5 Parsing methods 1 - Top down

Here we are going to consider three methods of parsing. First we take the commonest top down approach, recursive descent, then a simple bottom up approach, operator precedence parsing, and finally LR parsing. The intention is to introduce you to the commonest approaches to parsing, not to present an exhaustive survey. Consult Bornat or Aho, Sethi and Ullman for a more complete survey.

Recursive descent

Recursive descent compilers are popular with people wishing to construct a compiler quickly and by hand. They are very intuitive in their approach.

To write a successful recursive descent parser, any language description must be turned into the correct form. In general, recursive descent parsers are predictive, although this need not be the case. We shall assume that our aim is to eliminate backtracking and that the necessary starting point is an LL(1) grammar.

We must first remove all left recursion from our grammar, as discussed earlier.

We then need to partly eliminate alternatives within the right hand sides of all productions. This is known as *left factoring* the grammar. The general approach of left factoring is to modify the productions so that choices are not forced until the left context is clear enough to make an unambiguous decision. This makes our grammar suitable for predictive parsing.

Here is an example of left factoring.

$$\begin{array}{lcl} \mathit{stmt} & \rightarrow & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \\ & | & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt} \end{array}$$

This can be rewritten as

$$\begin{array}{lcl} \mathit{stmt} & \rightarrow & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt}\ \mathit{else-clause} \\ \mathit{else-clause} & \rightarrow & \textbf{else}\ \mathit{stmt} \\ & | & \epsilon \end{array}$$

Note that we still have an alternative in a right hand side, but that there is no *significant common prefix* any more. Thus it is possible to decide from the first symbol of the $else-clause$ which alternative is valid.

Note also that, strictly speaking, the presence of the empty string, ϵ, makes the choice just as ambiguous, since

$$\epsilon \text{ else } stmt \quad = \quad \text{else } stmt$$

We must, therefore, be clear that, in our parser, ϵ will be taken to mean a non-match for any of its alternatives within that production. i.e. in our example, if the next symbol is anything but **else**, we assume ϵ.

An example of improving a grammar

We may also use transition diagrams, like those we used for regular expressions. These may help us to simplify our parser.

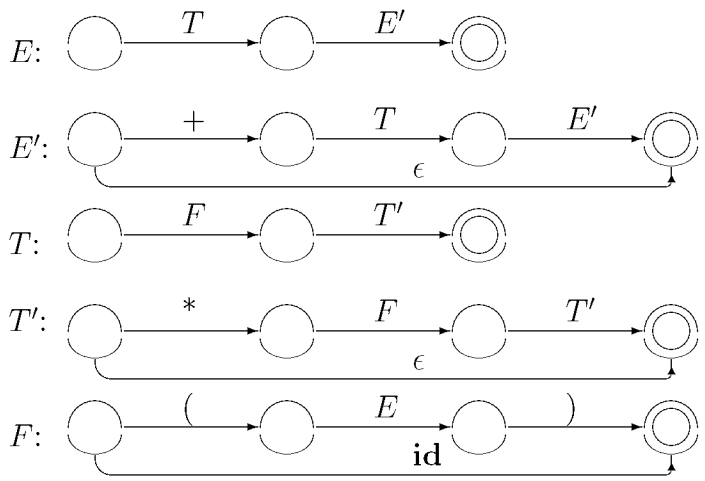
From the grammar for arithmetic expressions

$$\begin{array}{lcl} exp & \rightarrow & exp + term \mid term \\ term & \rightarrow & term * factor \mid factor \\ factor & \rightarrow & (\ exp\ ) \mid \mathbf{id} \end{array}$$

We first remove left recursion, generating

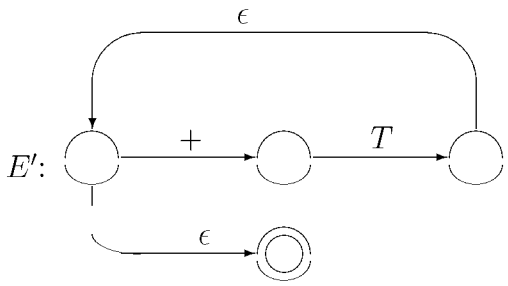
$$\begin{array}{lcl} exp & \rightarrow & term\ exp-tail \\ exp-tail & \rightarrow & +\ term\ exp-tail \mid \epsilon \\ term & \rightarrow & factor\ term-tail \\ \text{term-tail} & \rightarrow & *\ factor\ term-tail \mid \epsilon \\ factor & \rightarrow & (\ exp\ ) \mid \mathbf{id} \end{array}$$

This gives the following transition diagrams (all from A, S and U, pp184-5).

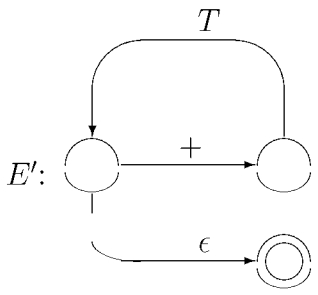


Simplifying these diagrams may make for a more compact and efficient parser. As they stand they are a very literal, recursive definition.

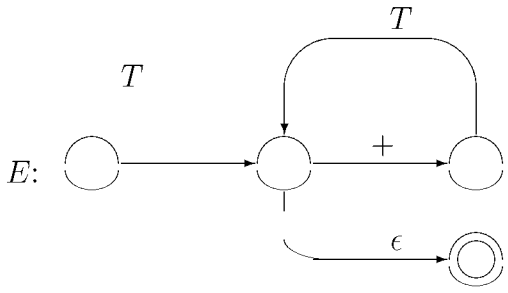
Firstly, we eliminate self-recursion in the $exp - tail$ transition diagram, substituting an iterative model. (This is the elimination of tail recursion. In some functional languages, like SML, this is performed by the compiler.)



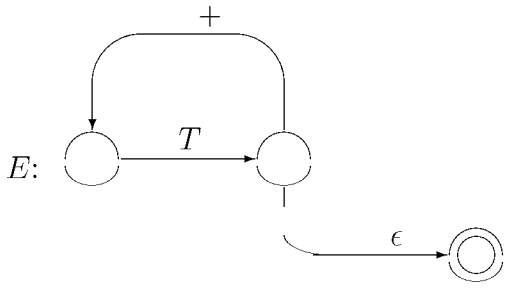
Further simplifying, by removing the redundant ϵ edge from 5 to 3, we get:



If we now substitute the exp-tail transition diagram into the exp one, replacing the exp-tail edge from 1 to 2, we get:



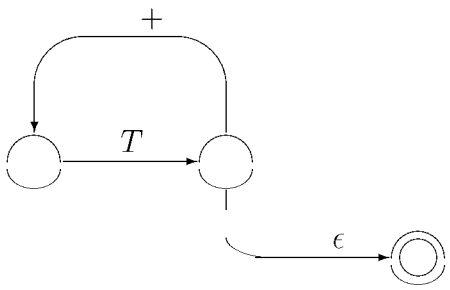
This, in turn, can be simplified to:



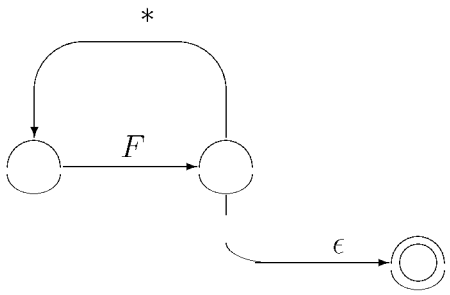
Applying the same approach to term and term-tail, we get a reduced set of diagrams for

arithmetic expressions.

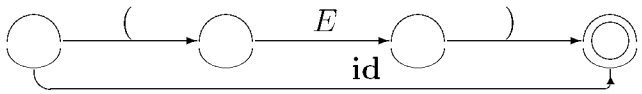
E:



T:



F:



Note that this is the set produced directly from the grammar:

exp	→	*term* ❨ + *term* ❩*
term	→	*factor* ❨ * *factor* ❩*
factor	→	(*exp*) | **id**

Thus, certain grammars will produce more efficient parsers than others, for the same language. The use of transition diagrams may help to reveal weaknesses in an initial grammar.

A recursive descent parser

One way to make the generation of a recursive descent parser clearer is to map BNF forms onto program constructs. Davie and Morrison give an outline of such an approach in their book. In Crookes paper (Software Engineering Journal May 1987) he outlines an automatic recursive descent parser generator.

In general we write a procedure for each non-terminal in the (post lexical analysis) language definition. Within these procedures, the coding is semi-automatic.

1. Where the right hand side contains alternatives the next symbol to be input provides the selector for a switch statement, each branch of which represents one alternative.

2. If ϵ is an alternative, this is the `default` of the switch.

3. Where a repetition occurs this may be implemented iteratively, i.e. by a while or until statement, with greater efficiency than is possible for recursion.

4. The sequence of actions within each branch (possibly only one) consists of an inspection of the lexical token for each terminal and a procedure call for each non-terminal. In some cases a lexical value or value set by a procedure will be assigned to a local or global variable, as required by the semantics of the language.

5. The order of statements matches the order of symbols in the production.

A recursive descent calculator

If we assume that the **id** in our earlier grammar is the lexical token for an integer, we can write a simple calculator program using these rules, as follows.

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

   void err(const char* s)
{
      perror(s);
      exit(0);
   }

int getint(int ch)
{

   int val;

   if ( isdigit(ch) ) val = ch - '0';
else err("Illegal character in place of integer");
   while( ((ch = getch())!=EOF) && isdigit(ch) )
      val = val * 10 + ch - '0';
   ungetc(ch, stdin);
   return val;
}

int expr();

int  factor()
{
   int ch, val;    printf("f \n");
   switch (ch=getch())
   {
case '(':   val = expr();
          if ((ch=getch())!=')') err("Missing closing parenthesis in
factor");
          break;

case '0':
case '1':
case '2':
```

```
case '3':
case '4':
case '5':
case '6':
case '7':
case '8':
case '9': val = getint(ch);
          break;

default: err("Illegal character at start of factor");
   }
   return val;
}

int term()
{
   int val, ch;    printf("t \n");
   val = factor();
   While ((ch=getch())=='*') val = val * factor();
   ungetc(ch,stdin);
   return val;
}

int expr()
{
   int ch, val; printf("e \n");
   val = term();
   While ((ch=getch())=='+') val = val + term();
   if(ch==')') ungetc(ch, stdin);
   return val;
}

void main()
{
   printf("%d\n",expr());
}
```

Parser generators

The rules for producing a parser using recursive descent from a BNF description are very systematic. Although some texts, such as A, S and U, only deal with automatic parser generators for bottom up, LR based parsers, a number of recursive-descent-parser generators have been produced.

Error detection and recovery

There are a number of strategies for error recovery in a top down parser. In general, a top down parser knows that one of a small set of symbols (terminal or non-terminal) must follow from the current left context. In a predictive parser this is certain knowledge. Thus an appropriate error message is easy to print.

The ideal of error recovery is to avoid halting when an error of syntax is found and, instead, to find the start of the next correct sequence, whence parsing may begin again. Of the many possibilities, two important categories exist.

- Skipping to a new start
- Inserting symbols to correct the current sequence
- Including common errors in the grammar

Skipping usually operates by defining the set of symbols which would follow the current one if it were correct. Upon detecting an error, the parser then skips until one of these is found. Typically the symbol skipped to defines the end of the current structure, such as a semi-colon or the keyword **end**.

To deal with the possibility that the next input symbolhas been inserted accidently into a correct program, the current predicted symbol type can be added to the set of those to be skipped to.

The Welsh and Hay Pascal compiler uses this skipping technique. It is described and its complete code given in "A Model Implementation of Standard Pascal", Welsh and Hay, published by Prentice Hall.

Inserting symbols to force the current symbol to be correct may recover from common mistakes, like missing semi-colons. This can be very effective for common errors such as typing = instead of := in Pascal. More sophisticated strategies allow many common errors to be detected.

Error productions are additional production rules added to the grammar, describing frequently found mistakes. The compiler can then report a very accurate diagnosis and "repair" the source, often allowing a correct compilation.

In fact many common errors can be ignored or repaired and some compilers choose to make such errors the subject of warnings and continue with code generation. This may be helpful, but can confuse inexperienced programmers.

There are always cases where an error causes the syntactic analysis to fail completely. In the IMP programming language, a common problem arises with missing end quotes in string literals. As IMP allows ends of line within strings, recovery may be very difficult.

In general, it is easy to write parsers for correct programs. Compilers are most frequently judged, however, by their ability to cope with incorrect ones.