CS143 Handout #7
Autumn 2000

Notes on top-down parsing

Handout written by Maggie Johnson and revised by me.

Approaches to parsing

The syntax analysis phase of a compiler verifies that the sequence of tokens passed in from the scanner represent valid sentences as defined by the grammar for the programming language. There are two major parsing approaches: *top-down* and *bottom-up*. In top-down parsing, you start with the start symbol and apply the productions until you arrive at the desired string. In bottom-up parsing, you start with the string and reduce it to the start symbol by applying the productions backwards. As an example, let's trace through the two approaches on this simple grammar that recognizes strings consisting of any number of a's followed by at least one (and possibly more)b's:

S	←	AB
A	←	a | aA | ∂
B	←	b | bB

A top-down parse of aaab choosing to expand the leftmost non-terminal at each step:

S	
AB	S ←AB
aAB	A ←aA
aaAB	A ←aA
aaaAB	A ←aA
aaa∂B	A ←∂
aaab	B ←b

A bottom-up parse of the same string is done by trying to find right-side productions that match parts of the working string. We end up applying productions in the reverse order of a top-down parse:

aaab	
aaaB	B ←b
aaa∂B	(insert ∂)
aaaAB	A ←∂
aaAB	A ←aA
aAB	A ←aA
AB	A ←aA
S	S ←AB

In a bottom-up parse, you know you are done when you have substituted all the way back to the start symbol.

In creating a parser for a compiler, we normally have to place some restrictions on how we process the input. In the above example, it was easy for us to see which productions were appropriate because we saw the entire string aaab. In a compiler's parser, however, we do not have such long-distance vision. We are usually limited to just one-symbol of *lookahead*. The lookahead symbol is the next symbol coming up in the input. This restriction certainly makes the parsing more challenging. Using the same grammar from above, if the parser sees only a single a in the input and it cannot lookahead any further than the symbol we are on, it can't know whether to use the production A ←a or A ←aA.

One solution to this problem would be to implement *backtracking*. Based on the information the parser currently has about the input, a decision is made to go with one particular production. If this choice leads to a dead end, the parser would have to backtrack to that decision point, moving backwards through the input, and start again making a different choice and so on until it either found the production that was the appropriate one or ran out of choices. A number of authors have described backtracking parsers but they are rarely used in practice because they are so slow. We will look at other ways to fix this problem as we proceed.

Top-down predictive parsing

Today, we will focus in on top-down parsing. We will look at a couple different ways to implement a non-backtracking top-down parser called a *predictive parser*. A predictive parser is characterized by its ability to make a decision on which production to apply, based solely on the next input symbol and the current non-terminal being processed. To do this, the grammar that drives the parser must have a particular form. We call such a grammar *LL(1)*. The first "L" means we scan the input from left to right; the second "L" means we create a leftmost derivation; and the 1 means one input symbol of lookahead. Informally, an LL(1) has no left-recursive productions and has been left-factored. Note that these are necessary conditions for LL(1) but not sufficient, i.e., there exist grammars with no left-recursion or common prefixes that are not LL(1). Note also that there exist many grammars that cannot be modified to become LL(1). In such cases, another parsing technique must be employed, or special rules must be embedded into the predictive parser.

Recursive-descent

The first technique for implementing a predictive parser is called *recursive-descent*. A recursive-descent parser consists of several small C functions, one for each non-terminal in the grammar. As we parse a sentence, we call the functions that correspond to the left side non-terminal of the productions we are applying. If these productions are recursive, we end up calling the functions recursively.

Let's start by examining some productions from a grammar for a simple Pascal-like programming language. In this programming language, all functions are preceded by the reserved word FUNC:

```
program          ←  function_list
function_list    ←  function *
function         ←  FUNC identifier ( parameter_list ) statement
```

What might the C function that is responsible for parsing a function definition look like? It expects to first find the token FUNC, then it expects an identifier (the name of the function), followed by an opening parenthesis, and so on. At it pulls each token from the parser, it must ensure that it matches the expected, and if not, will halt with an error. For the non-terminals, this function calls their associated function to handle its part of the parsing.

```
void ParseFunction()
{
   if (lookahead != T_FUNC) { // anything not FUNC here is wrong
      printf("syntax error \n");
      exit(0);
   } else
      lookahead = yylex();   // global variable holds next token
   ParseIdentifier();
   if (lookahead != T_LPAREN) {
      printf("syntax error \n");
      exit(0);
   } else
      lookahead = yylex();
   ParseParameterList();
```

```
   if (lookahead!= T_RPAREN) {
      printf("syntax error \n");
      exit(0);
   } else
      lookahead = yylex();
   ParseStatement();
}
```

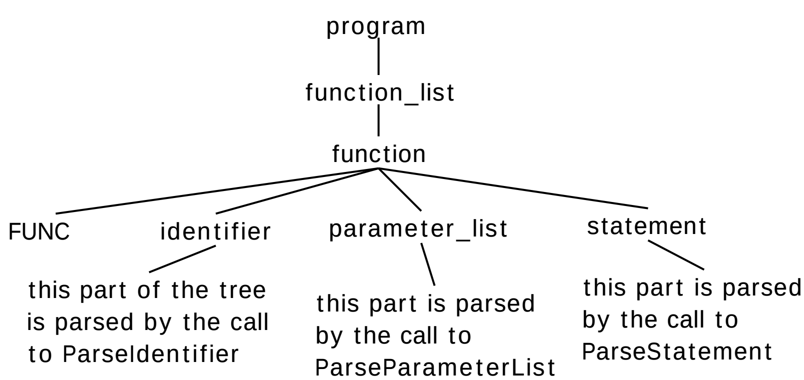
To make things a little cleaner, let's introduce a utility function that can be used to verify that the next token is what is expected and will error and exit otherwise. We will need this again and again in writing the parsing routines.

```
void MatchToken(int expected)
{
   if (lookahead != expected) {
      printf("syntax error \n");
      exit(0);
   } else      // if match, consume token and move on
      lookahead = yylex();
}
```

Now we can tidy up the `ParseFunction()` routine and make it clearer what it does:

```
void ParseFunction()
{
   MatchToken(T_FUNC);
   ParseIdentifier();
   MatchToken(T_LPAREN);
   ParseParameterList();
   MatchToken(T_RPAREN);
   ParseStatement();
}
```

The following diagram illustrates how the parse tree is built:



Here is the production for an if-statement in a somewhat Pascal-like language. In this mythical language, a closing ENDIF delimiter is used to avoid ambiguity about the dangling ELSE:

```
if_statement   ←   IF expression THEN statement ENDIF |
                   IF expression THEN statement ELSE statement ENDIF
```

To prepare this for recursive-descent, we must left-factor this production to shared the common parts in one production:

```
if_statement    ←   IF expression THEN statement close_if
close_if        ←   ENDIF | ELSE statement ENDIF
```

Now, let's examine the recursive-descent function that will parse an if-statement:

```
void ParseIfStatement()
{
   MatchToken(T_IF);
   ParseExpression();
   MatchToken(T_THEN);
   ParseStatement();
   ParseCloseIf();
}

void ParseCloseIf()
{
   if (lookahead == T_ENDIF)  // if we immediately find ENDIF
      lookahead = yylex();    // predict close_if -> ENDIF
   else {
      MatchToken(T_ELSE);     // otherwise we look for ELSE
      ParseStatement();       // predict close_if -> ELSE stmt ENDIF
      MatchToken(T_ENDIF);
   }
}
```

When trying to parse the closing portion of the if, we have to make a decision about which of the two right-hand side options we are expanding. In this case, it isn't too difficult. We try to match the first token again `ENDIF` and on non-match, we try to match the `ELSE` clause and if that doesn't match, it will report an error.

Navigating through two choices seemed simple enough, however, what happens where we have a lot of alternatives on the right-side?

```
statement    ← assg_statement | return_statement | print_statement | null_statement
               | if_statement | while_statement | block_of_statements
```

When implementing the `ParseStatement` function, how are we going to be able to determine which of the seven options to match for any given input? Remember, we are trying to do this without backtracking, and just one token of lookahead, so we have to be able to make quick decision with minimal information— this can be difficult!

To understand how to recognize and solve problem, we need a definition:

> The *first set* of a sequence of symbols $\underline{u}$, written as $\texttt{First}(\underline{u})$ is the set of terminals which start all the sequences of symbols derivable from $\underline{u}$. A bit more formally, consider all strings derivable from $\underline{u}$ by a leftmost derivation. If $\underline{u} \Rightarrow^* \underline{v}$, where $\underline{v}$ begins with some terminal, that terminal is in $\texttt{First}(\underline{u})$. If $\underline{u} \Rightarrow^* \partial$, then ∂ is in $\texttt{First}(\underline{u})$.

Informally, the first set of a sequence just a list of all the possible terminals that could start a string derived from that sequence. We will do an example of how to calculate first sets a bit later. For now, just keep in mind its intuitive meaning. Computing these sets for each of the right-hand sides of the available productions and matching our one token lookahead to the one of those sets determines the path to follow to find the valid parse for that non-terminal.

Given a production with a number of alternatives: A $\leftarrow \underline{u}_1$ | $\underline{u}_2$ | …, we can write a recursive-descent routine only if all the sets First($\underline{u}_i$) are disjoint. The general form of such a routine would be:

```
void ParseA()
{
   switch (lookahead)
   {
      case First(u1):         // predict production A -> u1
         /* code to recognize u1 */
         return;
      case First(u2):         // predict production A -> u2
         /* code to recognize u2 */
         return;
      etc....

      default:
         printf("syntax error \n");
         exit(0);
   }
}
```

The case notation above is not quite legal C, but we are trying to just abstractly indicate that the each case handles one production and thus lists of all symbols in the first set for the right-hand side of that production.

If first sets of the various productions are not disjoint, it's easy to see that a predictive parser will not know what to do. We would either need to re-write the language grammar or use a different parsing technique for this non-terminal in such a case. For programming languages, it is usually possible to re-structure the productions or embed certain rules into the parser to resolve conflicts, but this constraint is one of the weaknesses of the top-down non-backtracking approach.

A further complication arises if the non-terminal we are trying to recognize is *nullable*. A non-terminal A is nullable if there is a derivation of A that results in just the symbol ∂ (i.e. that non-terminal would completely disappear in the parse string) or, more precisely, ∂ First(A). In this case A could be replaced by nothing and the next token would be the first token of the symbol following A in the sentence being parsed. Thus if A is nullable, our predictive parser also needs to consider the possibility that the path to choose is the one corresponding to A =>* ∂. To deal with this we define the following:

> The *follow set* of a non-terminal A is the set of terminal symbols that can appear immediately to the right of A in a valid sentence. A bit more formally, for every valid sentence S =>* $\underline{u}$A$\underline{v}$, where $\underline{v}$ begins with some terminal, that terminal is in Follow(A).

Informally, you can think about the follow set like this: A can appear in various places within a valid sentence. The follow set describes what terminals could have followed the sentential form that was expanded from A. We will detail how to calculate the follow set a bit later. For now, realize follow sets are useful because they define the right context consistent with a given non-terminal and provide the lookahead that might signal a nullable non-terminal should be expanded to ∂.

With these two definitions, we can now generalize how to handle `A` ← $\underline{u}_1$ | $\underline{u}_2$ | ..., in a recursive-descent parser. In all situations, we need a case to handle each member in `First(`$\underline{u}_i$`)`. In addition if there is a derivation from any $\underline{u}_i$ that could yield ∂ (i.e. if it is nullable) then we also need to handle each of the members in `Follow(A)`.

```
void ParseA()
{
   switch (lookahead)
   {
      case First(u1):
         /* code to recognize u1 */
         return;
      case First(u2):
         /* code to recognize u2 */
         return;
      case Follow(A): // predict production A->epsilon
         /* usually do nothing here */
      ...
      default:
         printf("syntax error \n");
         exit(0);
   }
}
```

What about left-recursive productions? Now we see why these are such a problem in a predictive parser. Consider this left-recursive production that matches a list of one or more functions.

```
function_list    ←   function_list function | function
function         ←   FUNC identifier ( parameter_list ) statement
```

```
void ParseFunctionList()
{
   ParseFunctionList();
   ...
}
```

Such a production will send a recursive-descent parser into an infinite loop! We need to remove the left-recursion in order to be able to write the parsing function for a function_list.

```
function_list    ←   function_list function | function
```

becomes

```
function_list    ←   function function_list | function
```

then we must left-factor the common parts

```
function_list    ←   function more_functions
more_functions   ←   function more_functions | ∂
```

And now the parsing function looks like this:

```
void ParseFunctionList()
{
   ParseFunction();
   ParseMoreFunctions(); // may find no more fns (i.e. expand to
epsilon)
}
```

Computing first and follow

For a grammar to be suitable for LL(1) parsing, you must first remove ambiguity and left-factor and eliminate left-recursion from all productions. Now, follow these steps to compute first and follow:

Calculating first sets. To calculate First($\underline{u}$) where $\underline{u}$ has the form $X_1X_2...X_n$, do the following:

1. If X_1 is a terminal, then add X_1 to First($\underline{u}$), otherwise add First(X_1) - ∂ to First($\underline{u}$).
2. If X_1 is a nullable non-terminal, i.e., X_1 =>* ∂, add First(X_2) to First($\underline{u}$). Furthermore, if X_2 can also go to ∂, then add First(X_3) and so on, through all X_n until the first non-nullable one.
3. If $X_1X_2...X_n$ =>* ∂, add ∂ to the first set.

Calculating follow sets. For each non-terminal in the grammar, do the following:

1. Place EOF in Follow(S) where S is the start symbol and EOF is the input's right endmarker. The endmarker might be end of file, it might be newline, it might be a special symbol, whatever is the expected end of input indication for this grammar. We will typically use $ as the endmarker.
2. For every production A ←$\underline{u}$B$\underline{v}$ where $\underline{u}$ and $\underline{v}$ are any string of grammar symbols and B is a non-terminal, everything in First($\underline{v}$) except ∂ is placed in Follow(B).
3. For every production A ←$\underline{u}$B, or a production A ←$\underline{u}$B$\underline{v}$ where First($\underline{v}$) contains ∂ (i.e. $\underline{v}$ is nullable), then everything in Follow(A) is added to Follow(B).

Here is a complete example of first and follow set computation, starting with this grammar:

```
S    ←AB
A    ←Ca  | ∂
B    ←BaAC  | c
C    ←b | ∂
```

Notice we have a left-recursive production that must be fixed if we are to use LL(1) parsing:

```
B    ←BaAC  | c
```

becomes

```
B    ←cB'
B'   ←aACB'  | ∂
```

The new grammar is:

```
S    ←AB
A    ←Ca  | ∂
B    ←cB'
B'   ←aACB'  | ∂
C    ←b | ∂
```

It helps to first compute the nullable set (i.e. those non-terminals X that X =>* ∂), since you need to refer to the nullable status of various non-terminals when computing the first and follow sets:

```
Nullable(G) =  {A  B' C}
```

The first sets for each non-terminal are:

```
First(C) = {b ∂}
First(B') = {a ∂}
```

First(B) = {c}
First(A) = {b a ∂}

Start with First(C) - ∂, add a (since C is nullable) and ∂ (since A itself is nullable)

First(S) = {b a c}

Start with First(A) - ∂, add First(B) (since A is nullable). We don't add ∂ (since S itself is not-nullable— A can go away, but B cannot)

It is usually convenient to compute the first sets for the non-terminals that appear toward the bottom of the parse tree and work your way upward since the non-terminals toward the top may need to incorporate the first sets of the terminals that appears beneath them in the tree.

To compute the follow sets, take each non-terminal and go through all the right-side productions that the non-terminal is in, matching to the steps given earlier:

Follow(S) = {$}

S doesn't appear in the right hand side of any productions. We put $ in the follow set because S is the start symbol.

Follow(B) = {$}

B appears on the right hand side of the S ← AB production. Its follow set is the same as S.

Follow(B') = {$}

B' appears on the right hand side of two productions. The B' ← aACB' production tells us its follow set includes the follow set of B', which is tautological. From B ← cB', we learn its follow set is the same as B.

Follow(C) = {a $}

C appears in the right hand side of two productions. The production A ← Ca tells us a is in the follow set. From B' ← aACB', we add the First(B') which is just a again. Because B' is nullable, we must also add Follow(B') which is $.

Follow(A) = {c b a $}

A appears in the right hand side of two productions. From S ← AB we add First(B) which is just c. B is not nullable. From B' ← aACB', we add First(C) which is b. Since C is nullable, so we also include First(B') which is a. B' is also nullable, so we include Follow(B') which adds $.

It can be convenient to compute the follows sets for the non-terminals that appear toward the top of the parse tree and work your way down, but sometimes you have to circle around computing the follow sets of other non-terminals in order to complete the one you're on.

The calculation of the first and follow sets follow mechanical algorithms, but it is very easy to get tripped up in the details and make mistakes even when you know the rules. Be careful!

Table-driven LL(1) parsing

In a recursive-descent parser, the production information is embedded in the individual parse functions for each non-terminal and the run-time execution stack is keeping track of our progress through the parse. There is another method for implementing a predictive parser that uses a table to store that production along with an explicit stack to keep track of where we are in the parse.

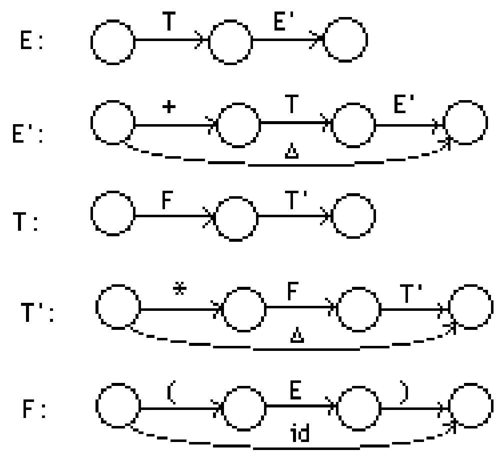
This grammar for add/multiply expressions is already set up to handle precedence and associativity:

```
E    ←  E + T | T
T    ←  T * F | F
F    ←  (E) | int
```

After removal of left recursion, we get:

```
E     ←  TE'
E'    ←  + TE' | ∂
T     ←  FT'
T'    ←  * FT' | ∂
F     ←  (E) | int
```

One way to illustrate the process is to study some transition graphs that represent the grammar:



A predictive parser behaves as follows. Let's assume the input string is `3 + 4 * 5`. Parsing begins in the start state of the symbol `E` and moves to the next state. This transition is marked with a `T`, which sends us to the start state for `T`. This in turn, sends us to the start state for `F`. `F` has only terminals, so we read a token from the input string. It must either be an open parenthesis or an integer in order for this parse to be valid. We consume the integer token, and thus we have hit a final state in the `F` transition diagram, so we return to where we came from which is the `T` diagram; we have just finished processing the `F` non-terminal. We continue with `T'`, and go to that start state. The current lookahead is + which doesn't match the * required by the first production, but + is in the follow set for `T'` so we match the second production which allows `T'` to disappear entirely. We finish `T'` and return to `T`, where we are also in a final state. We return to the `E` diagram where we have just finished processing the `T`. We move on to `E'`, etc...

A table-driven predictive parser uses a stack to store the productions to which it must return. A parsing table stores the actions the parser should take based on the input token and what value is on top of the stack. `$` is the end of input symbol.

Input / Top of parse stack	int	+	*	(	)	$
E	E ← TE'			E ← TE'		
E'		E' ← +TE'			E' ← ∂	E' ← ∂
T	T ← FT'			T ← FT'		
T'		T' ← ∂	T' ← *FT'		T' ← ∂	T' ← ∂
F	F ← int			F ← (E)		

Tracing a table-driven predictive parser

Here is how a predictive parser works. We push the start symbol on the stack and read the first input token. As the parser works through the input, there are the following possibilities for the top stack symbol `X` and the input token `a`:

1. If `X = a` and `a` = end of input (`$`): parser halts and parse completed successfully
2. If `X = a` and `a != $`: successful match, pop `X` and advance to next input token. This is called a *match* action.
3. If `X != a` and `X` is a non-terminal, pop `X` and consult table at [`X`,`a`] to see which production applies, push right side of production on stack. This is called a *predict* action.
4. If none of the preceding cases applies or the table entry from step 3 is blank, there has been a parse error

Here is an example parse of the string `int + int * int`:

PARSE STACK	REMAINING INPUT	PARSER ACTION
E$	int + int * int$	Predict E ← TE', pop E from stack, push TE', no change in input
TE'$	int + int * int$	Predict T ← FT'
FT'E'$	int + int * int$	Predict F ← int
intT'E'$	int + int * int$	Match int, pop from stack, move ahead in input
T'E'$	+ int * int$	Predict T' ← ε
E'$	+ int * int$	Predict E' ← +TE'
+TE'$	+ int * int$	Match +, pop
TE'$	int * int$	Predict T ← FT'
FT'E'$	int * int$	Predict F ← int
intT'E'$	int * int$	Match int, pop
T'E'$	* int$	Predict T' ← *FT'
*FT'E'$	* int$	Match *, pop
FT'E'$	int$	Predict F ← int
intT'E'$	int$	Match int, pop
T'E'$	$	Predict T' ← ε
E'$	$	Predict E' ← ε
$	$	Match $, pop, success!

Suppose, instead, that we were trying to parse the input `+$`. The first step of the parse would give an error because there is no entry at M[`E, +`].

Constructing the parse table

The next task is to figure out how we constructed the table. The construction of the table is somewhat involved and tedious (the perfect task for a computer, but error-prone for humans). The first thing we need to do is compute the first and follow sets for the grammar:

```
E     ← TE'
E'    ← + TE' | ∂
T     ← FT'
T'    ← * FT' | ∂
F     ← (E) | int

First(E) = First(T) = First(F) = { ( int }
First(T') = { * ∂ }
First(E') = { + ∂ }
Follow(E) = Follow(E') { $ ) }
Follow(T) = Follow(T') = { + $ ) }
Follow(F) = { * + $ ) }
```

Once we have the first and follow sets, we build a table M with the leftmost column labeled with all the non-terminals in the grammar, and the top row labeled with all the terminals in the grammar, along with $. The following algorithm fills in the table cells:

1. For each production A ← $\underline{u}$ of the grammar, do steps 2 and 3
2. For each terminal a in First($\underline{u}$), add A ← $\underline{u}$ to M[A,a]
3. If ∂ in First($\underline{u}$), (i.e. A is nullable) add A ← $\underline{u}$ to M[A,b] for each terminal b in Follow(A), If ∂ in First($\underline{u}$), and $ is in Follow(A), add A ← $\underline{u}$ to M[A,$]
4. All undefined entries are errors

The concept used here is to consider a production A ← $\underline{u}$ with a in First($\underline{u}$). The parser should expand A to $\underline{u}$ when the current input symbol is a. It's a little trickier when $\underline{u}$ = ∂ or $\underline{u}$ =>* ∂. In this case, we should expand A to $\underline{u}$ if the current input symbol is in Follow(A), or if the $ at the end of the input has been reached, and $ is in Follow(A).

If the procedure ever tries to fill in an entry of the table that already has a non-error entry, the procedure fails— the grammar is not LL(1).

Properties of LL(1) grammars

These predictive top-down techniques (either recursive-descent or table-driven) require a grammar that is LL(1), Probably the easiest fully general way to determine if a grammar is LL(1) is to build the table and see if you have conflicts, although in some cases you may be able to determine that a grammar is or isn't LL(1) via a shortcut. However, to give a formal of what is required:

- No ambiguity
- No left recursion
- A grammar G is LL(1) iff whenever A ← $\underline{u}$ | $\underline{v}$ are two distinct productions of G, the following conditions hold:
  - for no terminal a do both $\underline{u}$ and $\underline{v}$ derive strings beginning with a (i.e. first sets are disjoint)
  - at most one of $\underline{u}$ and $\underline{v}$ can derive the empty string
  - if $\underline{v}$ =>* ∂ then $\underline{v}$ does not derive any string beginning with a terminal in Follow(A)

All of this translates intuitively that when trying to recognize A, the parser must be able to examine just one input symbol of lookahead and from that uniquely determine which production to use.

Error-reporting and recovery

A few general principles apply to errors found regardless of parsing technique being used:

- A parser should try to determine that an error has occurred as soon as possible. Waiting too long before declaring an error can cause the parser to lose the actual location of the error.
- A suitable and comprehensive message should be reported. “Missing semicolon on line 36” is helpful, “unable to shift in state 425” is not.
- After an error has occurred, the parser must pick a likely place to resume the parse. Rather than giving up at the first problem, a parser should always try to parse as much of the code as possible in order to find as many real errors as possible during a single run.
- A parser should avoid *cascading errors*, which is when one error generates a lengthy sequence of spurious error messages.

Recognizing the input is not syntactically valid can be relatively straightforward. An error is detected in predictive parsing when the terminal on top of the stack does not match the next input symbol or when non-terminal `A` is on top of the stack, a is the next input symbol and the parsing table entry M[A,a] is empty.

Deciding how to handle the error is bit more complicated. By inserting specific error actions into the empty slots of the table, you can determine how a predictive parser will handle a given error condition. At the least, you can provide a precise error message that describes the mismatch between expected and found.

Recovering from errors and being able to resume and successfully parse is more difficult. The entire compilation could be aborted on the first error, but most users would like to find out more than one error per compilation. The problem is how to fix the error in some way to allow parsing to continue.

Many errors are relatively minor and involve syntactic violations for which the parser has a correction that it believes is likely to be what the programmer intended. For example, a missing semicolon at the end of the line or a misspelled keyword can usually be recognized. For many minor errors, the parser can “fix” the program by guessing at what was intended and reporting a warning, but allowing compilation to proceed unhindered. The parser might skip what appears to be an erroneous token in the input or insert a necessary, but missing, token or change a token into the one expected (substituting BEGIN for BGEIN). For more major or complex errors, the parser may have no reliable correction. The parser will attempt to continue but will probably have to skip over part of the input or take some other exceptional action to do so.

Panic-mode error recovery is a simple technique that just bails out of the current construct, looking for a safe symbol at which to restart parsing. The parser just discards input tokens until it finds what is called a *synchronizing* token. The set of synchronizing tokens are those that we believe confirm the end of the invalid statement and allow us to pick up at the next piece of code. For a non-terminal `A`, we could place all the symbols in `Follow(A)` into its synchronizing set. If `A` is the non-terminal for a variable declaration and the garbled input is something like `duoble d;` the parser might skip ahead to the semi-colon and act as though the declaration didn’t exist. This will surely cause some more cascading errors when the variable is later used, but it might get through the trouble spot. We could also use the symbols in `First(A)` as a synchronizing set for re-starting the parse of `A`. This would allow input `junk double d;` to parse as a valid variable declaration.

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.

D. Cohen, Introduction to Computer Theory. New York: Wiley, 1986.

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

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