

Programming Language Implementation IV

In this lecture we will look at **syntax analysis**. i.e. **parsing**.

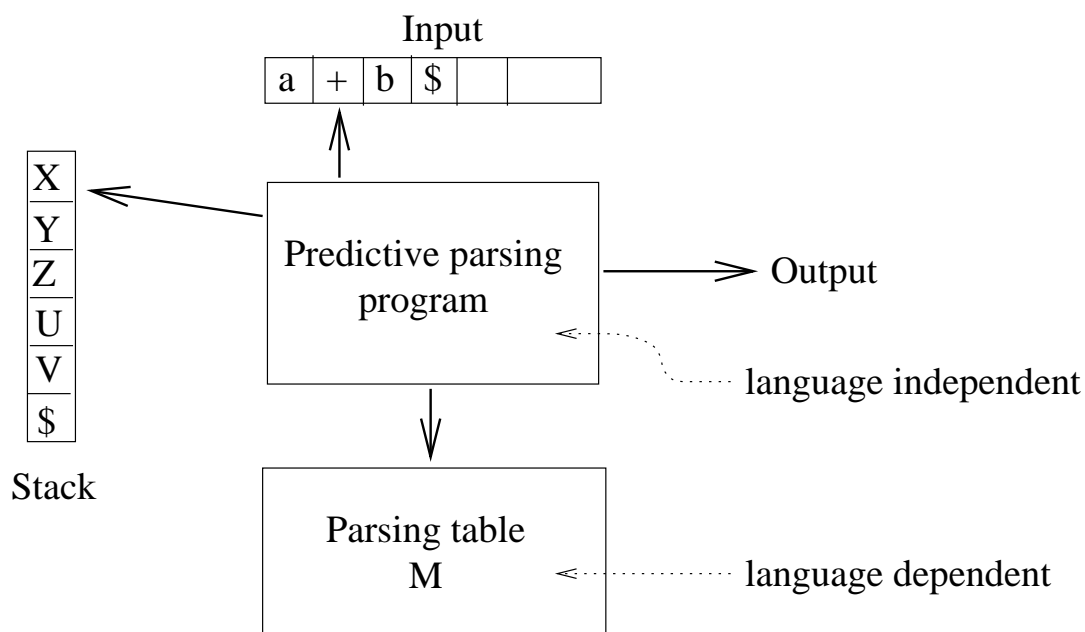
- **Table driven predictive parsing.**
- **LL(1) grammars.**

The material is (loosely) based on Aho et al Chapter 4.

Table-driven Predictive Parsing

It is possible to build a non-recursive predictive parser by maintaining a **stack** explicitly rather than implicitly via recursive calls.

In particular we can mechanically derive a **table driven** predictive parser from the grammar. This has the form



The **parsing table** is a 2-D array of entries $M[X, a]$ where X is a non-terminal and a is a terminal symbol or the special symbol $\$$ indicating end-of-input.

Predictive Parsing Algorithm

Independently of the target language, a predictive parser is controlled by a simple program which does the following:

```
initialize input to  $w\$$ ;  
inititalize stack to  $\$ S$ ;  
repeat{  
     $X := \text{top}()$ ;  
     $a := \text{currentSymbol}()$ ;  
    if  $X = \$$  or  $\text{isTerminal}(X)$  then  
        if  $X = a$  then {  $\text{pop}()$ ;  $\text{advanceInput}()$ ; }  
        else raise  $\text{syntaxError}$ ;  
    else  
        if  $M[X, a] = X \rightarrow Y_1, \dots, Y_k$  then {  
             $\text{pop}()$ ;  
             $\text{push}(Y_k)$ ; ...,  $\text{push}(Y_1)$ ;  
        }  
        else raise  $\text{syntaxError}$ ; (*  $M[X, a]$  empty *)  
    }  
until  $X = \$$ 
```

At each point the program considers the symbol on top of the stack X and a the current input symbol. There are 3 possibilities:

- If $X = a = \$$, the parser halts accepting the string.
- If $X = a \neq \$$, the parser pops X off the stack and “consumes” a , advancing to the next input symbol.
- If X is a non-terminal, the program determines the production to be applied from the parsing table.

Example

Recall the grammar

$$\begin{aligned} \textit{exp} &\rightarrow \textit{term exp}' && (P1) \\ \textit{exp}' &\rightarrow +\textit{exp} && (P2) \\ \textit{exp}' &\rightarrow \epsilon && (P3) \\ \textit{term} &\rightarrow \textit{factor term}' && (P4) \\ \textit{term}' &\rightarrow *\textit{term} && (P5) \\ \textit{term}' &\rightarrow \epsilon && (P6) \\ \textit{factor} &\rightarrow \mathbf{int} && (P7) \\ \textit{factor} &\rightarrow (\textit{exp}) && (P8) \end{aligned}$$

The parsing table for this grammar is

	int	+	*	()	\$
<i>exp</i>	<i>P1</i>			<i>P1</i>		
<i>exp'</i>		<i>P2</i>			<i>P3</i>	<i>P3</i>
<i>term</i>	<i>P4</i>			<i>P4</i>		
<i>term'</i>		<i>P6</i>	<i>P5</i>		<i>P6</i>	<i>P6</i>
<i>factor</i>	<i>P7</i>			<i>P8</i>		

Example (Cont.)

Consider parsing the symbol string

int + int

Stack	Input	Production
exp \$	int + int \$	P1
term exp' \$	int + int \$	P4
factor term' exp' \$	int + int \$	P7
int term' exp' \$	int + int \$	advance()
term' exp' \$	+ int \$	P6
exp' \$	+ int \$	P2
+ exp \$	+ int \$	advance()
exp \$	int \$	P1
term exp' \$	int \$	P4
factor term' exp' \$	int \$	P7
int term' exp' \$	int \$	advance()
term' exp' \$	\$	P6
exp' \$	\$	P3
\$	\$	accept!

Table-driven Predictive Parsing in ML

First we have to declare data types for the terminal and non-terminal symbols of the grammar and create data structures to store the parse table and the productions.

Note that only the right-hand side of the productions need to be stored (as the left-hand side is accessed via the parse table).

```
datatype lexResult = PLUS | TIMES | INT |
                    LPAREN | RPAREN | EOF ;
datatype nonTerminal = EXP | EXP1 | TERM | TERM1 |
                    FACTOR ;

datatype token = NT of nonTerminal | T of lexResult;

val productions =
  [[NT(TERM), NT(EXP1)],
   [T(PLUS), NT(EXP)],
   [],
   [NT(FACTOR), NT(TERM1)],
   [T(TIMES), NT(TERM)],
   [],
   [T(INT)],
   [T(LPAREN), NT(EXP), T(RPAREN)]]
];
```

Table-driven Predictive Parsing in ML (cont.)

Note how the parse table and its matching function can be implemented in an integrated form as a function using pattern matching. The parseTable lookup returns an index into the list of productions.

```
fun parseTable EXP      INT      = 1
  | parseTable EXP      LPAREN   = 1
  | parseTable EXP1     PLUS     = 2
  | parseTable EXP1     RPAREN   = 3
  | parseTable EXP1     EOF      = 3
  | parseTable TERM     INT      = 4
  | parseTable TERM     LPAREN   = 4
  | parseTable TERM1    PLUS     = 6
  | parseTable TERM1    TIMES    = 5
  | parseTable TERM1    RPAREN   = 6
  | parseTable TERM1    EOF      = 6
  | parseTable FACTOR   INT      = 7
  | parseTable FACTOR   LPAREN   = 8
  | parseTable _        _        = raise SYNTAX_ERROR;
```

Table-driven Predictive Parsing in ML (cont.)

The function `step` implements the core function of the table parser: It compares the top of the stack (first argument) with the current symbol of the input (second argument), pops the stack and either

- advances the input or
- performs a table lookup and pushes the right-hand side of the applicable production on the stack or
- raises a syntax error

```
fun step ([]:token list) rest = rest
| step _ [] = raise SYNTAX_ERROR
| step (T(top)::RestStack) (this::moreInput) =
    if (top=this) then
        (print "advance()\n";
         step RestStack moreInput
        )
    else raise SYNTAX_ERROR
| step (NT(top)::RestStack) (this::moreInput) =
    let val index = parseTable top this in
        (print ("executing production no. " ^
                Int.toString(index) ^ "\n");
         step (pushAll (List.rev
                        (List.nth(productions, index-1)))
              RestStack)
              (this::moreInput)
        )
    end;
```

```
fun pushAll [] stack = stack
  | pushAll (x::xs) stack = pushAll xs (x::stack);

fun parse () =
  let val allTokens = [INT, PLUS, INT, EOF] in
    let val rest = (step [NT(EXP)] allTokens) in
      if rest=[EOF] then print "accept\n"
      else raise SYNTAX_ERROR
    end
  end;
end;
```

How to Construct a Predictive Parser

To construct a predictive table parser we need to compute two sets: $FIRST(X)$ and $FOLLOW(X)$ for every grammar symbol X .

For a string (or symbol) X the set $FIRST(X)$ is the set of all terminal symbols with which strings derived from X can start (plus ϵ if X can be reduced to ϵ).

$FOLLOW(X)$ is the set of all terminal symbols that can immediately follow X in some sentential form (plus $\$$ if X can be the last symbol in a sentential form).

We can then apply the following algorithm to compute the parsing table

for each production $A \rightarrow w$

 for each terminal $a \in FIRST(w)$ add $A \rightarrow w$ to $M[A, a]$;

 if $\epsilon \in FIRST(w)$ then

 for each terminal b in $FOLLOW(A)$ add $A \rightarrow w$ to $M[A, b]$;

 if $\epsilon \in FIRST(w) \wedge \$ \in FOLLOW(A)$ then

 add $A \rightarrow w$ to $M[A, \$]$;

set all other entries to empty (error);

Computing FIRST

The table construction makes use of two functions **FIRST** and **FOLLOW** associated with any grammar.

For any sentence α , $FIRST(\alpha)$ is the set of terminals that begin the strings derived from α . If ϵ can be derived from α , then $\epsilon \in FIRST(\alpha)$.

For a string $\alpha = X_1, \dots, X_n$, we can compute $FIRST(\alpha)$ as follows:

1. $FIRST(\alpha) = FIRST(X_1) - \{\epsilon\}$
2. for $i = 2 \dots n$: if $\forall_{j < i} \epsilon \in FIRST(X_j)$ then add $FIRST(X_i) - \{\epsilon\}$ to $FIRST(\alpha)$.
3. add ϵ to $FIRST(\alpha)$ if $\forall_i : \epsilon \in FIRST(X_i)$.

We can compute $FIRST(X)$ for any grammar symbol X by exhaustively applying the following rules:

- If X is a terminal, $FIRST(X)$ is $\{X\}$.
- If $X \rightarrow \epsilon$ is a production, add ϵ to $FIRST(X)$.
- If $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production, then $FIRST(X) := FIRST(Y_1 Y_2 \dots Y_k)$

For example, consider the grammar:

$$\begin{aligned} \textit{exp} &\rightarrow \textit{term exp}' \\ \textit{exp}' &\rightarrow +\textit{exp} \\ \textit{exp}' &\rightarrow \epsilon \\ \textit{term} &\rightarrow \textit{factor term}' \\ \textit{term}' &\rightarrow *\textit{term} \\ \textit{term}' &\rightarrow \epsilon \\ \textit{factor} &\rightarrow \mathbf{int} \\ \textit{factor} &\rightarrow (\textit{exp}) \end{aligned}$$

Computing FOLLOW

For any non-terminal A , $FOLLOW(A)$ is the set of terminals that can immediately follow A in the strings derived from the start symbol S . If A can be the rightmost symbol in some sentence, then $\$ \in FOLLOW(A)$.

We can compute $FOLLOW(A)$ by exhaustively applying the following rules:

- Place $\$$ in $FOLLOW(S)$.
- If $A \rightarrow \alpha B \beta$ is a production, then everything in $FIRST(\beta)$ except for ϵ is placed in $FOLLOW(B)$.
- If there is a production $A \rightarrow \alpha B$ or a production $A \rightarrow \alpha B \beta$ where $\epsilon \in FIRST(\beta)$, then everything in $FOLLOW(A)$ is placed in $FOLLOW(B)$.

Exercise: Give the $FOLLOW$ sets for the previous grammar.

Construction Example

Recall

$$exp \rightarrow term\ exp' \quad (P1)$$

$$exp' \rightarrow +exp \quad (P2)$$

$$exp' \rightarrow \epsilon \quad (P3)$$

$$term \rightarrow factor\ term' \quad (P4)$$

$$term' \rightarrow *term \quad (P5)$$

$$term' \rightarrow \epsilon \quad (P6)$$

$$factor \rightarrow \mathbf{int} \quad (P7)$$

$$factor \rightarrow (exp) \quad (P8)$$

We have that

$$FIRST(exp) = FIRST(term) = FIRST(factor) = \{(\mathbf{int})\}$$

$$FIRST(exp') = \{+, \epsilon\}$$

$$FIRST(term') = \{*, \epsilon\}$$

$$FOLLOW(exp) = FOLLOW(exp') = \{), \$\}$$

$$FOLLOW(term) = FOLLOW(term') = \{+,), \$\}$$

$$FOLLOW(factor) = \{+, *,), \$\}.$$

The parsing table for this grammar is

	int	+	*	()	\$
<i>exp</i>						
<i>exp'</i>						
<i>term</i>						
<i>term'</i>						
<i>factor</i>						

Left Recursion Revisited

Left-recursive grammars cannot be processed with predictive table parsers either.

This can even be shown without knowing the precise parsing table.

Assume G contains the left-recursive production $E \rightarrow E + E \mid id$

Consider the parser state:

- top of stack = E ,
- current input = id ,
- $E \rightarrow E + E \in M[E, id]$

The predictive table parser would execute the following actions:

- pop();
- push(E); push(+); push(E);
- no advance in lookahead

Thus neither the top of the stack nor the current input symbol change and the parser will loop infinitely.

LL(1) Grammars

As we have seen not all grammars are suitable for predictive parsing since they cannot effectively predict which production should be applied.

This idea of “being suitable” for predictive parsing is captured in the $LL(k)$ property for a grammar. A grammar is called $LL(k)$ if the production to be applied can be determined with a maximum lookahead of k symbols. We are particularly interested in $LL(1)$.

A grammar is $LL(1)$ if the parse table generated for a lookahead of one symbol has no multiply-defined entries.

THEOREM: A grammar G is $LL(1)$ iff for each non-terminal A :

- If $A \rightarrow \alpha$ and $A \rightarrow \beta$ are different productions in G , then

$$FIRST(\alpha) \cap FIRST(\beta) = \emptyset.$$

(Note at most one of α and β can derive the empty string).

- If $\beta \Rightarrow^* \epsilon$,

$$FIRST(\alpha) \cap FOLLOW(A) = \emptyset.$$

If a grammar is not $LL(1)$ you can sometimes use left recursion removal and left factoring to transform it into an equivalent $LL(1)$ grammar.

However, not all context-free grammars have an equivalent $LL(1)$ grammar.

More generally we talk about $LL(k)$ grammars. These can be predictively parsed with k symbol lookahead.

Error Correction

Sensible error handling is extremely important: imagine your compiler would abort the compilation of a faulty program with just the message “error - compilation aborted”!

Four levels of behaviour are possible when an error is encountered:

1. locate & report (without further analysis),
2. diagnose (report the type of error),
3. recover (i.e. skip over the error and continue parsing afterwards),
4. correct.

Any good compiler implements some level of recovery. Correction is the “holy grail” of parsing, but only possible in special cases.

The stack in the table driven recursive parser makes explicit the terminals and non-terminals it expects to match with the rest of the input. This can be used to guide error recovery.

A **syntax error** is detected when

(a) the terminal on top of the stack does not match the input token,
or

(b) the non-terminal symbol on top of the stack A has an empty entry $M[A, a]$ for the current input symbol a .

Panic-mode error recovery is based on the idea of skipping input symbols until a token in a selected set of **synchronizing** tokens appears.

One heuristic is to place all of the symbols in $FOLLOW(A)$ into the synchronizing set for non-terminal A .

We then skip input until a member of $FOLLOW(A)$ is encountered, and pop A from the stack.

Another is to place all of the symbols in $FIRST(A)$ into the synchronizing set for non-terminal A .

We then skip input until a member of $FIRST(A)$ is encountered.

See Aho et al Section 4.4 for more details.

Error Correction – Example

Recall

$$exp \rightarrow term\ exp' \quad (P1)$$

$$exp' \rightarrow +exp \quad (P2)$$

$$exp' \rightarrow \epsilon \quad (P3)$$

$$term \rightarrow factor\ term' \quad (P4)$$

$$term' \rightarrow *term \quad (P5)$$

$$term' \rightarrow \epsilon \quad (P6)$$

$$factor \rightarrow \mathbf{int} \quad (P7)$$

$$factor \rightarrow (exp) \quad (P8)$$

The parsing table with error correction for this grammar is

	int	+	*	()	\$	<i>first</i>	<i>follow</i>
<i>exp</i>	<i>P1</i>	<i>skip</i>	<i>skip</i>	<i>P1</i>	<i>synch</i>	<i>synch</i>	(<i>int</i>) \$
<i>exp'</i>	<i>skip</i>	<i>P2</i>	<i>skip</i>	<i>skip</i>	<i>P3</i>	<i>P3</i>	+) \$
<i>term</i>	<i>P4</i>	<i>synch</i>	<i>skip</i>	<i>P4</i>	<i>synch</i>	<i>synch</i>	(<i>int</i>	+) \$
<i>term'</i>	<i>skip</i>	<i>P6</i>	<i>P5</i>	<i>skip</i>	<i>P6</i>	<i>P6</i>	*	+) \$
<i>factor</i>	<i>P7</i>	<i>synch</i>	<i>synch</i>	<i>P8</i>	<i>synch</i>	<i>synch</i>	(<i>int</i>	+ *) \$

A *skip* means to skip the current input terminal and continue.

A *synch* does the following where *A* is the non-terminal on top of the stack:

- skip input terminals until a symbol in $FIRST(A)$ or $FOLLOW(A)$ is encountered
- if the symbol is in $FOLLOW(A)$ pop *A* off the stack
- continue.

Error Correction – Example (Cont.)

Consider parsing the symbol string

)int + *int

Summary

We have looked at:

- Table-driven predictive parsing.
- LL(1) grammars.

Homework

- Read Section 4.4 of Aho et al.
- Modify the grammar for arithmetic expressions to include division and subtraction and identifier.
- Extend the table-driven predictive parser given in the lecture for this extended grammar and couple it to the `calcLex` lexer.
- Extend the table-driven predictive parser given in the lecture with error correction.