult to write a bottom-up parser by hand for anything but the most trivial of grammars, but fortunately, there are some excellent parser generator tools like `yacc` that can build a parser from an input specification.

Shift-reduce parsing is the most commonly used and most powerful of the bottom-up techniques. It takes as input a stream of tokens and produces as output a list of productions to be used to build the parse tree, but the productions are discovered in reverse order of a top-down parser. Like a table-drive predictive parser, a bottom-up parser makes use of a stack to keep track of the position in the parse and a parsing table to determine what to do next.

To illustrate stack-based shift-reduce parsing, consider the following grammar:

```
S  ← A
A  ← T | A + T
T  ← b | (A)
```

The shift-reduce strategy divides the string into two parts: an undigested part and a semi-digested part. The undigested part is the input, and the semi-digested part is put on a stack. If parsing a string v, it starts out completely undigested, so the input is initialized to v, and the stack is initialized to empty. A shift-reduce parser proceeds by taking one of two actions at each step:

Reduce: If we can find a rule $A \leftarrow w$, and if the contents of the stack are qw for some q (q may be empty), then we can reduce the stack to qA. (We are applying the production backwards.) For example, using the grammar above, if the stack contained `(b` we can use the rule $T \leftarrow b$ to reduce the stack to `(T`.

There is also one special case: reducing the entire contents of the stack to the start symbol with no remaining input means you have successfully parsed all the input. (e.g. the stack contains just w, the input is empty, and you apply $S \leftarrow w$)

The w being reduced is referred to as a *handle*. Formally, a handle of a right sentential form u is a production $A \leftarrow w$, and a position within u where the string w may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of u. Recognizing valid handles is the difficult part of shift-reduce parsing.

Shift: If it is impossible to perform a reduction and there are tokens remaining in the undigested input, then we transfer a token from the input onto the stack. This is called a shift. For example, using the grammar above, suppose the stack contained `(` and the

input contained `b+b)`. It is impossible to perform a reduction on `(` since it does not match the entire right side of any of our productions. So, we shift the first character of the input onto the stack, giving us `(b` on the stack and `+b)` remaining in the input.

The other possibility in shift-reduce parsing is we might encounter an error. This occurs when we reach a state where the current token cannot be part of a valid program. This might happen in the above grammar when we try to parse `b+)`.

The general idea is to pull tokens from the input, push them onto the stack in attempt to build sequences we recognize. When we find one, we can replace that sequence with its non-terminals and work our way up the parse tree. If all goes well, we will end up moving everything from the input to the stack and will have built a sequence on the stack that we have recognized as a right-hand side for the start symbol. This process builds the parse tree from the leaves upward, the inverse of the top-down parser.

Let's trace the operation of a general shift-reduce parser in terms of its actions (shift or reduce) and its data structure (a stack). The chart below traces a shift-reduce parsing of the string `(b+b)` using the example grammar from above:

PARSE STACK	REMAINING INPUT	PARSER ACTION
	`(b+b)$`	Shift (push next token from input onto stack, advance input)
`(`	`b+b)$`	Shift
`(b`	`+b)$`	Reduce: T ⇐ b (pop right-hand side of production off stack, push left-hand side, no change in input)
`(T`	`+b)$`	Reduce: A ⇐ T
`(A`	`+b)$`	Shift
`(A+`	`b)$`	Shift
`(A+b`	`)$`	Reduce: T ⇐ b
`(A+T`	`)$`	Reduce: A ⇐ A + T (Ignore: A ⇐ T)
`(A`	`)$`	Shift
`(A)`	`$`	Reduce: T ⇐ (A)
`T`	`$`	Reduce: A ⇐ T
`A`	`$`	Reduce: S ⇐ A
`S`	`$`	

Notice in the above parse that there is a step where there are two possible productions to use in a reduction (Ignore A ⇐ T). Later, we will learn a method for determining which one to use.

Not all type 2 grammars can be parsed with a shift-reduce parser. Ambiguous grammars are problematic because these grammars could yield more than one handle under some circumstances. These types of grammars cause either *shift-reduce* or *reduce-reduce* conflicts**.** The former refers to a state where the parser cannot decide whether to shift or reduce. The latter refers to a state where

the parser has more than one choice of production for reduction. An example of a shift-reduce conflict occurs with the if-then-else construct in programming languages. A typical production might be:

S → if E then S | if E then S else S

Consider what would happen to a shift-reduce parser deriving this string:

if E then if E then S else S

At some point the parser's stack would have:

if E then if E then S

with else as the next token. It could reduce because the contents of the stack match the right-hand side of the first production or shift the else trying to build the right-hand side of the second production. Reducing would close off the inner if and thus associate the else with the outer if. Shifting would continue building and later reduce the inner if with the else. Either is syntactically valid given the grammar, but two different parse trees result, showing the ambiguity. This quandary is commonly referred to as the *dangling else*. Does an else appearing within a nested if statement belong to the inner or the outer? The C and Java languages agree that an else is associated with its nearest unclosed if. Other languages, such as Ada and Modula, avoid the ambiguity by requiring a closing endif delimiter.

Reduce-reduce conflicts are not common and usually indicate a problem in the grammar definition.

Now that we have general idea of how a shift-reduce parser operates, we will look at how it recognizes a handle, and how it decides which production to use in a reduction. To deal with these two issues, we will look at a specific shift-reduce implementation called LR parsing.

LR parsing

LR parsers ("L" for left to right scan of input; "R" for rightmost derivation) are efficient, table-driven shift-reduce parsers. The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers. In fact, virtually all programming language constructs for which CFGs can be written can be parsed with LR techniques. As an added advantage, there is no need to apply transformations and or condition the grammar to make it acceptable for LR parsing the way that LL parsing requires.

The primary disadvantage is the amount of work it takes to build the tables by hand, which makes it infeasible to hand-code an LR parser for most grammars. Fortunately, there exist LR parser generator tools that create the parser from a CFG specification. The parser tool does all the tedious and complex work to build the necessary tables and can report any ambiguities or language constructs that interfere with the ability to parse it using LR techniques.

We begin by tracing how an LR parser works. Determining the handle to reduce in a sentential form depends on the sequence of tokens on the stack, not only the topmost ones that are to be reduced, but the context at which we are in the parse. Rather than reading and shifting tokens onto a stack, an LR parser pushes "states" onto the stack; these states describe what is on the stack so far. Think of each state as encoding the current left context. The state on top of the stack combined with the next input token enables us to figure out whether we have a handle to reduce, or whether we need to read the next input token and shift a new state on top of the stack.

An LR parser uses two tables:

1. The *action table* Action[s,a] tells the parser what to do when the state on top of the stack is s and non-terminal a is the next input token. The possible actions are to shift a state onto the stack, to reduce the handle on top of the stack, to accept the input, or to report an error.
2. The *goto table* Goto[s,X] indicates the new state to place on top of the stack after a reduce of the non-terminal X while state s is on top of the stack.

The two tables are usually combined, with the action table specifying entries for terminals, and the goto table specifying entries for non-terminals.

Tracing an LR parser

We start with the initial state s_0 on the stack. The next input token is a and the current state is s_t. The action of the parser is as follows:

- If Action[s_t,a] is shift, we shift the specified state onto the stack. We then call `yylex()` to get the next token a from the input.
- If Action[s_t,a] is reduce $Y \leftarrow X_1 \ldots X_k$ then we pop k states off the stack (one for each symbol in the right side of the production) leaving state s_u on top. We check Goto[s_u,Y] to find a new state s_V to push on the stack. The input token is still a (i.e. the input remains unchanged).
- If Action[s_t,a] is accept then the parse is successful and we are done.
- If A[s_t,a] is error (the table location is blank) then we have a syntax error. With the current stack symbol and input token we can never arrive at a sentential form with a handle to reduce.

As an example, consider the following expression grammar where the productions have been sequentially numbered so we can refer to them in the action table:

```
1)  E   ←E + T
2)  E   ←T
3)  T   ←T * F
4)  T   ←F
5)  F   ←(E)
6)  F   ←id
```

Here is the combined action and goto table. In the action columns sN means shift state numbered N onto the stack number and rN action means reduce using production numbered N. The goto column entries are the number of the new state to push onto the stack after reducing the specified non-terminal.

State on top of stack	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
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			

Here is a parse of id * id + id using the LR algorithm with the above action and goto table:

STACK STACK	REMAINING INPUT	PARSER ACTION
S_0	id * id + id$	Shift S_5 onto state stack, move ahead in input
S_0S_5	* id + id$	Reduce 6) F ⇐id, pop state stack, goto S_3, input unchanged
S_0S_3	* id + id$	Reduce 4) T ⇐F, goto S_2
S_0S_2	* id + id$	Shift S_7
$S_0S_2S_7$	id + id$	Shift S_5
$S_0S_2S_7S_5$	+ id$	Reduce 6) F ⇐id, goto S_{10}
$S_0S_2S_7S_{10}$	+ id$	Reduce 3) T ⇐T*F, goto S_2
S_0S_2	+ id$	Reduce 2) E ⇐T, goto S_1
S_0S_1	+ id$	Shift S_6
$S_0S_1S_6$	id$	Shift S_5
$S_0S_1S_6S_5$	$	Reduce6) F ⇐id, goto S_3
$S_0S_1S_6S_3$	$	Reduce4) T ⇐F, goto S_9
$S_0S_1S_6S_9$	$	Reduce 1) E ⇐E+T, goto S_1
S_0S_1	$	Accept

Types of LR parsers

There are three types of LR parsers: *LR(k), simple LR(k),* and *lookahead LR(k)* (abbreviated to LR(k), SLR(k), LALR(k))). We are usually interested in a single token lookahead (k=1). The different classes of parses all operate the same way (as shown above, being driven by their action and goto tables), but they differ in how their action and goto tables are constructed, and the size of those tables.

LR(k) parsing is the most general form of LR parsing, capable of parsing any language that can be parsed by a shift-reduce parser. The problem with LR(k) however, is these parsers have many thousands of states for a typical programming language. SLR(k) parsing is a variant with only a few hundred states for a typical programming language, but it is the weakest of the three in terms of the number of grammars for which it is applicable. LALR(k) parses a larger set of languages than SLR(k) but not quite as many as LR(k). In particular, LALR(k) parsers can parse all common programming language constructs with the same number of states as the equivalent SLR parser (but LALR is harder to construct). LALR(1) is the method used by the yacc parser generator.

In order to begin to understand how LR parsers work, we need to delve into how their tables are derived. The tables contain all the information that drives the parser. As an example, we will show how to derive SLR(1) parsing tables since they are the simplest (although none of them is particularly simple), and then discuss how to do LR(1) and LALR(1).

The essence of LR parsing is building the tables that tell us where there is a handle on the top of the stack that ca be reduced. Recognizing a handle is actually "easier" than predicting a production was in top-down parsing. The weakness of LL(k) parsing techniques is that they must be able to predict which product to use, having seen only k symbols of the right-hand side.. For LL(1), this means just one symbol has to tell all. In contrast, for an LR(k) grammar is able to postpone the decision until it has seen tokens corresponding to the entire right-hand side (plus k more tokens of lookahead). This doesn't mean the task is trivial. More than one production may have the same right-hand side and what looks like a right-hand side may not really be because of its context. The addition of nullable non-terminals is another complication because they can be matched in any parsing context.

Constructing SLR(1) parsing tables

Generating an LR parsing table consists largely of identifying what states are necessary and arranging the transitions among them. At the heart of the table construction is the notion of an LR(0) *configuration* or *item.* A configuration is a production of the grammar with a dot at some position on its right side. For example, A ← XYZ gives four items:

A ← • XYZ
A ← X • YZ
A ← XY • Z
A ← XYZ •

This dot marks how far we have gotten in parsing the production. Everything to the left of the dot has been shifted and next input token is in the `First()` set of the symbol after the dot (or in the follow set if that symbol is nullable). The state we push on the stack will correspond to specific sets of these configurations, indicating what we have read thus far. A dot at the right end of a configuration indicates that we have that entire configuration on the stack i.e., we have a handle that we can reduce. A dot in the middle of the configuration indicates that we need to shift a token that could start the symbol following the dot. For example, if we are currently in this position:

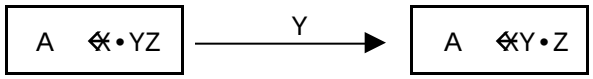
A ← X • YZ

We want to shift something from `First(Y)` (something that matches the next input token). Say we have a production Y ← u | w. Given that, these three productions all correspond to the same state of the shift-reduce parser:

A ← X • YZ
Y ← u
Y ← w

In other words, at some point as we parse a string in this language, we might expect to see a substring derivable from YZ as input. We might just as likely expect to see a string derivable from either u or w. We can put these three items into a set and call it a *configurating set* of the SLR parser. The action of adding equivalent configurations to create a configurating set is called *closure*. Our parsing tables will have one state corresponding to each configurating set (see the first column of the action and goto table above).

These configurating sets represent states that the parser can be in as it parses a string. We could model this as a finite automaton where we move from one state to another via transitions marked with a symbol of the CFG. For example,



Recall that we push states onto the stack in a LR parser. These states describe what is on the stack so far. The state on top of the stack combined with the next input token enables us to figure out whether we have a handle to reduce, or whether we need to read the next input token and shift a new state on top of the stack. We shift until we reach a state where the dot is at the end of a production, at which point we reduce. This finite automaton is the basis for a LR parser: each time we perform a shift we are following a transition to a new state.

Now for the formal rule for what to put in a configurating set. We start with a configuration:

$$A \leftarrow \alpha_1 \ldots X_i \bullet X_{i+1} \ldots X_j$$

which we place in the configurating set. We then perform the closure operation on the items in the configurating set. For each item in the configurating set where the dot precedes a non-terminal, we add configurations derived from the productions defining that non-terminal with the dot at the start of the right side of those productions. So, if we have

$$X_{i+1} \leftarrow \gamma_1 \ldots Y_g \mid Z_1 \ldots Z_h$$

in the above example, we would add the following to the configurating set.

$$X_{i+1} \leftarrow Y_1 \ldots Y_g$$
$$X_{i+1} \leftarrow Z_1 \ldots Z_h$$

We repeat this operation for all configurations in the configurating set where a dot precedes a non-terminal until no more configurations can be added. So, if Y_1 and Z_1 are terminals in the above example, we would just have the three productions in our configurating set. If they are non-terminals, we would need to add the Y_1 and Z_1 productions as well.

In summary, to create a configurating set for the starting configuration A ← u:

1. A ← u is in the configurating set
2 If u begins with a terminal, we are done with this production
3. If u begins with a non-terminal B, add all productions with B on the left side, with the dot at the start of the right side: B ← v
4. Repeat steps 2 and 3 for any productions added in step 3. Continue until you reach a fixed point.

One final thing before we build our tables: we need to define the *successor function* as it applies to a configurating set and a grammar symbol. Given a configurating set C and a grammar symbol X, we need to compute a successor configurating set C' = succ(C,X). We take all the configurations in C where there is a dot preceding X, move the dot past X and put the new configurations in C'. The succesor configurating set is the closure of C'. The successor configurating set C' represents the state the FA moves to when it encounters symbol X when in state C.

An example will make this more clear. Consider the following configuration from our example expression grammar:

E ← • + T

To obtain the successor configurating set on + we first put the following configuration in C':

E ← + • T

We then perform a closure on this set:

E ← + • T
T ← T * F
T ← F
F ← (E)
F ← id

Now, to create the action and goto tables, we need to construct all the configurating sets and successor functions for the expression grammar. At the highest level, we want to start with a configuration with a dot before the start symbol and move to a configuration with a dot after the start symbol. This represents shifting and reducing an entire sentence of the grammar. To do this, we need the start symbol to appear on the right side of a production. This may not happen in the grammar so we have to create a special production. We can create an *augmented grammar* by just adding the production:

S' ← S

where S is the start symbol. So we start with the initial configurating set C_0 which is the closure of S' ← S. The augmented grammar for the example expression grammar:

0) E' ← E
1) E ← E + T
2) E ← T
3) T ← T * F
4) T ← F
5) F ← (E)
6) F ← id

We create the complete family F of configurating sets as follows:

1. Start with F containing the configurating set C_0, derived from the configuration S' ← S
2. For each configurating set C in F and each grammar symbol X such that successor(C,X) is not empty, add C to F
3. Repeat step 2 until no more configurating sets can be added to F

Configurating set			Successor
I_0:	E'	←E	I_1
	E	←E+T	I_1
	E	←T	I_2
	T	←T*F	I_2
	T	←F	I_3
	F	←(E)	I_4
	F	←id	I_5
I_1:	E'	E←	%0
	E	E←+T	I_6
I_2:	E	T←	%2
	T	T←*F	I_7
I_3:	T	F←	%4
I_4:	F	(←E)	I_8
	E	←E+T	I_8
	E	←T	I_2
	T	←T*F	I_2
	T	←F	I_3
	F	←(E)	I_4
	F	←id	I_6
I_5:	F	id←	%6
I_6:	E	E+←T	I_9
	T	←T*F	I_9
	T	←F	I_3
	F	←(E)	I_4
	F	←id	I_5
I_7:	T	T*←F	I_{10}
	F	←(E)	I_4
	F	←id	I_5
I_8:	F	(E←)	I_{11}
	E	E←+T	I_6
I_9:	E	E+T←	%1
	T	T←*F	I_7
I_{10}:	T	T*F←	%3
I_{11}:	F	(E)←	%5

Note that the order of defining the sets is not important; what is important is that all the sets are included. A %n in the successor column indicates that the dot is at the end of production numbered n in this state, indicating there is a valid handle on the stack that can be reduced.

The successor function defines an FA that recognizes *viable prefixes* (the set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser). In other words, there will be a transition in this FA from $A \rightarrow \underline{u} \bullet X\underline{v}$, to $A \rightarrow \underline{u}X \bullet \underline{v}$ labeled X. This FA, because of the way we have constructed it, accepts viable prefixes derivable from the grammar being parsed. (The FA is pictured on p.226 of Aho/Sethi/Ullman if you're curious).

To construct the tables we use the following algorithm. The input is an augmented grammar G' and the output is the action/goto tables:

1. Construct $F = \{I_0, I_1, \ldots I_n\}$, the collection of configurating sets for G'.
2. State i is determined from I_i. The parsing actions for the state are determined as follows:
   a) If $A \rightarrow \underline{u} \bullet a\underline{v}$ is in I_i and $succ(I_i, a) = I_j$, then set Action[i,a] to shift j (a must be a terminal).
   b) If $A \rightarrow \underline{u} \bullet$ is in I_i then set Action[i,a] to reduce $A \rightarrow \underline{u}$ for all a in Follow(A) (note A may not be S').
   c) If $S' \rightarrow S \bullet$ is in I_i then set Action[i,$] to accept.
3. The goto transitions for state i are constructed for all non-terminals A using the rule: If $succ(I_i, A) = I_j$, then Goto [i, A] = j.
4. All entries not defined by rules 2 and 3 are errors.
5. The initial state is the one constructed from the configurating set containing $S' \rightarrow \bullet S$.

Notice how the shifts in the action table and the goto table are just transition tables for the FA, so we can get that information from the FA, i.e., the successor functions. The reductions are where we have a handle on the stack that we pop off and replace with the non-terminal for the handle; this occurs in the states where the • is at the end of a production. We get this information from creating the follow sets and looking at the configurating sets where a configuration has a dot at the end.

Follow(E) = { $ +) } Follow(T) = { $ + *) } Follow(F) = { $ + *) }

Recall a follow set lists the terminals that can start sequences that follow a given non-terminal. We decide which production to use for reduction based on the terminal symbol following A in the input. In the table, we only want to have entries for symbols in Follow(A); any other symbol would be an error (if such a symbol is next in the input, it is not something defined by the grammar as appropriate).

At this point, we should go back and look at the parse of id * id + id from before and trace what the states mean. (Refer to the action and goto tables and the parse diagrammed on page 4 and 5).

First of all, here is the actual parse (notice it is an inverse rightmost derivation, if you read from the bottom upwards, it is always the rightmost non-terminal that operated on).

id * id + id
F * id + id
T * id + id
T * F + id
T + id
E + id

```
E + F
E + T
E
E'
```

We start by pushing s_0 on the stack. The first token we read is an id. In configurating set I_0, the successor of id is set I_5, which means pushing s_5 onto the stack. This is a final state for id (the • is at the end of the production). In the table row for s_5, we have r6 for all terminal symbols that can follow the non-terminal F. The next input is * which is in the follow set and thus Action[5, *] is reduce F ⇐ id. We pop s_5 to simulate that we have processed the id and we are get back to state s_0. We just advanced past F, so we use the goto part of the table. Goto[0, F] tells us to push s_3 on the stack. This entry came from the fact that in set I_0 the successor for F was set I_3. The only configuration in I_3 is a final state (• at the end) so we want to reduce. For s_3 in the table we have a reduce T ⇐ F for all the symbols that can follow T in the input. The next input token is * which is valid to follow T so we pop off the s_3 state and are back in s_0. Goto[0, T] tells us to push s_2; set I_2 is where we encounter a * after the •. From set I_2 seeing a * takes us to set I_7 (push s_7 on the stack).

From set I_7 we encounter an id and that takes us to set I_5 (push s_5 on the stack). I_5 is a final state for id (the • is at the end of the production) which means we do a reduce. We have a + coming up which can follow non-terminal F and thus Action[5, +] is reduce F ⇐ id. We pop s_5 to simulate the fact that we have processed the id and we are back in state s_7. We use the goto table Goto[7, F] to get to set I_{10}. Since there is only production in this set, we must be on our way to recognizing T ⇐ T * F. Action[10, +] tells us to reduce by T ⇐ T * F. We pop the top three states off (one for each symbol in the right-hand side of the production being reduced) and we are back in s_0 again. Goto[0,T] tells us to push s_2 (which is where E ⇐ •F is located). Action[2, +] tells us to reduce by E ⇐ F. We pop off state s_2 and we are back in s_0. Goto[0, E] takes us to set I_1 (recall that the goto part of the table comes from the successor function, i.e., the lookahead is being used here). We then proceed to reduce the id to F, then to T, and then the E+T to E. The accept happens because we pop off three states s_9 (T), s_6 (+) and s_1 (E) and we are back in s_0. Because we just reduced E we goto s_1 where the next input symbol is $ which means we are using the production E' ⇐ E and the parse is complete.

The stack allows us to keep track of what we have seen so far and what we are in the middle of processing. We shift states that represent the amalgamation of the possible options onto the stack until we reach the end of a production in one of the states. Then we reduce. After a reduce, states are popped off the stack to match the symbols of the matching right-side. What's left on the stack is what we have yet to process.

Consider what happens when we try and parse id**. We start in s_0 and do the same as above to parse the id to F and then to T. Now we are in set I_7 and we encounter another * which is an error because the action table is empty for that transition. This means there was no successor for * from that configurating set.

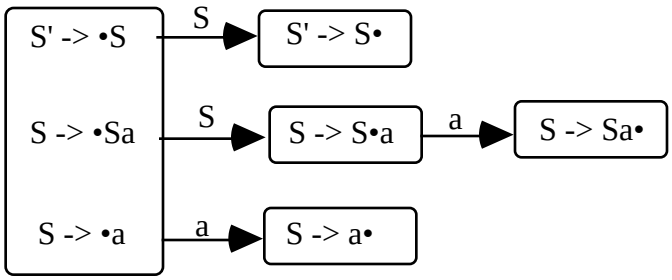
Subset construction and closure

You may have noticed a similarity between subset construction and the closure operation. If you think back to a few lectures, we explored the subset construction algorithm for converting an NFA into a DFA. The basic idea was create new states that represent the non-determinism by grouping the possibilities that look the same at that stage and only diverging when you get more information. The same idea applies to creating the configurating sets for the grammar and the successor function

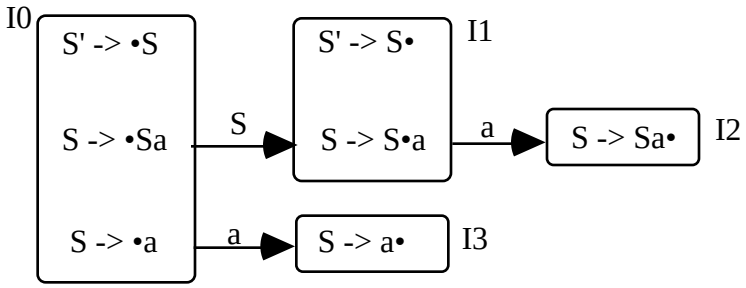
for the transitions. We create a NFA whose states are all the different individual configurations. We put all the initial configurations into one start state. Then draw all the transitions from this state to the other states where all the other states have only one configuration each. This is the NFA we do subset construction on to convert into a DFA. Here is a simple example starting from the grammar consisting of strings with one or more a's:

1) S' ⇦ S
2) S ⇦ Sa
3) S ⇦ a

Close on the augmented production and put all those configurations in a set:



Do subset construction on the resulting NFA:



This gives us our configurating sets as labeled above. Next compute the first and follow sets:

First(S') = { a }
First(S) = { a }
Follow(S') = { $ }
Follow(S) = { $ a }

Now we can fill in the action and goto tables:

State on top of stack	Action		Goto
	a	$	S
0	s3		1
1	s2	accept	
2	r 2	r 2	
3	r 3	r 3	

Here are the steps in a parse of the string aaa

STACK STACK	REMAINING INPUT	PARSER ACTION
S_0	aaa$	Shift S_3
S_0S_3	aa$	Reduce 3) S ← a, goto S_1
S_0S_1	a$	Shift S_2
$S_0S_1S_2$	a$	Reduce 2) S ← Sa, goto S_1
S_0S_1	$	Shift S_2
$S_0S_1S_2$	$	Reduce 2) S ← Sa, goto S_1
S_0S_1	$	Accept

SLR(1) grammars

A grammar is SLR(1) if the following two conditions hold for each configurating set:

1. For any item A ← u•xv in the set, with terminal x, there is no complete item B ← w• in that set with x in Follow(B). In the tables, this translates no shift-reduce conflict on any state. This means the successor function for x from that set either shifts to a new state or reduces, but not both.
2. For any two complete items A ← u• and B ← v• in the same configurating set, the follow sets must be disjoint, e.g. Follow(A) ∩ Follow(B) is empty. This translates to no reduce-reduce conflict on any state. If more than one non-terminal could be reduced from this set, it must be possible to uniquely determine which using only one token of lookahead.

LR(k) vs LL(k)

LR grammars are more general than LL grammars. Neither can handle all unambiguous grammars but all LL(1) grammars are LR(1). LL(1) grammars are very strict about grammar forms, forbidding left recursion and productions that share a common prefix. This is not the case with LR(1). You won't need to obsfucate your grammar with a lot of conditioning in order for it to be suitable for LR parsing.

Note: every SLR(1) grammar is unambiguous, but there are many unambiguous grammars that are not SLR(1). Consider this example from the Aho/Sethi/Ullman that defines a small grammar for assignment statements, using the non-terminal L for l-value and R for r-value and * for contents-of.

S ← L = R
S ← R
L ← *R
L ← id
R ← L

I_0: S' ← •S
S ← •L = R
S ← •R
L ← •*R
L ← •id
R ← •L

I_5: L ← id•

I_6: S ← S= •R
R ← •L
L ← •*R
L ← •id

I_1: S' ⟵ S•

I_2: S ⟵ L• = R
R ⟵ L•

I_3: S ⟵ R•

I_4: L ⟵ *•R
R ⟵ L
L ⟵ *R
L ⟵ id

I_7: L ⟵ R•

I_8: R ⟵ L•

I_9: S ⟵ L=R•

Consider parsing the expression `*id = id`. After working our way to configurating set I_2 having reduced `*id` to `L`, we have a choice upon seeing `=` coming up in the input. The first configuration in the set wants to set `Action[2,=]` be shift 6, which corresponds to moving on to find the rest of the assignment. However, `=` is also in `Follow(R)` because `S => L=R => *R = R`. Thus, the second configuration wants to reduce in that slot `R ⟵ L`. This is a shift-reduce conflict but not because of any problem with the grammar. It happens because with SLR, the parser cannot remember enough left context to decide what should happen when it encounters a `=` in the input having seen a string reducible to `L`. In the next lecture, we will discuss techniques that have a bit more "memory" than SLR and can avoid this problem.

Bibliography

A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools. Reading, MA: Addison-Wesley, 1986.

J.P. Bennett, Introduction to Compiling Techniques. Berkshire, England: McGraw-Hill, 1990.

C. Fischer, R. LeBlanc, Crafting a Compiler. Menlo Park, CA: Benjamin/Cummings, 1988.

K. Loudon, Compiler Construction. Boston, MA: PWS, 1997

A. Pyster, Compiler Design and Construction. New York, NY: Van Nostrand Reinhold, 1988.

J. Tremblay, P. Sorenson, The Theory and Practice of Compiler Writing. New York, NY: McGraw-Hill, 1985.