UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS 164 **R. J. Fateman**
Fall 2001

CS 164: Programming Languages and Compilers:
Parsing*

Previously, we discussed what it means to have a *derivation* of a sentence according to a grammar, and how one can use a derivation (once found) to guide that application of *semantic actions* that compute a *semantic value* (i.e., meaning or translation) of a sentence (where "sentence" can mean an entire program). In this handout, we turn to the question of how one finds such a derivation.

1 Recursive Descent (LL(1) grammars)

Context-free grammars resemble recursive programs: they are certainly recursively defined, and one can read a rule '$A \rightarrow\ BCD$' as "to parse an A in the input, first parse a B, then a C, and then a D." This insight leads to a rather intuitive form of parsing known as *recursive descent*.

1.1 From Grammars to Programs

Consider the following grammar with augments written in Lisp. The way the augments work is that if a grammar rule has 3 symbols on the right, then the augment is a Lisp function with 3 arguments, each corresponding to a symbol's semantics, in order. Presumably only the arguments corresponding to semantics of non-terminal symbols are of interest, since the others are constants.

Grammar 1.

```
s -> e $          #'(lambda (E dol) E)
e -> t e2         #'(lambda (T E2 (list '+ T E2)))
e2 -> '+' e       #'(lambda (p E) E)
```

*Adapted from the Spring 1999 notes of Prof. Paul Hilfinger.

```
e2 ->              #'(lambda () nil)
t -> '(' e ')'     #'(lambda (l E r) E)
t -> i             #'(lambda (i) (getvalue i))
```

Here, i, '+', '(', ')', and $ (end of file) are terminal symbols, and the semantic value of an i is, we'll say, a leaf node of an abstract syntax tree, or a value associated with it somehow. Depending on your outlook you will either expect a $ uniquely just at the last character before the end of the file or maybe it is just a cheap trick to make "end-of-file" easier to read in some programming languages. In Lisp we can easily make the reading program return any agreed-upon object when it reads to the end of a file. The `s` symbol (s for "sentence") is by convention the symbol representing the program we are trying to recognize.

Let's assume that there is a function `nextToken()` that returns the syntactic category of the next token of the input (one of `i, '+', '(', ')'`, and `$`, and a function `popToken()` that returns the token, removing it from the input. To be consistent with the text we combine these into a function `eat`. The program `eat` takes zero or one argument: given an argument, if the next token does not equal that argument, the procedure makes a fuss, exiting with an error. Given no argument, `eat` just pops the token off the input, returning it. We can define both versions at once in Lisp:

```
(defun eat( &optional (x nil supplied?))
  (cond ((not supplied?) (popToken))
        ((equal x (nextToken))(popToken))
        (t (error))))
```

Our strategy will be to produce a set of functions, one for each non-terminal symbol. The body of each function will directly transcribe the grammar rules for the corresponding non-terminal. To start with, we'll ignore the semantic actions:

```
(defun S () (E) eat ($)

(defun E () (T)(E2))

(defun E2 () (cond ((equal (nextToken) '+)
                    (eat)
                    (E))
                   ((equal (nextToken) '$)nil)
                   (t (error))))

(defun T ()
    (cond ((equal (nextToken) '\( )
           (eat) (E) (eat '\) ))
          ((equal (nextToken) 'i)
           (eat))
          (t (error))))
```

[Lisp programming note: the symbol `t` is used by convention for the "true" value in various places, e.g. the final clauses of the `cond` construction. While it is actually possible to have a program `t` and a symbol `t`, we will distinguish them by using capital `T` for our programs[1]. Textbooks tend to uses E' for a symbol which is inconvenient because of the special use of the single-quote. We use E2 or EP for "E prime".]

If you examine these programs closely, you should see each grammar rule is transcribed into program text. Nonterminals on the right-hand sides turn into function calls; terminals turn into calls to 'eat'. To parse a program (to start things off), you simply call '(S)'. If you trace the execution of this program for a given sentence and look at the order in which calls occur, comparing it to the parse tree for that sentence, you will see that the program essentially performs a *preorder walk* (also called "top down") of the parse tree, corresponding to a *leftmost derivation* of the tree.

Adding in semantic actions complicates things only a little. Now we make the functions return the semantic value for their tree:

```
(defun S ()
 (let ((t1 (E)))
    (eat '$ )
    t1))

(defun E ()
  (let ((t1 (T))
        (t2 (E2)))
    (list +  t1 t2)))

(defun E2 ()
  (cond ((equal (nextToken) '+)
         (eat)
          (E))
        ((equal (nextToken) '$))
        (t (error))))

(defun T ()
  (cond ((equal (nextToken) '\( )
         (eat)(prog1 (E)(eat `\) )))
        ((equal (nextToken) 'i)
         (eat))
        (t (error)))
```

[Programming note: `(prog1 a b c..  )` evaluates `a, b, c` in order, and then returns the first value (of `a`). It is shorthand for `(let ((temp a)) b c ...  temp)`.]

[1] This is not quite the end of the story: for historical reasons, ANSI Standard Lisp by default maps all symbols to uppercase. This is such a terrible convention that our lisp system is initially set to non-standard mode with regard to case.

1.2 Choosing a branch: FIRST and FOLLOW

We still haven't told you exactly where the '`cond`' constructions came from. In general, you'll be faced with several rules for a given nonterminal—let's say $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, etc., where each α_i is a string of terminal and nonterminal symbols. These recursive descent parsers work by choosing ("predicting") which of the α_i to pursue based on the next, as yet unscanned input token. Assuming first that none of the α_i can produce the empty string, we can choose the branch of the function for A that corresponds to rule $A \rightarrow \alpha_i$ if the next symbol of input is in FIRST(α_i), which (when α_i does not produce the empty string) is defined as "the set of terminal symbols that can begin a sentence produced from α_i". As long as these sets of symbols do not overlap, we can unambiguously choose which branch to take.

Suppose one of the branches, say α_k, *can* produce the empty string, in which case we define FIRST(α_k) to contain the empty string as well as any symbols that can begin α_k. We should choose the α_k branch if *either* the next input symbol is in FIRST(α_k) *or* the next input symbol is in FOLLOW(A), which is defined as "the set of terminal symbols that can come immediately after an A in some sentential form produced from the start symbol." Clearly, we're in trouble if more than one α_i can produce the empty string, so for this translation to recursive descent to work, we must insist that at most one branch can produce the empty string.

If there is no overlap in any of the sets of terminal strings produced by the procedure above, then we say that *the grammar is LL(1)*, meaning that it can be parsed Left to right to give a Leftmost derivation, looking ahead at most 1 symbol of input.

1.3 Dealing with non-LL(1) grammars.

You will have noticed, no doubt, that the grammar above is a bit odd, compared to a normal expression grammar. For one thing, it looks rather contorted, and for another, it groups expressions to the right rather than the left (it treats 'a+b+c' as 'a+(b+c)'). If we try to write a more natural grammar, however, we run into trouble:

Grammar 2.

```
A. s -> e $
B. e -> e '+' t
C. e -> t
D. t -> '(' e ')'
E. t -> i
```

The problem is that the test to determine whether to apply the first or second rule for 'e' breaks down: the same symbols can start an 'e' as can start a 't'. Another problem is that the grammar is *left recursive:* from 'e', one can produce a sentential form that begins with 'e'; in a program this causes an infinite recursion. Both of these cause the grammar to be non-LL(1).

Most books go into a great deal of hair to get around problems like this. Frankly, we can take a more practical stance. The pattern above is quite common. So much so that the grammar is often written as

```
A. s -> e $
B. e -> t { '+' t }
D. t -> '(' e ')'
E. t -> i
```

where the braces {} are grammar meta-symbols that indicate "optionally zero or more times". Thus rule B means `e` can be rewritten as `t`, or `t+t` or `t+t+t` etc. This is easily dealt with by means of a loop:

```
(defun E()
  (let ((t1 (T)))
    (while (equal (nextInput) '+)
        (eat)
        (setf t1 (list '+ t1 (T))))
    t1))

;; for fans of the Lisp ``do'' form, here's another version
(defun E()
 (do ((t1 (T)(list '+ t1 (T))))
     ((not (equal (nextInput) '+)) t1)
    (eat)))
```

At this point it may occur to you that

1. Writing a parser for a large grammar (hundreds of rules) can represent a major expenditure of time writing rather boring repetitive rules.
2. There should be a program that can do this automatically, since there is a clear correspondence between rules and programs.

Indeed there are programs that can take a grammar, produce an LL table, check for conflicts, and do the appropriate dance to construct a parser. If time permits, we will show you one.

Automatically constructed parsers are usually not based on LL(1) principles. We turn instead to an alternative approach.

2 Bottom-up Parsing

You can think of the difficulties with LL(1) parsing—reflected in the need to munge the grammars or "cheat" with special loops—as arising from their *predictive* requirements. A pure recursive descent parser chooses which right-hand side to use on the basis of just the *first* terminal symbol matched by that right-hand side. If we could instead wait until we had read an entire (alleged) right-hand side, we might do a better job. This is in fact the case, but designing such parsers (called bottom-up parsers) requires accurate computation of some detailed tables, difficult to do by hand. Fortunately there is no need to write them by hand because automatic computation aids exist, in the form of tools such as Yacc or Bison, or CUP (mentioned in Appel's text) or Johnson's LALR generator (which we will use for Lisp code). We could simply end the story here and tell you to use one such tools, but it is conventional in CS 164 to encourage you to learn how your tools work before starting to treat them as magic.

2.1 Shift-reduce Parsing

Let's again consider Grammar 2 from above, and look at a *reverse* derivation of the string 'i+(i+i)':

```
 1. i       + ( i + i ) $
 2. t       + ( i + i ) $
 3. e + ( i       + i ) $
 4. e + ( t       + i ) $
 5. e + ( e + i       ) $
 6. e + ( e + t       ) $
 7. e + ( e )           $
 8. e + t               $
 9. e $
10. s
```

Read from the bottom up, this is a straightforward rightmost derivation, but with a mysterious gap in the middle of each sentential form. The gap marks the position of the *handle* in each sentential form—the portion of the sentential form up to and including the symbols produced (reading upwards) by applying the next production or (reading downwards) the symbols about to be reduced by applying the next reverse production. Reading this downwards, you see that the gap proceeds through the input (i.e., the sentence to be parsed) from left to right. We call the symbols left of the gap "the stack" (right symbol on top) and the symbol just to the right of the gap "the lookahead symbol".

To add semantic actions, we just apply the rules attached to a given production each time we use it to reduce, attaching the resulting semantic value to the resulting nonterminal instance. For example, suppose that the semantic values attached to the three 'i's in the preceding example are leaf nodes 1, 2, and 3, respectively. Then, using $x : E$ to mean "semantic value E is attached to symbol x," we have the following parse

```
 1. i:1       + ( i:2 + i:3 ) $
 2. t:1       + ( i:2 + i:3 ) $
 3. e:1 + ( i:2       + i:3 ) $
 4. e:1 + ( t:2       + i:3 ) $
```

```
 5. e:1 + ( e:2 + i:3       ) $
 6. e:1 + ( e:2 + t:3       ) $
 7. e:1 + ( e:(+ 2 3) )       $
 8. e:1 + t:(+ 2 3)           $
 9. e:(+ 1 (+ 2 3)) $
10. p
```

Initially, only the terminal symbols have semantic values (as supplied by the lexer). Each reduction computes a new semantic value for the nonterminal symbol produced, as directed by the grammar.

With or without semantic actions, the process illustrated above is called "shift-reduce parsing." Each step consists either of *shifting* the lookahead symbol from the remaining input (right of the gap) to the top of the stack (left of the gap), or of *reducing* some symbols (0 or more) on top of the stack to a nonterminal according to one of the grammar productions (and performing any semantic actions). Each line in the examples above represents one reduction, plus some number of shifts. For example, line 5 represents the reduction of 't' to 'e', followed by the shift of '+' and 'i'.

In all these grammars, it is convenient to have the end-of-file symbol ('$') and the start symbol (in the examples, 'p') occur in exactly one production. This first production then has no important semantic action attached to it. This means that as soon as we shift the end-of-file symbol, we have effectively accepted the string and can stop.

2.2 Recognizing Possible Handles: the LR(0) machine

We can completely mechanize the process of shift-reduce parsing as long as we can determine when we have a handle on the stack, and which handle we have. The algorithm then becomes

```
while ($ not yet shifted) {
   if (handle is on top of the stack)
      reduce the handle;
   else
      shift the lookahead symbol;
}
```

It turns out, interestingly enough, that although context-free languages cannot be recognized in general by finite-state machines, their rightmost handles *can* be recognized. That is, we can build a DFA that allows us to perform the "handle is on top of the stack" test by pushing the stack through the DFA from bottom to top (left to right in the diagrams above). This DFA will also tell us which production to use to reduce the handle.

To do this, we will first show how to construct a "handle grammar"—a grammar that describes all possible handles. The terminal symbols of this grammar will be all the symbols (terminal and nonterminal) of the grammar we are trying to parse. We will then show how to convert the handle grammar into an NFA, after which the usual NFA-to-DFA construction will finish the job.

The nonterminals of the handle grammar for Grammar 2 are H_p, H_e, and H_t. H_p means "a handle that occurs during a rightmost derivation of a string from 'p' ". Likewise, H_e means "a handle that occurs during a rightmost derivation of a string from 'e', and so forth. Let's start with H_p. There are two cases: either the stack consists of the handle "e \$" and we are ready for the final reduction, or we are still in the process of forming the 'e' and haven't gotten around to shifting the '\$' yet—in other words, we have some handle that occurs during a derivation of some string from 'e'. Such a handle is supposed to be described by H_e. This gives us the rules:

$H_p \rightarrow$ e \$

$H_p \rightarrow H_e$

(Again, the symbols 'e' and '\$' are *both* terminal symbols here; H_p is the nonterminal.)

Now let's consider H_e. From the grammar, we see that one possible handle for 'e' is 't'. It is also possible that we are part way through the process of reducing to this 't', so that we have the two rules

$$H_e \rightarrow t$$
$$H_e \rightarrow H_t$$

Likewise, we also see that another possible handle is 'e + t'. It is therefore possible to have 'e +' on the stack, followed by a handle for an as-yet-incomplete 't', or finally, it is possible that the 'e' before the '+' is not yet complete. These considerations lead to the following rules for H_e:

$$H_e \rightarrow e' +' t$$
$$H_e \rightarrow e' +' H_t$$
$$H_e \rightarrow H_e$$

(The last rule is useless, but harmless).

Continuing, the full handle grammar looks like this:

$$H_p \rightarrow e\$ \mid H_e$$
$$H_e \rightarrow t \mid H_t$$
$$H_e \rightarrow e + t | e + H_t \mid H_e$$
$$H_t \rightarrow i$$
$$H_t \rightarrow (e) \mid (H_e$$

This grammar has a special property: the only place that a nonterminal symbol appears on a right-hand side is at the end (the nonterminals in the handle grammar are H_p, H_e, and H_t). This is significant because grammars with this property can be converted into NFAs very easily.

Consider, for example, the grammar

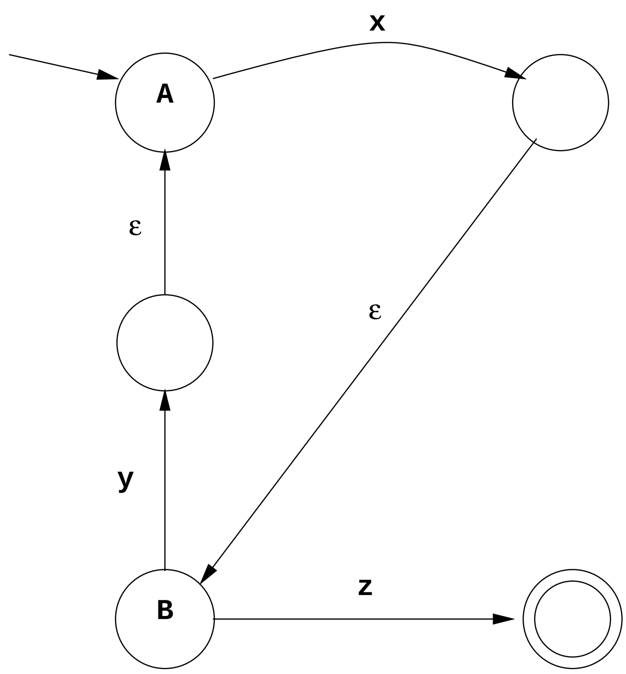

Figure 1: Example of converting a regular grammar into a NFA.

```
A -> x B
B -> y A
B -> z
```

The NFA in Fig. 1 recognizes this grammar. We simply translate each nonterminal into a state, and transfer to that state whenever a right-hand side calls for recognizing the corresponding nonterminal. The translation in the figure uses some epsilon transitions where they really could be avoided, because this will be convenient in the production of a machine for the handle grammar.

When we use this technique to convert the handle grammar into a NFA, we get the machine shown in Fig. 2. We have put labels in the states that hint at why they are present. For example, the state labeled "`e -> e .  + t`" is supposed to mean "the state of being part way through a handle for the production "`e -> e + t`" just before the '+'." These labels (productions with a dot in them) are known as *LR(0) items*. Some of the states have the same labels; however, if you examine them, you will see that any string that reaches one of them also reaches the other, so that the identical labels are appropriate. There are no final states mentioned, because all the information we'll need resides in the labels on the states.

The final step is to convert the NFA of Fig. 2 into a DFA (so that we can easily turn it into a program). We use the set-of-states construction that you learned previously. The labels on the resulting states are sets of LR(0) items; We leave out the labels H_p, H_e, and H_t, since they turn out to be redundant. You can verify that we get the machine shown in

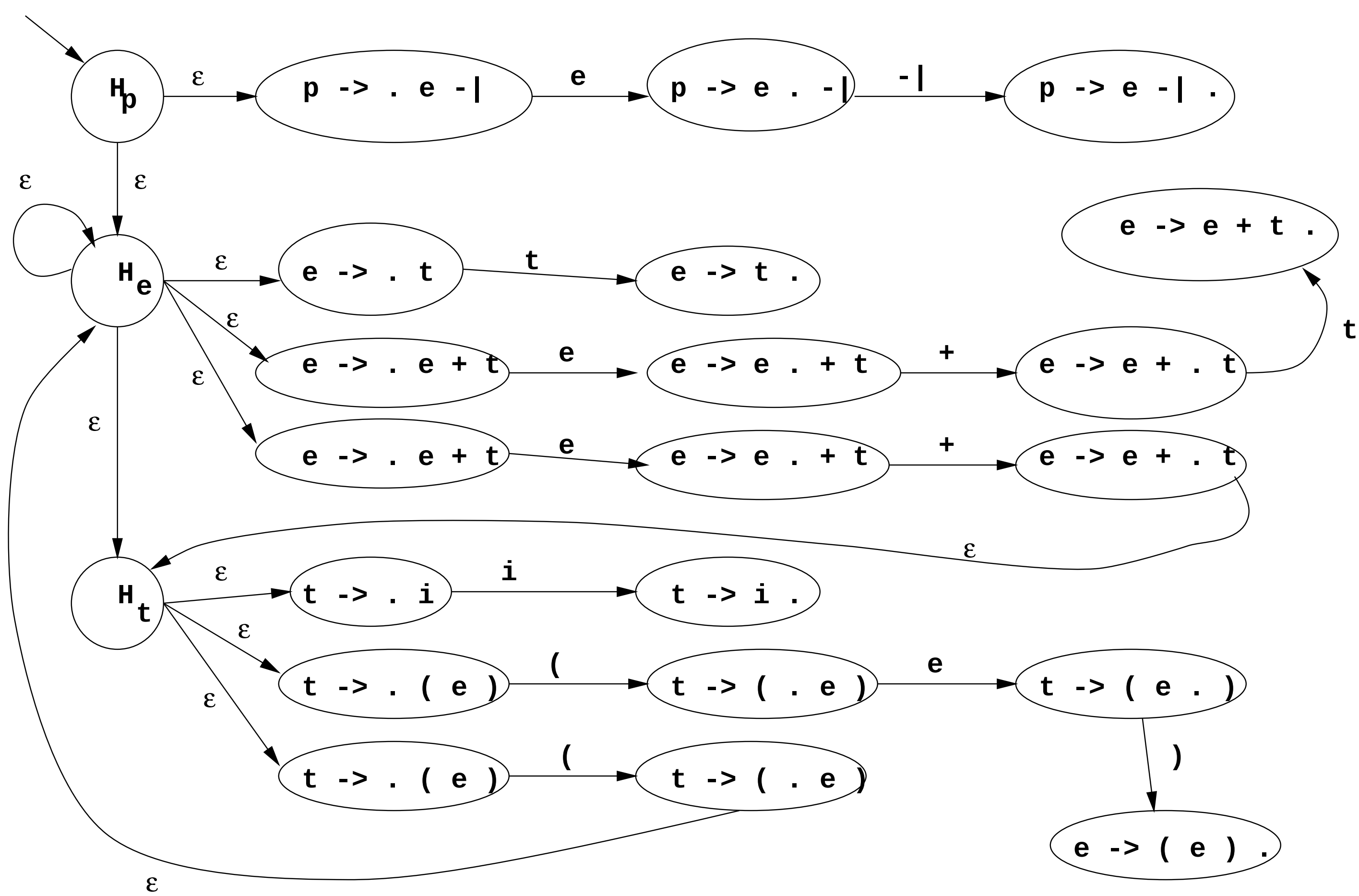

Figure 2: NFA from the handle grammar.

Fig. 3.

2.3 Using the Machine

We may represent the LR(0) machine from Fig. 3 as a state-transition table:

	Action					*Goto*	
State	`i`	`+`	`(`	`)`	$	`e`	`t`
0.	s2		s4			1	3
1.		s6			s5		
2.	rE	rE	rE	rE	rE		
3.	rC	rC	rC	rC	rC		
4.	s2		s4			7	3
5.			*ACCEPT*				
6.	s2		s4				8
7.		s6		s9			
8.	rB	rB	rB	rB	rB		
9.	rD	rD	rD	rD	rD		

The numeric entries in this table (preceded by 's' (shift) in the action table and appearing plain in the goto table) come from the state transitions (arcs) in Fig. 3. The 'r' (reduce) entries come from LR(0) items with a dot at the right (indicating a state of "being to the right of a potential handle.") The letters after 'r' refer to productions in Grammar 2.

To see how to use this table, consider again the string 'i+(i+i)$'. Initially, we have the situation

1. $_0$ `|` i + (i + i) $

Here, the '`|`' separates the stack on the left from the unprocessed input on the right; the lookahead symbol is right after '`|`'. The subscript '0' indicates that the DFA state that corresponds to the left end of the stack is state 0. We use the table, starting in state 0. There is nothing on the stack, and row 0 of the table tells us that there is no reduction possible, but that we could process an 'i' token if it were on the stack. Therefore, we shift the 'i' token from the input, giving

2. $_0i_2$ `|` + (i + i) $

(The subscript 2 shows the DFA's state after scanning the 'i' on the stack). Again, we start in state 0 and scan the stack, using the transitions in the table. This leaves us in state 2. Row 2 in the table tells us that no shifts are possible, but we may reduce ('r') using production E (`t -> i`). We therefore pop the 'i' off the stack, and push a 't' back on, giving

2. $_0t_3$ `|` + (i + i) $

Running the machine over this new stack lands us in state 3, which says that no shifts are possible, but we can use reduction C (`e -> t`), which gives

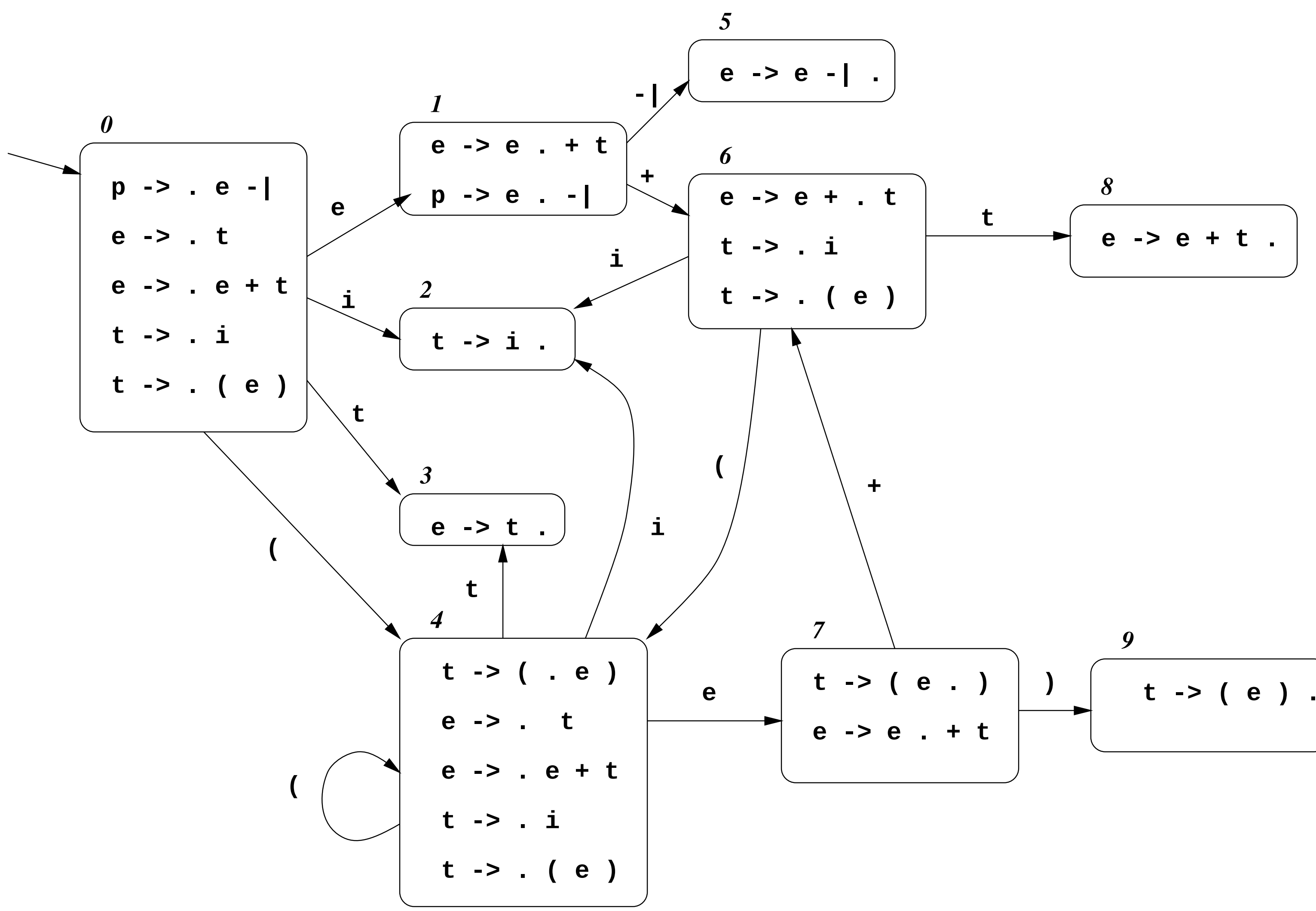

Figure 3: DFA constructed from Fig. 2: The canonical LR(0) machine.

3. ${}_0e_1 \quad | \ + \ (\ i \ + \ i \ ) \ \$$

Now the machine ends up in state 1, whose row tells us that either a '+' or an '$' could be next on the stack, so that we can shift either of these, leading to

3a. ${}_0e_1 +_6 \quad | \ (\ i \ + \ i \ ) \ \$$

The state 6 entry tells us that we can shift '(', and then the state 4 entry tells us we can shift 'i', giving

3b. ${}_0e_1 +_6 (_4 i_2 \quad | \ + \ i \ ) \ \$$

whereupon we see, again from the state 2 entry, that we should reduce using production E:

4. ${}_0\mathrm{e}_1 +_6 (_4 \mathrm{t}_3 \quad | \ + \ i \ ) \ \$$

and the state 3 entry tells us to reduce using production C:

5. ${}_0\mathrm{e}_1 +_6 (_4 \mathrm{e}_7 \quad | \ + \ i \ ) \ \$$

and so forth.

In general, then, we repeatedly perform the following steps for each shift and reduction the parser takes:

FINDSTATE: state = 0; for each symbol, s, on the stack, state = table[state][s];

FINDACTION:
if table[state][lookahead] is sn
push the lookahead symbol on the stack;
advance the input;
else if table[state][lookahead] is rk
Let $A \rightarrow x_1 \cdots x_m$ be production k;
pop m symbols from the stack;
push symbol A on the stack;
else if table[state][lookahead] is ACCEPT
end the parse;

The FINDACTION part of this fragment takes a constant amount of time for each action. However, the time required for FINDSTATE increases with the size of the stack. We can speed up the parsing process with a bit of "memoization". Rather than save the stack symbols, we instead save the *states* that scanning those symbols results in (the subscripts in my examples above). Each parsing step then looks like this:

FINDACTION:
if table[top(stack)][lookahead] is sn
push n on the stack;
advance the input;
else if table[top(stack)][lookahead] is rk

Let $A \rightarrow\ x_1 \cdots x_m$ be production k;
pop m states from the stack;
// Reminder: top(stack) is now changed!
push table[top(stack)][A] on the stack;
else if table[state][lookahead] is ACCEPT
end the parse;

and our sample parse looks like this:

```
 0. 0      |   i + ( i + i ) $
 1. 0 2    | + ( i + i ) $
 2. 0 3    | + ( i + i ) $
 3. 0 1    | + ( i + i ) $
3a. 0 1 6    | ( i + i ) $
3b. 0 1 6 4    | i + i ) $
3c. 0 1 6 4 2    | + i ) $
 4. 0 1 6 4 3    | + i ) $
 5. 0 1 6 4 7    | + i ) $
5a. 0 1 6 4 7 6    | i ) $
5b. 0 1 6 4 7 6 2    | ) $
 6. 0 1 6 4 7 6 8    | ) $
 7. 0 1 6 4 7        | ) $
7a. 0 1 6 4 7 9        | $
 8. 0 1 6 8            | $
 9. 0 1                | $
9a. 0 1 5                |
10. ACCEPT
```

It's important to see that all we have done with this change is to speed up the parse.

2.4 Resolving conflicts

Grammar 2 is called an *LR(0) grammar*, meaning that its LR(0) machine has the property that each state contains either no reduction items (items with a dot at the far right) or exactly one reduction item and nothing else. In other words, an LR(0) grammar is one that can be parsed from *L*eft to right to produce a *R*ightmost derivation using a shift-reduce parser that does not consult the lookahead character (uses *0* symbols of lookahead). Few grammars are so simple. Consider, for example,

Grammar 3.

```
A. s -> e $
B. e -> t '@' e
C. e -> t
```

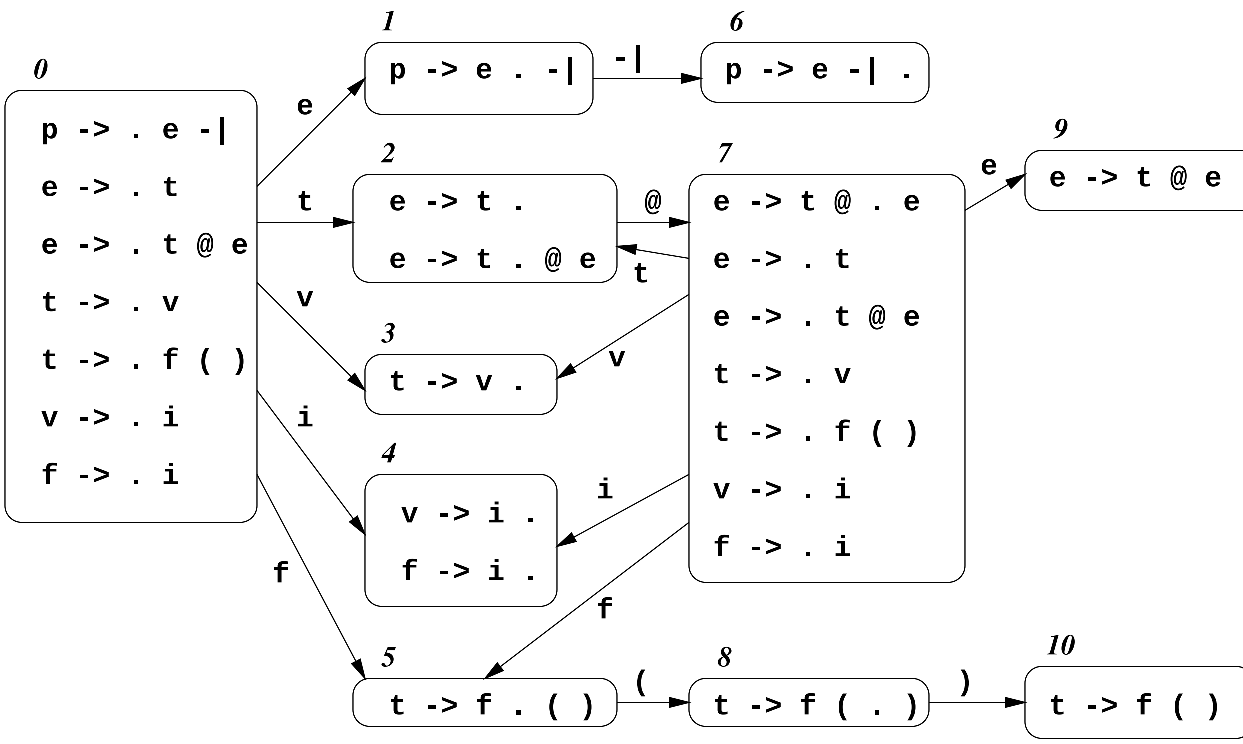

Figure 4: LR(0) machine for Grammar 3.

```
D. t -> f '(' ')'
E. t -> v
F. f -> i
G. v -> i
```

which gives us the DFA in Fig. 4.

As you can see from the figure, there are problems in states #2 and #4. State #2 has an *LR(0) shift/reduce conflict:* it is possible *both* to reduce by reduction C *or* to shift the symbol '@'. In this particular case, it turns out that the correct thing to do is to shift when the lookahead symbol is '@' and to reduce otherwise; that is, reducing on '@' will always cause the parse to fail later on. State #4 has an *LR(0) reduce/reduce conflict:* it is possible to reduce either by reduction F or G. In this case, the correct thing to do is to reduce using F if the next input symbol is '(' and by G otherwise. We end up with the following parsing table:

State	Action					Goto			
	`i`	`@`	`(`	`)`	$	`e`	`t`	`f`	`v`
0.	s4					1	2	5	3
1.					s6				
2.	rC	s7	rC	rC	rC				
3.	rE	rE	rE	rE	rE				
4.	rG	rG	rF	rG	rG				
5.			s8						
6.			ACCEPT						
7.	s4					9	2	5	3
8.				s10					
9.	rB	rB	rB	rB	rB				
10.	rD	rD	rD	rD	rD				

Because the choice between reduction and shift, or between two reductions, depends on the lookahead symbol (in contrast to Grammar 2), we say Grammar 3 is not LR(0). However, since one symbol of lookahead suffices, we say that it is *LR(1)*—parseable from *L*eft to right producing a *R*ightmost derivation using a shift-reduce parser with *1* symbol of lookahead. In fact, Grammar 3 is what we call *LALR(1)*, the subclass of *LR(1)* for which the parsing table has the same states and columns as for the LR(0) machine, and we merely have to choose the entries properly to get the desired result. (LALR means "Lookahead LR." Since LR parsers *do* look ahead anyway, it's a terrible name, but we're stuck with it.) Yacc and Bison produce LALR(1) parsers. The class LR(1) is bigger, but few practical grammars are LR(1) without being LALR(1), and LALR(1) parsing tables are considerably smaller.

Unfortunately, it is not clear from just looking at the machine that we have filled in the problematic entries correctly. In particular, while the choice between reductions F and G in state #4 is clear in this case, the general rule is not at all obvious. As for the LR(0) shift-reduce conflict in state #2, it is obvious that if '@' is the lookahead symbol, then shifting *has* to be acceptable, but perhaps this is because the grammar is ambiguous and *either* the shift or the reduction could work, or perhaps if we looked two symbols ahead instead of just one, we would sometimes choose the reduction rather than the shift.

One systematic approach is to use the FOLLOW sets that we used in LL(1) parsing. Faced with an LR(0) reduce/reduce conflict such as '`f` → `i`' vs. '`v` → `i`' in state #4, we choose to reduce to `f` if the lookahead symbol is in FOLLOW(`f`), choose to reduce `v` if the lookahead symbol is in FOLLOW(`v`), and choose either one otherwise (or leave the entry blank). Likewise, we can assure that the LR(0) shift-reduce conflict in state #2 is properly resolved in favor of shifting '@' as long as '@' does not appear in FOLLOW(`e`), as in fact it doesn't. When this simple method resolves all conflicts and tells us how to fill in the LR(0) conflicts in the table, we say that the grammar is *SLR(1)* (the 'S' is for "Simple"). Grammar 3 happens to be SLR(1).

However, there are cases where the FOLLOW sets fail to resolve the conflict because they are not sensitive to the context in which the reduction takes place. Therefore, typical shift-reduce parser generators go a step further and use the full LALR(1) lookahead computation.

This works similarly to the FOLLOW computation described in the textbook (pg. 189 of *ASU*). We attach a set of *lookahead symbols* to the end of each LR(0) item, giving what is called an LR(1) item. Think of an LR(1) item such as

```
t -> . v,   $, @
```

as meaning "we could be at the left end of a handle for `t`, and after that handle, we expect to see either an '\$' or a '@'." We add these lookaheads to the LR(0) machine in Fig. 4 by applying the following two operations repeatedly until nothing changes, starting with empty lookahead sets:

- If we see an item of the form '$A \to \ \alpha.B\beta, L_1$' in a state (where L is a set of lookaheads, B is a nonterminal, and α and β are sequences of 0 or more terminal and nonterminal symbols), then for every other item in that same state of the form '$B \to \ .\gamma, L_2$,' add the set of terminal symbols FIRST(βL_1) to the set L_2. (Here, we define FIRST(L_1) to be simply L_1. Therefore, FIRST(βL_1) is simply FIRST(β) if β does not produce the empty string, and otherwise it is FIRST(β) $\cup\, L_1 - \epsilon$).

- If we see an item of the form '$A \to \ \alpha.X\beta, L_1$' in a state, then find the transition from that state on symbol X and find item '$A \to \ \alpha X.\beta, L_2$' in the target of that transition. Add the symbols in L_1 to L_2.

Applying these operations to the machine in Fig. 4 gives the LALR(1) machine in Fig. 5.

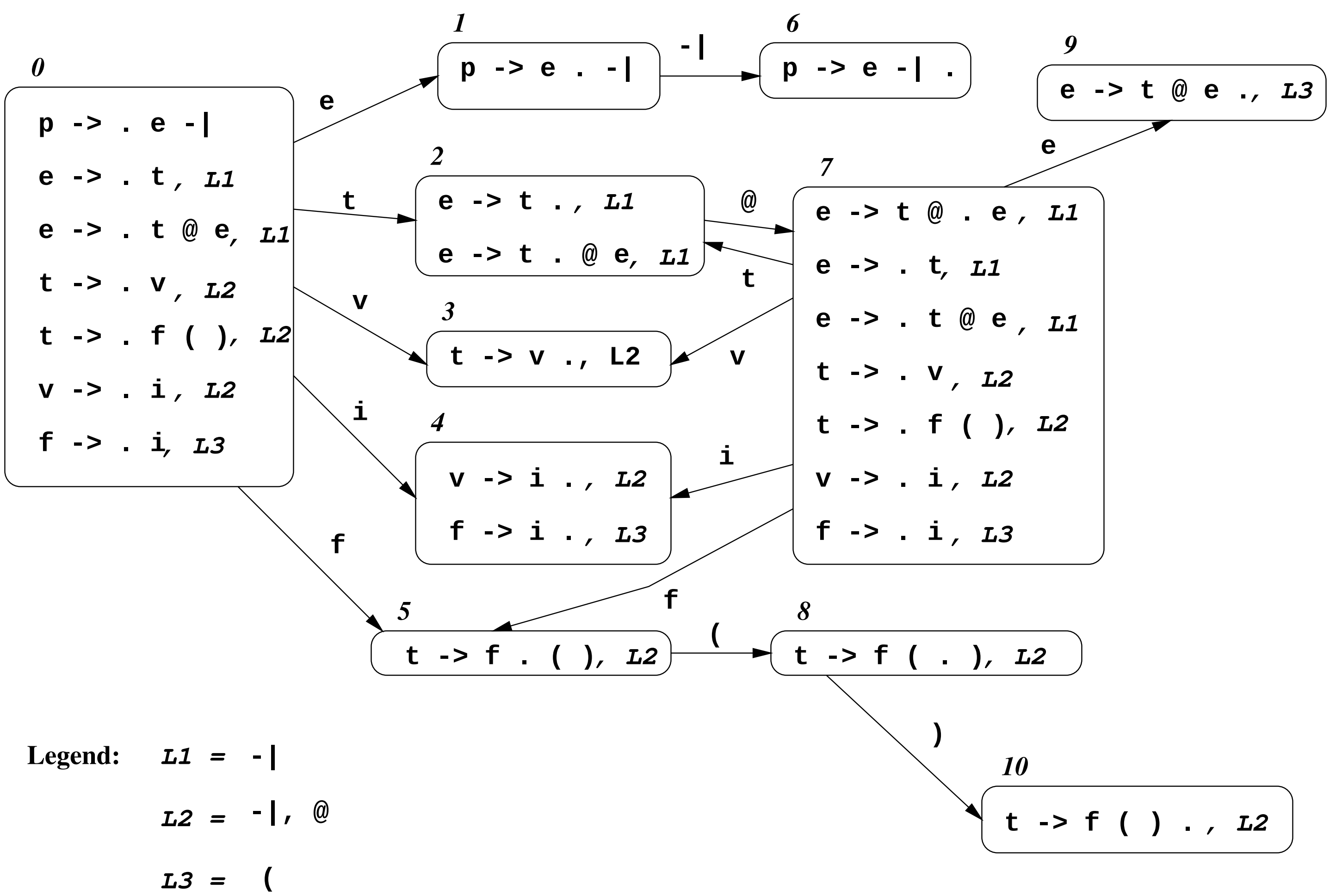

Figure 5: LALR(1) machine for Grammar 3.