Top-down Parsing Cont'd

MICHAEL WOLLOWSKI

Left Recursion

Formally, a grammar is *left recursive* if there exist an $A \in NT$ such that there is a derivation $A \Rightarrow^{+} A\alpha$, for some string $\alpha \in (NT \cup T)^{+}$

Left-recursion typically, leads to non-termination in a top-down parser

In a top-down parser, any recursion must be right recursion

We would like to convert the left recursion to right recursion

Indirect Left Recursion

In addition to left-recursion that occurs for a given production, there is indirect left-recursion.

Indirect left-recursion occurs when a sequence of productions creates left-recursion.

Example:

S → Aa | b
A → Ac | Sd

Derivation: S → Aa → Sda

Removing Indirect Left Recursion

```
Arrange the NTs in some order $A_1, A_2, \ldots, A_n$
for i ← 1 to n
  for s ← 1 to i – 1
    replace each production $A_i \rightarrow A_s\gamma$
    with $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$, where $A_s \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
    are all the current productions for $A_s$

    eliminate any immediate left recursion on $A_i$
    using the direct transformation
```

The inner loop must start with 1 to ensure that $A_1 \rightarrow A_1 \beta$ is transformed

Assumes that the initial grammar has no cycles ($A_i \Rightarrow^+ A_i$) and no epsilon productions

Removing Indirect Left Recursion

Example revised:

S → Aa | b

A → Ac | Sd

Order of non-terminals: S, A
Pairings according to algorithm: <S, S> and <A, S>
There is no production S → S
We do have a production of the form A → Sγ
In A → Sd we replace S with Aa and b, giving us:
A → Aad | bd

Removing Indirect Left Recursion

We now replace all immediate left-recursion in the two A productions:

A → Ac | Aad | bd

like so:

A → bdA’

A’ → cA’ | adA’ | ε

We’ll throw in the unmodified productions of S for free:

S → Aa | b

Predictive Parsing

An LL(1) grammar is considered a predictive grammar.

Reminder: Left-to-right scan, Left-most derivation, (1) word look-ahead

By removing left-recursion, i.e. by making the grammar right-recursive, we can create left-most derivations.

A predictive parser is based on a predictive grammar.

We will now focus on the look-ahead.

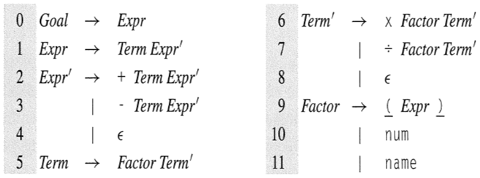
Predictive Parsing

Given the input a+b*c, the lexical analyzer will eventually produce the following sequence of tokens:

<ID, a> <Operator, +> <ID, b> <Operator, *> <ID, c>

A parser with 0 token look-ahead will proceed as follows:

```
Goal
Expr
Term Expr'
Factor Term' Expr'
```



0	*Goal*	→	*Expr*
1	*Expr*	→	*Term Expr′*
2	*Expr′*	→	+ *Term Expr′*
3		|	- *Term Expr′*
4		|	ϵ
5	*Term*	→	*Factor Term′*
6	*Term′*	→	x *Factor Term′*
7		|	÷ *Factor Term′*
8		|	ϵ
9	*Factor*	→	(*Expr*)
10		|	num
11		|	name

With 0 token look ahead, we have three choices for `Factor`:

- (
- num
- name

The parser would try all three.

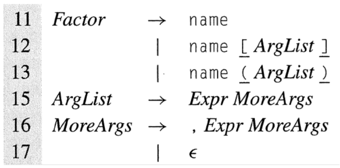
Predictive Parsing

- For each attempt, the parser would ask the lexical analyzer what token it has
- If it is not the right token, it would backtrack and try again
- This goes on until the parser reaches the last production and has success.
- This is silly, instead, grab the next symbol and make it available to the parser.
- This is an LL(1) grammar, also called a *predictive grammar*.

Left-Factoring to Eliminate Backtracking

We now have an almost back-track free grammar.

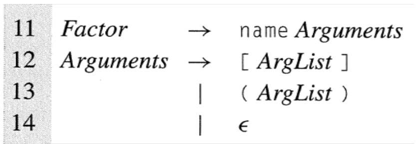
Consider:



11 *Factor* → `name`
12 | `name` [*ArgList*]
13 | `name` (*ArgList*)
15 *ArgList* → *Expr MoreArgs*
16 *MoreArgs* → , *Expr MoreArgs*
17 | ϵ

Rules 11, 12 and 13 all begin with **name**.

Name is a common pre-fix to all three rules and can be eliminated by introducing a new production:



11 *Factor* → `name` *Arguments*
12 *Arguments* → [*ArgList*]
13 | (*ArgList*)
14 | ϵ

Left-Factoring to Eliminate Backtracking

In general, we can *left-factor* any set of rules that has alternate right-hand sides with a common prefix.

Convert a set of productions:

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_j$

where $\boldsymbol{\alpha}$ is a common prefix and the γ_i's represent rhs that do not begin with $\boldsymbol{\alpha}$.

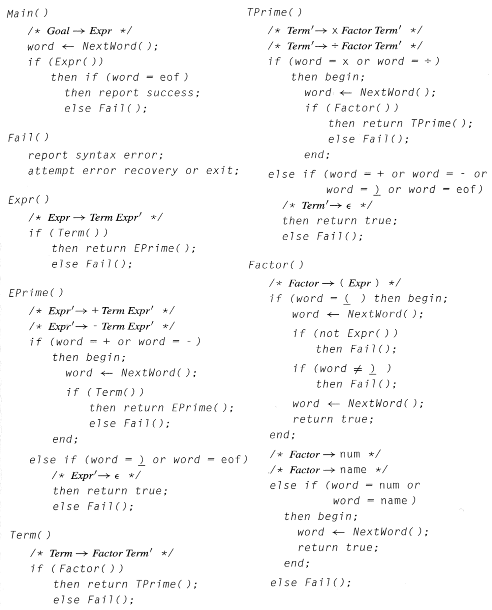
To:

$A \rightarrow \alpha B \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_j$

$B \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

Top-Down Recursive-Descent Parsers

- We will now have a look at top-down recursive descent parsers
- To make them work, we need to know the leading words that a production might encounter



```
Main( )
    /* Goal → Expr */
    word ← NextWord( );
    if (Expr( ))
        then if (word = eof )
            then report success;
            else Fail( );

Fail( )
    report syntax error;
    attempt error recovery or exit;

Expr( )
    /* Expr → Term Expr' */
    if ( Term( ) )
        then return EPrime( );
        else Fail( );

EPrime( )
    /* Expr' → + Term Expr' */
    /* Expr' → - Term Expr' */
    if (word = + or word = - )
        then begin;
            word ← NextWord( );
            if ( Term( ) )
                then return EPrime( );
                else Fail( );
        end;
    else if (word = ) or word = eof)
        /* Expr' → ε */
        then return true;
        else Fail( );

Term( )
    /* Term → Factor Term' */
    if ( Factor( ) )
        then return TPrime( );
        else Fail( );

TPrime( )
    /* Term' → x Factor Term' */
    /* Term' → ÷ Factor Term' */
    if (word = x or word = ÷ )
        then begin;
            word ← NextWord( );
            if ( Factor( ) )
                then return TPrime( );
                else Fail( );
            end;
    else if (word = + or word = - or
             word = ) or word = eof)
        /* Term' → ε */
        then return true;
        else Fail( );

Factor( )
    /* Factor → ( Expr ) */
    if (word = (  ) then begin;
        word ← NextWord( );
        if (not Expr( ) )
            then Fail( );
        if (word ≠ ) )
            then Fail( );
        word ← NextWord( );
        return true;
    end;
    /* Factor → num */
    /* Factor → name */
    else if (word = num or
             word = name )
        then begin;
            word ← NextWord( );
            return true;
        end;
    else Fail( );
```

First(α)

If α is any string of grammar symbols, let *First(α)* be the set of terminals that begin the strings derived from α.

$\alpha \in \text{Terminals} \cup \text{Non-terminals} \cup \{ \varepsilon \}$

Intuitively, for a non-terminal *A*, First(α) contains the complete set of terminal tokens that can appear as a leading symbol in a sentential form derived from *A*.

Calculating First

1. If X is a terminal, then First(X) is {X}
2. If X → ε is a production, then add ε to First(X)
3. Let $X \rightarrow Y_1 Y_2 \ldots Y_k$ be a production:
    a) If *a* is in First(Y_1), then place *a* is in First(X).
    b) If ε is in all of First(Y_1), …, First(Y_{i-1}), that is $Y_1 \ldots Y_{i-1} \Rightarrow^* \varepsilon$, then place all *a* in First(Y_i) into First(X).
    c) If ε is in First(Y_j) for all j = 1, 2, …,k, then add ε to First(X).

Example of Calculating First

Consider:

```
Expr  →  Term Expr’
Expr’ →  + Term Expr’  |  - Term Expr’  |  ε
Term  →  Factor Term’
Term’ →  * Factor Term’  |  / Factor Term’  |  ε
Factor→  (Expr)  |  num  | id
```

Initially, we create First sets for all the terminals:

	num	id	+	-	*	/	(	)	ε
First	num	id	+	-	*	/	(	)	ε

Next, the algorithm iterates over the productions, using First sets for the right-hand side of a production to derive the First set for the non-terminal on its left-hand side:

	Expr	Expr’	Term	Term’	Factor
First	(, id, num	+, -, ε	(, id, num	*, /, ε	(, id, num

Follow(A)

Define *Follow*(*A*), for a non-terminal *A*, to be the set of terminals *a* that can appear immediately to the right of *A* in some sentential form.

In other words, the set of terminals *a* are such that there exists a derivation of some form $S \Rightarrow^* \alpha A a \beta$ for some α and β.

Notice that at some point during the derivation there may have been non-terminals between *A* and *a*, but if so, they derived ε.

If *A* can be a right-most symbol in some sentential form, then **`eof`** is in Follow(*A*).

Calculating Follow

Recall that we do this only for non-terminals.

1. Put **`eof`** in Follow(S), where S is the start symbol
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in First(β) except for ε is placed in Follow(B)
3. If there is a production $A \rightarrow \alpha B$, then everything in Follow(A) is in Follow(B)
4. If there is a production $A \rightarrow \alpha B \beta$ where First(β) contains ε, i.e. $\beta \Rightarrow^{*} \varepsilon$, then everything in Follow(A) is in Follow(B)

Example of Calculating Follow

Consider:

```
Expr   →  Term Expr′
Expr′  →  + Term Expr′  |  - Term Expr′  |  ε
Term   →  Factor Term′
Term′  →  * Factor Term′  |  / Factor Term′  |  ε
Factor→  (Expr)  |  num  | id
```

Here are our First sets which we calculated earlier:

	Expr	Expr’	Term	Term’	Factor
First	(, id, num	+, -, ε	(, id, num	*, /, ε	(, id, num

Now, let’s calculate the Follow sets:

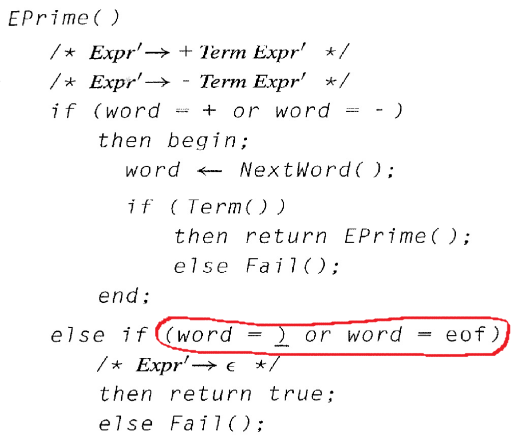
	Expr	Expr’	Term	Term’	Factor
1.	eof				
2.	eof		+, -		*, /
3.	eof,)		+, -		*, /
4.	eof,)	eof,)	+, -,	+, -,	*, /,
Follow	eof,)	eof,)	+, -, eof,)	+, -, eof,)	*, /, +, -, eof,)

Using Follow

In recursive-descent parsers the Follow sets are used for error checking in those cases where there is an e-production.

Consider the implementation of Expr’:

Near the bottom, we implement the e-production using the follow set of Expr’



```
EPrime( )
    /* Expr' → + Term Expr'  */
    /* Expr' → - Term Expr'  */
    if (word = + or word = - )
        then begin;
          word ← NextWord( );
          if ( Term( ) )
              then return EPrime( );
              else Fail();
        end;
    else if (word = ) or word = eof)
        /* Expr' → ε  */
        then return true;
        else Fail();
```

Table-Driven LL(1) Parsers

We have seen the core of a (top-down) recursive-descent parser.

It implements the productions directly through recursive procedures.

An abstraction of such an approach is offered by what is called a “table-driven” parser.

Rather than implement the rules in code, they are represented in a table.

We then write a procedure that based on the current grammar symbol and the current token provided by the Lexer looks up a production that is to be followed.

Table-driven Predictive Parser

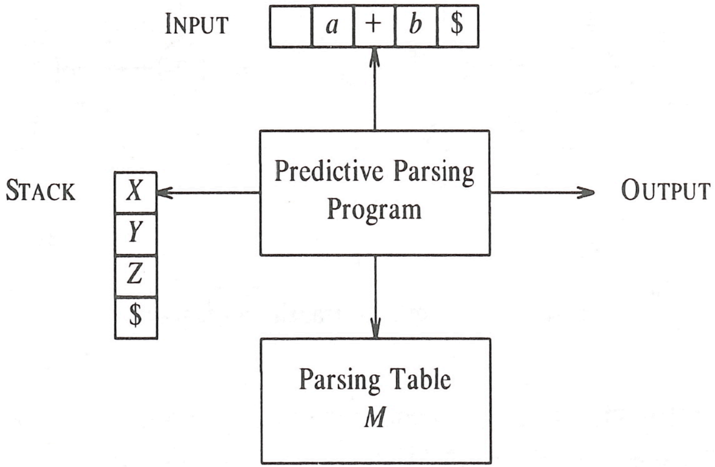


Table-Driven LL(1) Parsers

We will use the First and Follow sets to populate the entries in the parsing table *M* as follows:

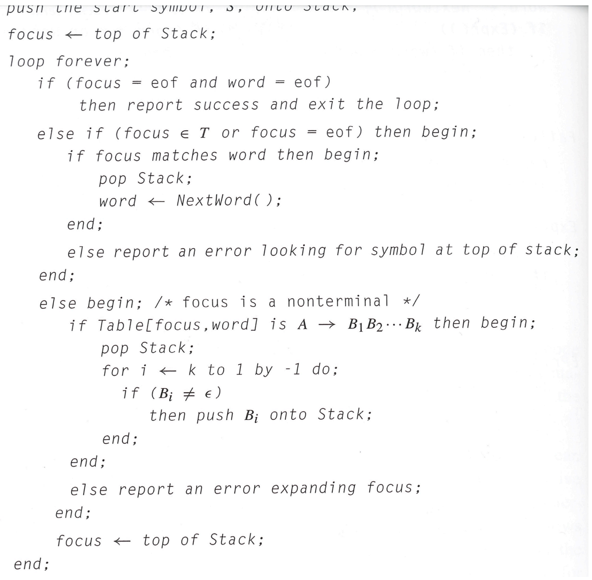
1. For each production *A* → α of the grammar, perform steps 2, 3 and 4.
2. For each terminal *a* in First(α), add the production *A* → α to *M*[*A*, *a*]
3. If ε is in First(α), add *A* → ε to *M*[*A*, *b*] for each terminal *b* in Follow(*A*).
4. If ε is in First(α) and **`eof`** is in Follow(*A*), add *A* → α to *M*[*A*, **`eof`**].
5. Make each undefined entry be an **error**.

Table-Driven LL(1) Parsers

Parse-table for our grammar:

	eof	+	-	*	/	(	)	id	num
Expr						E→TE′		E→TE′	E→TE′
Expr′	E′→ ε	E′→+TE′	E′→-TE′				E′→ ε		
Term						T→FT′		T→FT′	T→FT′
Term′	T′→ ε	T′→ ε	T′→ ε	T′→*FT′	T′→/FT′		T′→ ε		
Factor						F→(E)		F→ id	F→ num

Table-Driven LL(1) Parsers



```
push the start symbol, S, onto Stack;
focus ← top of Stack;
loop forever;
   if (focus = eof and word = eof)
       then report success and exit the loop;
   else if (focus ∈ T or focus = eof) then begin;
      if focus matches word then begin;
         pop Stack;
         word ← NextWord( );
      end;
      else report an error looking for symbol at top of stack;
   end;
   else begin; /* focus is a nonterminal */
      if Table[focus,word] is $A \rightarrow B_1 B_2 \cdots B_k$ then begin;
         pop Stack;
         for i ← k to 1 by -1 do;
           if ($B_i \neq \epsilon$)
              then push $B_i$ onto Stack;
         end;
      end;
      else report an error expanding focus;
   end;
   focus ← top of Stack;
end;
```

First/Follow Recap

First:

- Let A be a non-terminal, then *FIRST(A)* is defined to be the set of terminals that can appear in the first position of any string derived from A.
- FIRST is also defined for terminals, but its value is just equal to the terminal itself.

Follow:

- Let A be a non-terminal, then *FOLLOW(A)* is the union over FIRST(B) where B is any non-terminal that immediately follows A in the right hand side of a production rule.

FIRST shows us the terminals that can be at the beginning of a derived non-terminal, **FOLLOW** shows us the terminals that can come *after* a derived non-terminal.