CS143 Handout 10
Summer 2011 June 29th, 2011

Bottom-Up Parsing

Handout written by Maggie Johnson and revised by Julie Zelenski.

Bottom-up parsing

As the name suggests, bottom-up parsing works in the opposite direction from top-down. A top-down parser begins with the start symbol at the top of the parse tree and works downward, driving productions in forward order until it gets to the terminal leaves. A bottom-up parse starts with the string of terminals itself and builds from the leaves upward, working backwards to the start symbol by applying the productions in reverse. Along the way, a bottom-up parser searches for substrings of the working string that match the right side of some production. When it finds such a substring, it *reduces* it, i.e., substitutes the left side nonterminal for the matching right side. The goal is to reduce all the way up to the start symbol and report a successful parse.

In general, bottom-up parsing algorithms are more powerful than top-down methods, but not surprisingly, the constructions required are also more complex. It is difficult to write a bottom-up parser by hand for anything but trivial grammars, but fortunately, there are excellent parser generator tools like **`bison`** that build a parser from an input specification, not unlike the way **`flex`** builds a scanner to your spec.

Shift-reduce parsing is the most commonly used and the most powerful of the bottom-up techniques. It takes as input a stream of tokens and develops the list of productions used to build the parse tree, but the productions are discovered in reverse order of a top-down parser. Like a table-driven 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 this simplified expression grammar:

```
S   –>   E
E   –>   T | E + T
T   –>   id | (E)
```

The shift-reduce strategy divides the string that we are trying parse into two parts: an undigested part and a semi-digested part. The undigested part contains the tokens that are still to come in the input, and the semi-digested part is put on a stack. If parsing the string $\underline{v}$, it starts out completely undigested, so the input is initialized to $\underline{v}$, and the stack is initialized to empty. A shift-reduce parser proceeds by taking one of three actions at each step:

Reduce: If we can find a rule A –> 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 for the nonterminal A backwards. For example, using the grammar above, if the stack contained (id we can use the rule T –> id 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 we have recognized the input as a valid sentence (e.g., the stack contains just w, the input is empty, and we apply S –> w). This is the last step in a successful parse.

The w being reduced is referred to as a *handle*. Formally, a handle of a right sentential form u is a production A –> 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 id+id). 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 (id on the stack and +id) remaining in the input.

Error: If neither of the two above cases apply, we have an error. If the sequence on the stack does not match the right-hand side of any production, we cannot reduce. And if shifting the next input token would create a sequence on the stack that cannot eventually be reduced to the start symbol, a shift action would be futile. Thus, we have hit a dead end where the next token conclusively determines the input cannot form a valid sentence. This would happen in the above grammar on the input id+). The first id would be shifted, then reduced to T and again to E, next + is shifted. At this point, the stack contains E+ and the next input token is). The sequence on the stack cannot be reduced, and shifting the) would create a sequence that is not viable, so we have an error.

The general idea is to read tokens from the input and push them onto the stack attempting to build sequences that we recognize as the right side of a production. When we find a match, we replace that sequence with the nonterminal from the left side and continue working our way up the parse tree. This process builds the parse tree from the leaves upward, the inverse of the top-down parser. If all goes well, we will end up moving everything from the input to the stack and eventually construct a sequence on the stack that we recognize as a right-hand side for the start symbol.

Let's trace the operation of a shift-reduce parser in terms of its actions (shift or reduce) and its data structure (a stack). The chart below traces a parse of (id+id) using the previous example grammar:

PARSE STACK	REMAINING INPUT	PARSER ACTION
	(id + id)$	Shift (push next token from input on stack, advance input)
(	id + id)$	Shift
(id	+ id)$	Reduce: T –> id (pop right-hand side of production off stack, push left-hand side, no change in input)
(T	+ id)$	Reduce: E –> T
(E	+ id)$	Shift
(E +	id)$	Shift
(E + id	)$	Reduce: T –> id
(E + T	)$	Reduce: E –> E + T (Ignore: E –> T)
(E	)$	Shift
(E)	$	Reduce: T –> (E)
T	$	Reduce: E –> T
E	$	Reduce: S –> E
S	$	

In the above parse on step 7, we ignored the possibility of reducing E –> T because that would have created the sequence (E + E on the stack which is not a *viable prefix* of a right sentential form. Formally, viable prefixes are the set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser, i.e. prefixes of right sentential forms that do not extend past the end of the rightmost handle. Basically, a shift-reduce parser will only create sequences on the stack that can lead to an eventual reduction to the start symbol. Because there is no right-hand side that matches the sequence (E + E and no possible reduction that transforms it to such, this is a dead end and is not considered. Later, we will see how the parser can determine which reductions are valid in a particular situation.

As they were for top-down parsers, ambiguous grammars are problematic for bottom-up parsers because these grammars could yield more than one handle under some circumstances. These types of grammars create 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 rare 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 LL 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 for lots of grammar rearrangement 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 are LR parser generators that create the parser from an unambiguous 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 possibly augmented by peeking at a lookahead token enables us to figure out whether we have a handle to reduce, or whether we need to shift a new state on top of the stack for the next input token.

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 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 reduction of the nonterminal 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 nonterminals.

LR Parser Tracing

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

- If Action[s_t,a] is shift, we push 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 –> $X_1...X_k$ 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. Goto[s_u,Y] gives 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 Action[s_t,a] is error (the table location is blank) then we have a syntax error. With the current top of stack and next input we can never arrive at a sentential form with a handle to reduce.

As an example, consider the following simplified expression grammar. 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 –> (E)
4) T –> 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 nonterminal. This is an LR(0) table (more details on table construction will come in a minute).

State on top of stack	Action					Goto	
	id	+	(	)	$	E	T
0	s4		s3			1	2
1		s5			accept		
2	r2	r2	r2	r2	r2		
3	s4		s3			6	2
4	r4	r4	r4	r4	r4		
5	s4		s3				8
6		s5		s7			
7	r3	r3	r3	r3	r3		
8	r1	r1	r1	r1	r1		

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

STATE STACK	REMAINING INPUT	PARSER ACTION
S_0	id + (id)$	Shift S_4 onto state stack, move ahead in input
S_0S_4	+ (id)$	Reduce 4) T –> id, pop state stack, goto S_2, input unchanged
S_0S_2	+ (id)$	Reduce 2) E –> T, goto S_1
S_0S_1	+ (id)$	Shift S_5
$S_0S_1S_5$	(id)$	Shift S_3
$S_0S_1S_5S_3$	id)$	Shift S_4
$S_0S_1S_5S_3S_4$	)$	Reduce 4) T –> id, goto S_2
$S_0S_1S_5S_3S_2$	)$	Reduce 2) E –> T, goto S_6
$S_0S_1S_5S_3S_6$	)$	Shift S_7
$S_0S_1S_5S_3S_6S_7$	$	Reduce 3) T –> (E), goto S_8
$S_0S_1S_5S_8$	$	Reduce 1) E –> E + T, goto S_1
S_0S_1	$	Accept

LR Parser Types

There are three main types of LR parsers: *LR(k), simple LR(k),* and *lookahead LR(k)* (abbreviated to LR(k), SLR(k), LALR(k))). The k identifies the number of tokens of lookahead. We will usually only concern ourselves with 0 or 1 tokens of lookahead, but the techniques do generalize to k > 1. The different classes of parsers 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.

We will consider LR(0) parsing first, which is the simplest of all the LR parsing methods. It is also the weakest and although of theoretical importance, it is not used much in practice because of its limitations. LR(0) parses without using any lookahead at all. Adding just one token of lookahead to get LR(1) vastly increases the parsing power. Very few grammars can be parsed with LR(0), but most unambiguous CFGs can be parsed with LR(1). The drawback of adding the lookahead is that the algorithm becomes somewhat more complex and the parsing table gets much, much bigger. The full LR(1) parsing table for a typical programming language has many thousands of states compared to the few hundred needed for LR(0). A compromise in the middle is found in the two variants SLR(1) and LALR(1) which also use one token of lookahead but employ techniques to keep the table as small as LR(0). SLR(k) is an improvement over LR(0) but much weaker than full LR(k) 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). 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 construct an LR(0) parsing table since they are the simplest and then discuss how to do SLR(1), LR(1), and LALR(1) in later handouts.

The essence of LR parsing is identifying a handle on the top of the stack that can 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. But in general, the fact that we see the entire right side before we have to commit to a production is a useful advantage.

Constructing LR(0) parsing tables

Generating an LR parsing table consists identifying the possible states 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 has four possible 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 onto the parsing stack 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). 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 to continue further, we need to shift a token that could start the symbol following the dot. For example, if we are currently in this position:

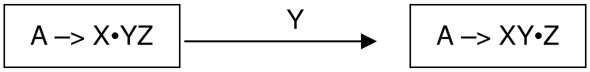
A –> X•YZ

We want to shift something from First(Y) (something that matches the next input token). Say we have productions 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

At the above point in parsing, we have just recognized an X and expect the upcoming input to contain a sequence derivable from YZ. Examining the expansions for Y, we furthermore expect the sequence to be derivable from either u or w. We can put these three items into a set and call it a *configurating set* of the LR 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.

These configurating sets represent states that the parser can be in as it parses a string. Each state must contain all the items corresponding to each of the possible paths that are concurrently being explored at that point in the parse. 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 (potentially combined with some lookahead) 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:

$$A \rightarrow X_1 \ldots X_i \bullet X_{i+1} \ldots X_j$$

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 nonterminal, we add configurations derived from the productions defining that nonterminal with the dot at the start of the right side of those productions. So, if we have

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

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

$$X_{i+1} \rightarrow \bullet\, Y_1 \ldots Y_g$$
$$X_{i+1} \rightarrow \bullet\, Z_1 \ldots Z_h$$

We repeat this operation for all configurations in the configurating set where a dot precedes a nonterminal 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 nonterminals, 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 $A \rightarrow \bullet\underline{u}$, we follow the closure operation:

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

The other information we need to build our tables is the transitions between configurating sets. For this, we define the *successor* function. Given a configurating set C and a grammar symbol X, the successor function computes the successor configurating set $C' = \text{successor}(C,X)$. The successor function describes what set the parser moves to upon recognizing a given symbol.

The successor function is quite simple to compute. 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', then we apply the closure operation to C'. The successor configurating set C' represents the state we move to when encountering symbol X in state C.

The successor function is defined to only recognize viable prefixes. There is a transition from A –> $\underline{u}$•x$\underline{v}$ to A –> $\underline{u}$x•$\underline{v}$ on the input x. If what was already being recognized as a viable prefix and we've just seen an x, then we can extend the prefix by adding this symbol without destroying viability.

Here is an example of building a configurating set, performing closure, and computing the successor function. Consider the following item from our example expression grammar:

```
E  –> E •+ T
```

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

```
E  –> E +•T
```

We then perform a closure on this set:

```
E  –> E +•T
T  –> •(E)
T  –> • 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 modify it. We create an *augmented grammar* by 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  –> (E)
4) T  –> id
```

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

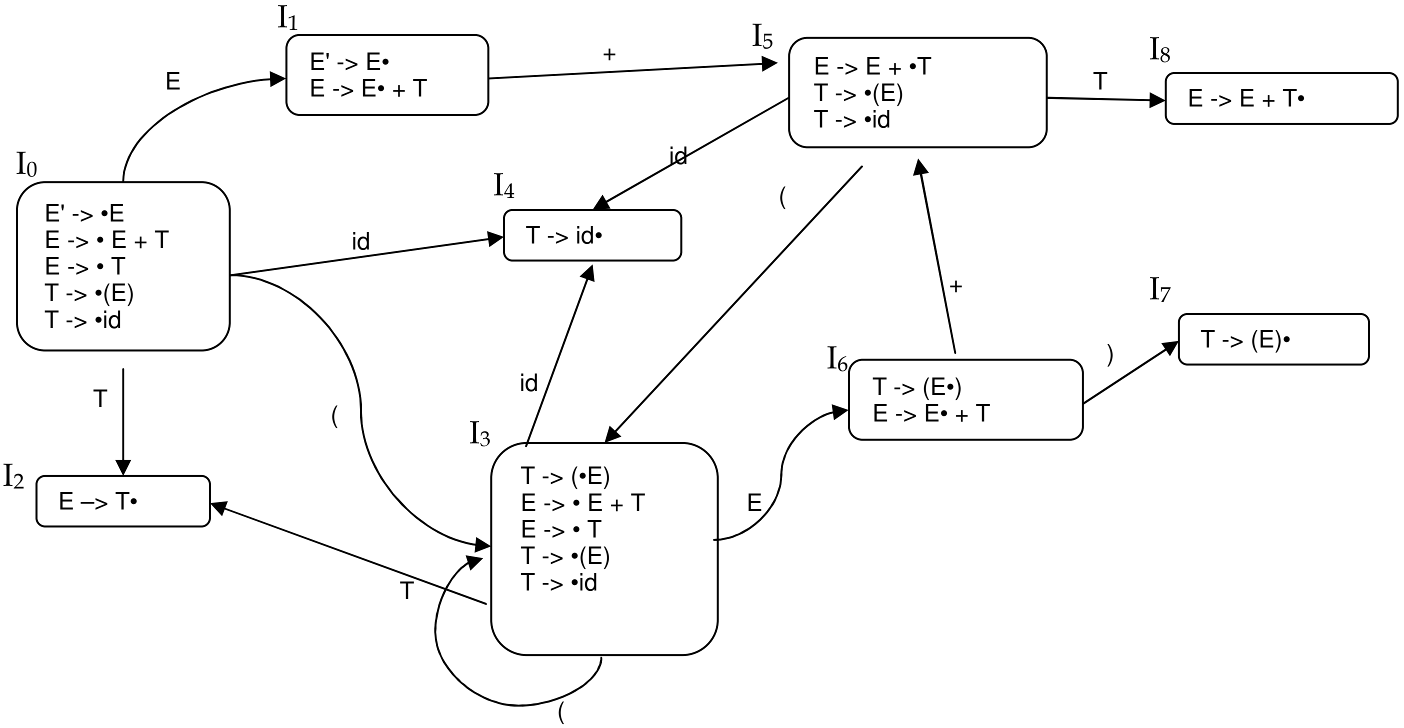
1. Start with F containing the configurating set C_0, derived from the configuration S' –> • S
2. For each configurating set C in F and each grammar symbol X such that successor(C,X) is not empty, add successor(C,X) to F
3. Repeat step 2 until no more configurating sets can be added to F

Here is the full family of configurating sets for the grammar given above.

Configurating set		***Successor***
I_0:	E' –> •E	I_1
	E –> •E+T	I_1
	E –> •T	I_2
	T –> •(E)	I_3
	T –> •id	I_4
I_1:	E' –> E•	Accept
	E –> E•+T	I_5
I_2:	E –> T•	Reduce 2
I_3:	T –> (•E)	I_6
	E –> •E+T	I_6
	E –> •T	I_2
	T –> •(E)	I_3
	T –> •id	I_4
I_4:	T –> id•	Reduce 4
I_5:	E –> E+•T	I_8
	T –> •(E)	I_3
	T –> •id	I_4
I_6:	T –> (E•)	I_7
	E –> E•+T	I_5
I_7:	T –> (E)•	Reduce 3
I_8:	E –> E+T•	Reduce 1

Note that the order of defining and numbering the sets is not important; what is important is that all the sets are included.

A useful means to visualize the configurating sets and successors is with a diagram like the one shown below. The transitions mark the successor relationship between sets. We call this a *goto-graph* or *transition diagram.*



To construct the LR(0) table, 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, ... 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 –> $\underline{u}$• is in I_i then set Action[i,a] to reduce A –> $\underline{u}$ for all input. (A not equal to S').
    b) If S' –> S• is in I_i then set Action[i,$] to accept.

   c) If A –> $\underline{u}$•a$\underline{v}$ is in I_i and successor(I_i, a) = I_j, then set Action[i,a] to shift j (a is a terminal).

3. The goto transitions for state i are constructed for all nonterminals A using the rule: If successor($I_{i,}$ 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' –> •S.

Notice how the shifts in the action table and the goto table are just transitions to new states. The reductions are where we have a handle on the stack that we pop off and replace with the nonterminal for the handle; this occurs in the states where the • is at the end of a production.

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

Here is the parse (notice it is an reverse rightmost derivation, if you read from the bottom upwards, it is always the rightmost nonterminal that was operated on).

id + (id)	T –> id
T + (id)	E –> T
E + (id)	T –> id
E + (T)	E –> T
E + (E)	T –> (E)
E + T	E –> E+T
E	E' –> E
E'	

Now let's examine the action of the parser. 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 $I_{4,}$ this means pushing s_4 onto the stack. This is a final state for id (the • is at the end of the production) so we reduce the production T –> id. We pop s_4 to match the id being reduced and we are back in state s_0. We reduced the handle into a T, so we use the goto part of the table, and Goto[0, T] tells us to push s_2 on the stack. (In set I_0, the successor for T was set I_2). In set I_2, the action is to reduce E –> T, so we pop off the s_2 state and are back in s_0. Goto[0, E] tells us to push s_1. From set I_1 seeing a + takes us to set I_5 (push s_5 on the stack).

From set I_5 we read an open (which that takes us to set I_3 (push s_3 on the stack). We have an id coming up and so we shift state s_4. Set 4 reduces T –> id, so we pop s_4 to remove right side and we are back in state s_3. We use the goto table Goto[3, T] to get to set I_2. From here we reduce E –> T, pop s_2 to get back to state s_3 now we goto s_6. . Action[6,)] tells us to shift s_7. Now in s_7 we reduce T –> (E). 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_5 again. Goto[5,T] tells us to push s_8. We reduce by E –> E + T which pops off three states to return to s_0. Because we just reduced E we goto s1. The next input symbol is $ means we completed the production E' –> E and the parse is successful.

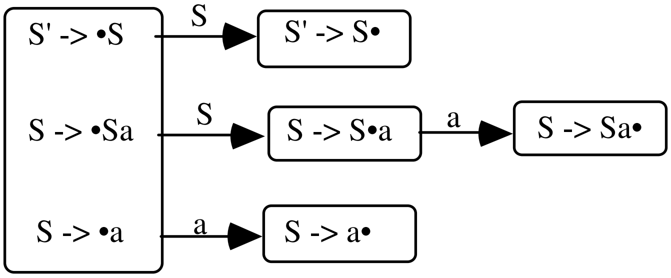
The stack allows us to keep track of what we have seen so far and what we are in the middle of processing. We shift states that represent the amalgamation of the possible options onto the stack until we reach the end of a production in one of the states. Then we reduce. After a reduce, states are popped 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 to parse id++. We start in s0 and do the same as above to reduce the id to T and then to E. Now we are in set I_5 and we encounter another +. This is an error because the action table is empty for that transition. There is no successor for + from that configurating set, because there is no viable prefix that begins E++.

Subset construction and closure

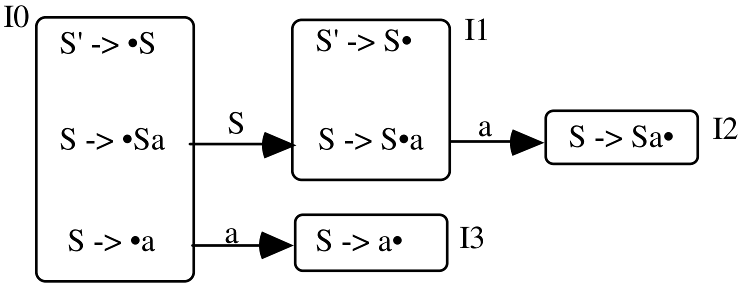
You may have noticed a similarity between subset construction and the closure operation. If you think back to a few lectures, we explored the subset construction algorithm for converting an NFA into a DFA. The basic idea was create new states that represent the non-determinism by grouping the possibilities 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 to get the configurating sets:



Interesting, isn't it, to see the parallels between the two processes? They both are grouping the possibilities into states that only diverge once we get further along and can be sure of which path to follow.

Limitations of LR(0) Parsing

The LR(0) method may appear to be a strategy for creating a parser that can handle any context-free grammar, but in fact, the grammars we used as examples in this handout were specifically selected to fit the criteria needed for LR(0) parsing. Remember that LR(0) means we are parsing with zero tokens of lookahead. The parser must be able to determine what action to take in each state without looking at any further input symbols, i.e. by only considering what the parsing stack contains so far. In an LR(0) table, each state must only shift or reduce. Thus an LR(0) configurating set cannot have both shift and reduce items, and can only have exactly one reduce item. This turns out to be a rather limiting constraint.

To be precise, a grammar is LR(0) if the following two conditions hold:

1. For any configurating set containing the item A –> $\underline{u}$•x$\underline{v}$ there is no complete item B –> $\underline{w}$• in that set. In the tables, this translates to no shift-reduce conflict on any state. This means the successor function from that set either shifts to a new state or reduces, but not both.
2. There is at most one complete item A –> $\underline{u}$• in each configurating set. This translates to no reduce-reduce conflict on any state. The successor function has at most one reduction.

Very few grammars meet the requirements to be LR(0). For example, any grammar with an ε-rule will be problematic. If the grammar contains the production A –> ε, then the item A –> •ε will create a shift-reduce conflict if there is any other non-null production for A. ε-rules are fairly common programming language grammars, for example, for optional features such as type qualifiers or variable declarations.

Even modest extensions to earlier example grammar cause trouble. Suppose we extend it to allow array elements, by adding the production rule T–>id[E]. When we construct the configurating sets, we will have one containing the items T–>id• and T–>id•[E] which will be a shift-reduce conflict. Or suppose we allow assignments by adding the productions E –> V = E and V –> id. One of the configurating sets for this grammar contains the items V–>id• and T–>id•, leading to a reduce-reduce conflict.

The above examples show that the LR(0) method is just too weak to be useful. This is caused by the fact that we try to decide what action to take only by considering what we have seen so far, without using any information about the upcoming input. By adding just a single token lookahead, we can vastly increase the power of the LR parsing technique and work around these conflicts. There are three ways to use a one token lookahead: SLR(1), LR(1) and LALR(1), each of which we will consider in turn in the next few lectures.

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.

D. Grune, H. Bal, C. Jacobs, K. Langendoen, Modern Compiler Design. West Sussex, England: Wiley, 2000.

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.