CMSC 22610	Implementation of	Handout 3
Winter 2009	Computer Languages - I	January 29, 2009

Predictive Parsing Notes

1 Grammars for Propostional Formulae

Infix

$$\begin{array}{lcl} P & \rightarrow & var \\ P & \rightarrow & \neg\, P \\ P & \rightarrow & P \wedge P \\ P & \rightarrow & P \vee P \end{array}$$

Infix with parens

$$\begin{array}{lcl} P & \rightarrow & var \\ P & \rightarrow & \neg\, P \\ P & \rightarrow & P \wedge P \\ P & \rightarrow & P \vee P \\ P & \rightarrow & \texttt{(}\ P\ \texttt{)} \end{array}$$

Prefix

$$\begin{array}{lcl} P & \rightarrow & var \\ P & \rightarrow & \neg\, P \\ P & \rightarrow & \wedge\ P\ P \\ P & \rightarrow & \vee\ P\ P \end{array}$$

Postfix

$$\begin{array}{lcl} P & \rightarrow & var \\ P & \rightarrow & P\ \neg \\ P & \rightarrow & P\ P\ \wedge \\ P & \rightarrow & P\ P\ \vee \end{array}$$

Prefix with parens

$$\begin{array}{lcl} P & \rightarrow & var \\ P & \rightarrow & \neg\, P \\ P & \rightarrow & \wedge\ P\ P \\ P & \rightarrow & \vee\ P\ P \\ P & \rightarrow & \texttt{(}\ P\ \texttt{)} \end{array}$$

Postfix with parens

$$\begin{array}{lcl} P & \rightarrow & var \\ P & \rightarrow & P\ \neg \\ P & \rightarrow & P\ P\ \wedge \\ P & \rightarrow & P\ P\ \vee \\ P & \rightarrow & \texttt{(}\ P\ \texttt{)} \end{array}$$

Function-style prefix

$$\begin{array}{lcl} P & \rightarrow & var \\ P & \rightarrow & \neg\ \texttt{(}\ P\ \texttt{)} \\ P & \rightarrow & \wedge\ \texttt{(}\ P\ \texttt{,}\ P\ \texttt{)} \\ P & \rightarrow & \vee\ \texttt{(}\ P\ \texttt{,}\ P\ \texttt{)} \end{array}$$

Function-style postfix

$$\begin{array}{lcl} P & \rightarrow & var \\ P & \rightarrow & \texttt{(}\ P\ \texttt{)}\ \neg \\ P & \rightarrow & \texttt{(}\ P\ \texttt{,}\ P\ \texttt{)}\ \wedge \\ P & \rightarrow & \texttt{(}\ P\ \texttt{,}\ P\ \texttt{)}\ \vee \end{array}$$

Function-style prefix with parens

$$\begin{array}{lcl} P & \rightarrow & var \\ P & \rightarrow & \neg\ \texttt{(}\ P\ \texttt{)} \\ P & \rightarrow & \wedge\ \texttt{(}\ P\ \texttt{,}\ P\ \texttt{)} \\ P & \rightarrow & \vee\ \texttt{(}\ P\ \texttt{,}\ P\ \texttt{)} \\ P & \rightarrow & \texttt{(}\ P\ \texttt{)} \end{array}$$

Function-style postfix with parens

$$\begin{array}{lcl} P & \rightarrow & var \\ P & \rightarrow & \texttt{(}\ P\ \texttt{)}\ \neg \\ P & \rightarrow & \texttt{(}\ P\ \texttt{,}\ P\ \texttt{)}\ \wedge \\ P & \rightarrow & \texttt{(}\ P\ \texttt{,}\ P\ \texttt{)}\ \vee \\ P & \rightarrow & \texttt{(}\ P\ \texttt{)} \end{array}$$

Scheme-style prefix		
P	$\rightarrow$	var
P	$\rightarrow$	$\texttt{(}\ \neg\ P\ \texttt{)}$
P	$\rightarrow$	$\texttt{(}\ \wedge\ P\ P\ \texttt{)}$
P	$\rightarrow$	$\texttt{(}\ \vee\ P\ P\ \texttt{)}$

Scheme-style postfix		
P	$\rightarrow$	var
P	$\rightarrow$	$\texttt{(}\ P\ \neg\ \texttt{)}$
P	$\rightarrow$	$\texttt{(}\ P\ P\ \wedge\ \texttt{)}$
P	$\rightarrow$	$\texttt{(}\ P\ P\ \vee\ \texttt{)}$

Scheme-style prefix with parens		
P	$\rightarrow$	var
P	$\rightarrow$	$\texttt{(}\ \neg\ P\ \texttt{)}$
P	$\rightarrow$	$\texttt{(}\ \wedge\ P\ P\ \texttt{)}$
P	$\rightarrow$	$\texttt{(}\ \vee\ P\ P\ \texttt{)}$
P	$\rightarrow$	$\texttt{(}\ P\ \texttt{)}$

Scheme-style postfix with parens		
P	$\rightarrow$	var
P	$\rightarrow$	$\texttt{(}\ P\ \neg\ \texttt{)}$
P	$\rightarrow$	$\texttt{(}\ P\ P\ \wedge\ \texttt{)}$
P	$\rightarrow$	$\texttt{(}\ P\ P\ \vee\ \texttt{)}$
P	$\rightarrow$	$\texttt{(}\ P\ \texttt{)}$

- Which grammars denote the same context-free languages?
- Which grammars are unambiguous? which are ambiguous?
- Which grammars have immediate left recursion?
- Which grammars can be left factored?
- Which grammars are LL(1)?
- Which grammars are LL(2)?
- Which grammars are not LL(k) for any k?
- Which grammars denote languages that are LL(1)?
- Which grammars denote languages that are LL(2)?
- Which grammars denote languages that are not LL(k) for any k?

2 Recursive-Descent Parsing

The idea of recursive descent parsing is that a CFG can be mapped directly to a collection of mutually-recursive functions, with one function for each nonterminal and one clause for each production. For example, consider the following grammar:

Imperative Boolean Language		
S	$\rightarrow$	**if** P **then** S **else** S
S	$\rightarrow$	**while** P **do** S
S	$\rightarrow$	*var* **=** P
S	$\rightarrow$	**begin** S L
S	$\rightarrow$	**print** P
L	$\rightarrow$	**;** S L
L	$\rightarrow$	**end**
P productions from *Function-style prefix*		

Given a suitable definition of tokens and functions for fetching tokens from the token stream, we can write SML functions for parsing S, L, and P:

```
datatype token = EOP (* special end-of-parse token *)
 | KW_if | KW_then | KW_else | KW_while | KW_do
 | KW_begin | KW_end | KW_print | Var of string
 | EQ | SEMI | NOT | AND | OR | LPAREN | RPAREN | COMMA

(* returns the current token. *)
fun curTok () : token = ...
(* discards the current token, and moves to the next token *)
fun advanceTok () : unit = ...

fun advanceIfTok (tok) =
  if (curTok () = tok) then advanceTok () else error ()
fun error () = raise ParseError

fun parseS () = (case curTok () of
     KW_if => (advanceTok (); (* consume KW_if token *)
               parseP ();
               advanceIfTok (KW_then); parseS ();
               advanceIfTok (KW_else); parseS ())
   | KW_while => (advanceTok (); (* consume KW_while token *)
                  parseP ();
                  advanceIfTok (KW_do); parseS ())
   | Var _ => (advanceTok (); (* consume Var token *)
               advanceIfTok (EQ); parseP ())
   | KW_begin => (advanceTok (); parseS (); parseL ())
   | KW_print => (advanceTok (); parseP ())
   | _ => error ())
```

```
and parseL () = (case curTok () of
     SEMI => (advanceTok (); parseS (); parseL ())
   | KW_end => (advanceTok ())
   | _ => error ())
and parseP () = (case curTok () of
     Var _ => (advanceTok ())
   | NOT => (advanceTok (); advanceIfTok (LPAREN);
             parseP (); advanceIfTok (RPAREN))
   | AND => (advanceTok (); advanceIfTok (LPAREN);
             parseP (); advanceIfTok (COMMA);
             parseP (); advanceIfTok (RPAREN))
   | OR => (advanceTok (); advanceIfTok (LPAREN);
            parseP (); advanceIfTok (COMMA);
            parseP (); advanceIfTok (RPAREN))
   | _ => error ())

(* Parsing the whole input requires that, after parsing an S, *)
(* the final token is the EOP token.                          *)
fun parse () = (parseS (); advanceIfTok (EOP))
```

2.1 Constructing the abstract parse tree

A realistic parser, in addition to recognizing that a string is derivable in the grammar, will construct an abstract parse tree, reflecting the relevant portions of the derivation tree.

```
and parseP () = (case curTok () of
     Var s => Prop.Var s
   | NOT => let
              val _ = advanceTok ()  val _ = advanceIfTok (LPAREN)
              val p = parseP ()  val _ = advanceIfTok (RPAREN)
            in
              Prop.Not p
            end
   | AND => let
              val _ = advanceTok ()  val _ = advanceIfTok (LPAREN)
              val p = parseP ()  val _ advanceIfTok (COMMA)
              val q = parseP ()  val _ advanceIfTok (RPAREN)
            in
              Prop.And (p, q)
            end
   | OR => let
             val _ = advanceTok ()  val _ = advanceIfTok (LPAREN)
             val p = parseP ()  val _ advanceIfTok (COMMA)
             val q = parseP ()  val _ advanceIfTok (RPAREN)
           in
             Prop.Or (p, q)
           end
   | _ => error ())
```

2.2 Limitations of Recursive-Descent Parsing

Can we apply the same technique to parse the *P* productions for *Postfix*? If we try, then we are led to write the following function for `parseP`:

```
and parseP () = (case curTok () of
      Var _ => (advanceTok ())
    | ?? => (parseP () advanceIfTok (NOT))
    | ?? => (parseP (); parseP (); advanceIfTok (AND))
    | ?? => (parseP (); parseP (); advanceIfTok (OR))
    | _ => error ())
```

The `parseP` function has no way to know which clause to use; consider parsing the strings $x\ y \wedge \neg$ and $x\ y \neg \wedge$. In the former case, the initial call to `parseP` should use the $P \rightarrow P\ P \wedge$ production, but the latter case should use the $P \rightarrow P \neg$

- Which grammars from Section 1 can be parsed with recursive-descent parsing?

3 Grammar Transformations

There are a few grammar transformations that can turn a grammar that cannot be parsed with recursive-descent parsing into one that is more amenable to recursive-descent parsing.

3.1 Eliminating Immediate Left-recursion

A nonterminal A exhibits *immediate left-recursion* if there is production of the form $A \rightarrow A\ \beta$. (A grammar exibits *left-recursion* if there is a nonterminal A such that $A \Rightarrow^{+} A\ \beta$.)

Infix
without immediate left recursion

$$
\begin{array}{lcl}
P & \rightarrow & var\ P' \\
P & \rightarrow & \neg\ P\ P' \\
\\
P' & \rightarrow & \wedge\ P\ P' \\
P' & \rightarrow & \vee\ P\ P' \\
P' & \rightarrow & \epsilon
\end{array}
$$

Infix with parens
without immediate left recursion

$$
\begin{array}{lcl}
P & \rightarrow & var\ P' \\
P & \rightarrow & \neg\ P\ P' \\
P & \rightarrow & \texttt{(}\ P\ \texttt{)}\ P' \\
\\
P' & \rightarrow & \wedge\ P\ P' \\
P' & \rightarrow & \vee\ P\ P' \\
P' & \rightarrow & \epsilon
\end{array}
$$

Postfix
without immediate left recursion

$$
\begin{array}{lcl}
P & \rightarrow & var\ P' \\
\\
P' & \rightarrow & \neg\ P' \\
P' & \rightarrow & P \wedge P' \\
P' & \rightarrow & P \vee P' \\
P' & \rightarrow & \epsilon
\end{array}
$$

Postfix with parens
without immediate left recursion

- Which of these grammars are ambiguous?
- Which of these grammars can be parsed with recursive-descent parsing?
- How do we know when to use the $P' \rightarrow \epsilon$ productions?

3.2 Left Factoring

A grammar for which there are two productions for the same nonterminal that begin with same terminal ($A \longrightarrow \underline{a}\ \beta_1$ and $A \longrightarrow \underline{a}\ \beta_2$) cannot be parsed with recursive-descent parsing. We can delay the choice of production by using *left-factoring*.

Postfix
without immediate left recursion
with left factoring

$$
\begin{array}{lcl}
P & \longrightarrow & var\ P' \\
\\
P' & \longrightarrow & \neg\ P' \\
P' & \longrightarrow & P\ Z \\
P' & \longrightarrow & \epsilon \\
\\
Z & \longrightarrow & \wedge\ P' \\
Z & \longrightarrow & \vee\ P'
\end{array}
$$

Postfix with parens
without immediate left recursion

$$
\begin{array}{lcl}
P & \longrightarrow & var\ P' \\
P & \longrightarrow & \texttt{(}\ P\ \texttt{)}\ P' \\
\\
P' & \longrightarrow & \neg\ P' \\
P' & \longrightarrow & P\ Z \\
P' & \longrightarrow & \epsilon \\
\\
Z & \longrightarrow & \wedge\ P' \\
Z & \longrightarrow & \vee\ P'
\end{array}
$$

Function-style postfix
with left factoring

$$
\begin{array}{lcl}
P & \longrightarrow & var \\
P & \longrightarrow & \texttt{(}\ P\ Z \\
\\
Z & \longrightarrow & \texttt{)}\ \neg \\
Z & \longrightarrow & \texttt{,}\ P\ \texttt{)}\ Y \\
\\
Y & \longrightarrow & \wedge \\
Y & \longrightarrow & \vee
\end{array}
$$

Function-style postfix with parens
with left factoring

Scheme-style prefix
with left factoring

$$
\begin{array}{lcl}
P & \longrightarrow & var \\
P & \longrightarrow & \texttt{(}\ Z \\
\\
Z & \longrightarrow & \neg\ P\ \texttt{)} \\
Z & \longrightarrow & \wedge\ P\ P\ \texttt{)} \\
Z & \longrightarrow & \vee\ P\ P\ \texttt{)}
\end{array}
$$

Scheme-style postfix
with left factoring

Scheme-style prefix with parens with left factoring

Scheme-style postfix with parens with left factoring

- Which of these grammars are "obviously" grammars for boolean expressions?

- Which of these grammars can be parsed with recursive-descent parsing?

- How do we know when to use the $P' \rightarrow \epsilon$ productions?

- For a grammar that can be parsed with recursive-descent parsing, how would you construct the abstract parse tree?

3.2.1 Constructing the Abstract Parse Tree with Higher-Order Functions

When we delay the choice of production by using left-factoring, the new nonterminal can return a function representing the chosen production.

- Suppose we did not require the `parseZ` and `parseY` functions to have types of the form `unit -> ...`. Is there a simpler way to construct the abstract parse tree?

3.3 Specifing Precedence and Associativity

A common source of ambiguity in grammars is the *precedence* and *associativity* of operators. We can specify precedence in a grammar by requiring lower precedence operators to contain equal-or-higher precedence operators (and prohibit higher precedence operators from containing lower precedence operators).

Infix with parens
with $\{\vee\} < \{\wedge\} < \{\neg\}$

$$
\begin{array}{lcl}
P & \rightarrow & O \\
\\
O & \rightarrow & O \vee O \\
O & \rightarrow & A \\
\\
A & \rightarrow & A \wedge A \\
A & \rightarrow & Z \\
\\
Z & \rightarrow & \mathit{var} \\
Z & \rightarrow & \neg\, Z \\
Z & \rightarrow & \textbf{(}\ P\ \textbf{)}
\end{array}
$$

Infix with parens
with $\{\wedge\} < \{\vee\} < \{\neg\}$

Infix with parens
with $\{\neg\} < \{\vee\} < \{\wedge\}$

$$
\begin{array}{lcl}
P & \rightarrow & N \\
\\
N & \rightarrow & \neg\, N \\
N & \rightarrow & O \\
\\
O & \rightarrow & O \vee O \\
O & \rightarrow & A \\
\\
A & \rightarrow & A \wedge A \\
A & \rightarrow & Z \\
\\
Z & \rightarrow & \mathit{var} \\
Z & \rightarrow & \textbf{(}\ P\ \textbf{)}
\end{array}
$$

- How would the string $\neg\, x \wedge \neg\, y \vee z$ be parsed in each of these grammars?
  - $\{\vee\} < \{\wedge\} < \{\neg\}$:
  - $\{\wedge\} < \{\vee\} < \{\neg\}$:
  - $\{\neg\} < \{\vee\} < \{\wedge\}$:

We can specify associativity in a grammar by requiring equal-or-higher precedence operators in one branch and strictly higher precedence operators in the other branch.

Infix with parens with $\{\vee_L\} < \{\wedge_L\} < \{\neg\}$

$$\begin{array}{lcl}
P & \rightarrow & O \\
\\
O & \rightarrow & O \vee A \\
O & \rightarrow & A \\
\\
A & \rightarrow & A \wedge Z \\
A & \rightarrow & Z \\
\\
Z & \rightarrow & var \\
Z & \rightarrow & \neg\, Z \\
Z & \rightarrow & \textbf{(}\ P\ \textbf{)}
\end{array}$$

Infix with parens with $\{\wedge_R\} < \{\vee_R\} < \{\neg\}$

$$\begin{array}{lcl}
P & \rightarrow & A \\
\\
A & \rightarrow & O \wedge A \\
A & \rightarrow & O \\
\\
O & \rightarrow & Z \vee O \\
O & \rightarrow & Z \\
\\
Z & \rightarrow & var \\
Z & \rightarrow & \neg\, Z \\
Z & \rightarrow & \textbf{(}\ P\ \textbf{)}
\end{array}$$

Infix with parens with $\{\vee_L, \wedge_L\} < \{\neg\}$

$$\begin{array}{lcl}
P & \rightarrow & OA \\
\\
OA & \rightarrow & OA \wedge Z \\
OA & \rightarrow & OA \vee Z \\
OA & \rightarrow & Z \\
\\
Z & \rightarrow & var \\
Z & \rightarrow & \neg\, Z \\
Z & \rightarrow & \textbf{(}\ P\ \textbf{)}
\end{array}$$

Infix with parens with $\{\vee_R, \wedge_R\} < \{\neg\}$

$$\begin{array}{lcl}
P & \rightarrow & OA \\
\\
OA & \rightarrow & Z \wedge OA \\
OA & \rightarrow & Z \vee OA \\
OA & \rightarrow & Z \\
\\
Z & \rightarrow & var \\
Z & \rightarrow & \neg\, Z \\
Z & \rightarrow & \textbf{(}\ P\ \textbf{)}
\end{array}$$

- How would the string $a \wedge b \wedge c \vee d \wedge e \vee f \wedge g \vee h \vee i$ be parsed in each of these grammars?
  - $\{\vee_L\} < \{\wedge_L\} < \{\neg\}$:
  - $\{\wedge_R\} < \{\vee_R\} < \{\neg\}$:
  - $\{\vee_L, \wedge_L\} < \{\neg\}$:
  - $\{\vee_R, \wedge_R\} < \{\neg\}$:
- Which of these grammars are ambiguous?
- Which of these grammars can be parsed with recursive-descent parsing?

Infix with parens
with $\{\vee_L\} < \{\wedge_L\} < \{\neg\}$
without immediate left recursion

P	$\rightarrow$	*O*
O	$\rightarrow$	*A O'*
O'	$\rightarrow$	$\vee$ *A O'*
O'	$\rightarrow$	ϵ
A	$\rightarrow$	*Z A'*
A'	$\rightarrow$	$\wedge$ *Z A'*
A'	$\rightarrow$	ϵ
Z	$\rightarrow$	*var*
Z	$\rightarrow$	$\neg$ *Z*
Z	$\rightarrow$	**(** *P* **)**

Infix with parens
with $\{\wedge_R\} < \{\vee_R\} < \{\neg\}$
with left factoring

P	$\rightarrow$	*A*
A	$\rightarrow$	*O A'*
A'	$\rightarrow$	$\wedge$ *A*
A'	$\rightarrow$	ϵ
O	$\rightarrow$	*Z O'*
O'	$\rightarrow$	$\vee$ *O*
O'	$\rightarrow$	ϵ
Z	$\rightarrow$	*var*
Z	$\rightarrow$	$\neg$ *Z*
Z	$\rightarrow$	**(** *P* **)**

- Are the *A'* and *O'* productions reminiscient of any other transformations we have seen?

Next week, we'll see how shift-reduce parsing admits an easy specification of precedence and associativity of operators in an LR parser specification. Nonethless, Extended BNF can be used to give a concise specification for a parser generator that supports EBNF.

Infix with parens
with $\{\vee_L\} < \{\wedge_L\} < \{\neg\}$
using Extended BNF

$$
\begin{array}{lcl}
P & \rightarrow & O \\
O & \rightarrow & A\ (\vee\ A)^* \\
A & \rightarrow & Z\ (\wedge\ Z)^* \\
Z & \rightarrow & var \\
Z & \rightarrow & \neg\ Z \\
Z & \rightarrow & \textbf{(}\ P\ \textbf{)}
\end{array}
$$

Infix with parens
with $\{\wedge_R\} < \{\vee_R\} < \{\neg\}$
using Extended BNF

$$
\begin{array}{lcl}
P & \rightarrow & A \\
A & \rightarrow & O\ (\wedge\ A)^? \\
O & \rightarrow & Z\ (\vee\ O)^? \\
Z & \rightarrow & var \\
Z & \rightarrow & \neg\ Z \\
Z & \rightarrow & \textbf{(}\ P\ \textbf{)}
\end{array}
$$

Infix with parens
with $\{\vee_L, \wedge_L\} < \{\neg\}$
using Extended BNF

$$
\begin{array}{lcl}
P & \rightarrow & OA \\
OA & \rightarrow & Z\ ((\wedge \mid \vee)\ Z)^* \\
Z & \rightarrow & var \\
Z & \rightarrow & \neg\ Z \\
Z & \rightarrow & \textbf{(}\ P\ \textbf{)}
\end{array}
$$

Infix with parens
with $\{\vee_R, \wedge_R\} < \{\neg\}$
using Extend BNF

$$
\begin{array}{lcl}
P & \rightarrow & OA \\
OA & \rightarrow & Z\ ((\wedge \mid \vee)\ Z)^? \\
Z & \rightarrow & var \\
Z & \rightarrow & \neg\ Z \\
Z & \rightarrow & \textbf{(}\ P\ \textbf{)}
\end{array}
$$

- For these grammars using Extended BNF, how would you construct the abstract parse tree?

4 $LL(k)$ Parsing

Recursive-descent parsing can be generalized to automatically generated table-driven top-down parsers, known as $LL(k)$ parsing algorithms.

- L : left-to-right parse
- L : leftmost derivation
- k : k-symbols lookahead

4.1 Nullable, First, and Follow

For a grammar $G = \langle \mathcal{N}, \mathcal{T}, S, \mathcal{P} \rangle$ we define the following properties:

$$
\begin{array}{lrcl}
A \in \mathcal{N} & \mathit{Nullable}(A) & = & \begin{cases} \textbf{true} & \text{if } A \Rightarrow^* \epsilon \\ \textbf{false} & \text{otherwise} \end{cases} \\
A \in \mathcal{N} & \mathit{First}(A) & = & \{\underline{a} \mid \underline{a} \in \mathcal{T} \text{ and } A \Rightarrow^* \underline{a}\beta\} \\
A \in \mathcal{N} & \mathit{Follow}(A) & = & \{\underline{a} \mid \underline{a} \in \mathcal{T} \text{ and } S \Rightarrow^* \beta A \underline{a} \gamma\} \\
 & & & \\
\underline{a} \in \mathcal{T} & \mathit{First}(\underline{a}) & = & \{\underline{a}\} \\
\underline{a} \in \mathcal{T} & \mathit{Nullable}(\underline{a}) & = & \textbf{true} \\
 & & & \\
\alpha \in (\mathcal{N} \cup \mathcal{T})^* & \mathit{Nullable}(\alpha) & = & \begin{cases} \textbf{true} & \text{if } \alpha \Rightarrow^* \epsilon \\ \textbf{false} & \text{otherwise} \end{cases}
\end{array}
$$

4.1.1 Computing Nullable, First, and Follow

Nullable

> **foreach** $\underline{a} \in \mathcal{T}$ **do**
> $\mathit{Nullable}(\underline{a}) \leftarrow$ **false**
> **end**
> **foreach** $A \in \mathcal{N}$ **do**
> $\mathit{Nullable}(A) \leftarrow$ **false**
> **end**
> **do**
>   **foreach** $A \rightarrow X_1 \cdots X_n \in \mathcal{P}$ **do**
>   **if** $\mathit{Nullable}(X_1)$ **and** $\cdots$ **and** $\mathit{Nullable}(X_n)$ (* *true if* $X_1 \cdots X_n = \epsilon$ *)
>   **then** $\mathit{Nullable}(A) \leftarrow$ **true**
>   **end**
> **until** $\mathit{Nullable}$ *does not change*

Note that we can easily extend $\mathit{Nullable}$ to sequences of terminals and non-terminals:

$$
\begin{array}{rcl}
\mathit{Nullable}(\epsilon) & = & \textbf{true} \\
\mathit{Nullable}(X\alpha) & = & \mathit{Nullable}(X) \wedge \mathit{Nullable}(\alpha)
\end{array}
$$

First

foreach $\underline{a} \in \mathcal{T}$ **do**
  $First(\underline{a}) \leftarrow \{\underline{a}\}$
end
foreach $A \in \mathcal{N}$ **do**
  $First(A) \leftarrow \{\}$
end
do
  foreach $A \rightarrow X_1 \cdots X_n \in \mathcal{P}$ **do**
    foreach $i \in \{1, \ldots, n\}$ **do**
      if $Nullable(X_1 \cdots X_{i-1})$
        then $First(A) \leftarrow First(A) \cup First(X_i)$
    end
  end
until *First does not change*

Note that we can easily extend $First$ to sequences of terminals and non-terminals:

$$\begin{array}{rcl} First(\epsilon) & = & \{\} \\ First(X\alpha) & = & \begin{cases} First(X) & \text{if } Nullable(X) = \textbf{false} \\ First(X) \cup First(\alpha) & \text{if } Nullable(X) = \textbf{true} \end{cases} \end{array}$$

Follow

foreach $A \in \mathcal{N}$ **do**
  $Follow(A) \leftarrow \{\}$
end
do
  foreach $A \rightarrow X_1 \cdots X_n \in \mathcal{P}$ **do**
    foreach $i \in \{1, \ldots, n\}$ **do**
      if $X_i \in \mathcal{N}$ **and** $Nullable(X_{i+1} \cdots X_n)$
        then $Follow(X_i) \leftarrow Follow(X_i) \cup Follow(A)$
      foreach $j \in \{i+1, \ldots, n\}$ **do**
        if $X_i \in \mathcal{N}$ **and** $Nullable(X_{i+1} \cdots X_{j-1})$
          then $Follow(X_i) \leftarrow Follow(X_i) \cup First(X_j)$
      end
    end
  end
until *Follow does not change*

4.1.2 Examples

Consider $Nullable$, $First$, and $Follow$ for ***Infix with parens with*** $\{\vee_L\} < \{\wedge_L\} < \{\neg\}$ ***without immediate left recursion***. The subscripts indicate the iteration in which the boolean was set or the symbol was added to the set.

	$Nullable$	$First$	$Follow$
S	$\textbf{false}_0$	$\{var_5, \neg_5, \textbf{(}_5\}$	$\{\}$
P	$\textbf{false}_0$	$\{var_4, \neg_4, \textbf{(}_4\}$	$\{\textbf{)}_1, \textbf{\$}_1\}$
O	$\textbf{false}_0$	$\{var_3, \neg_3, \textbf{(}_3\}$	$\{\textbf{)}_2, \textbf{\$}_2\}$
O'	$\textbf{true}_1$	$\{\vee_1\}$	$\{\textbf{)}_2, \textbf{\$}_2\}$
A	$\textbf{false}_0$	$\{var_2, \neg_2, \textbf{(}_2\}$	$\{\vee_1, \textbf{)}_2, \textbf{\$}_2\}$
A'	$\textbf{true}_1$	$\{\wedge_1\}$	$\{\vee_1, \textbf{)}_2, \textbf{\$}_2\}$
Z	$\textbf{false}_0$	$\{var_1, \neg_1, \textbf{(}_1\}$	$\{\vee_1, \wedge_2, \textbf{)}_3, \textbf{\$}_3\}$

Consider $Nullable$, $First$, and $Follow$ for ***Infix with parens without immediate left recursion***.

	$Nullable$	$First$	$Follow$
S	$\textbf{false}_0$	$\{var_2, \neg_2, \textbf{(}_2\}$	$\{\}$
P	$\textbf{false}_0$	$\{var_1, \neg_1, \textbf{(}_1\}$	$\{\wedge_1, \vee_1, \textbf{)}_1, \textbf{\$}_1\}$
P'	$\textbf{true}_1$	$\{\wedge_1, \vee_1\}$	$\{\wedge_2, \vee_2, \textbf{)}_2, \textbf{\$}_2\}$

Consider $Nullable$, $First$, and $Follow$ for ***Prefix***.

	$Nullable$	$First$	$Follow$
S			
P			

Consider $Nullable$, $First$, and $Follow$ for ***Postfix***.

	$Nullable$	$First$	$Follow$
S			
P			

Consider $Nullable$, $First$, and $Follow$ for ***Scheme-style prefix***.

	$Nullable$	$First$	$Follow$
S			
P			

Consider $Nullable$, $First$, and $Follow$ for ***Scheme-style postfix***.

	$Nullable$	$First$	$Follow$
S			
P			

4.2 $LL(1)$ Parse Tables

4.2.1 Computing $LL(1)$ Parse Tables

foreach $A \in \mathcal{N}$ **do**
    foreach $\underline{a} \in \mathcal{T}$ **do**
        $M[A, \underline{a}] \leftarrow \{\}$
    end
end
foreach $A \rightarrow X_1 \cdots X_n \in \mathcal{P}$ **do**
    if $Nullable(X_1 \cdots X_n)$ **then**
        foreach $\underline{a} \in Follow(A)$ **do**
            $M[A, \underline{a}] \leftarrow M[A, \underline{a}] \cup \{A \rightarrow X_1 \cdots X_n\}$
        end
    foreach $\underline{a} \in First(X_1 \cdots X_n)$ **do**
        $M[A, \underline{a}] \leftarrow M[A, \underline{a}] \cup \{A \rightarrow X_1 \cdots X_n\}$
    end
end

If any $M[A, \underline{a}]$ has more than one production, then the grammar is not $LL(1)$.

4.3 Examples

Consider the $LL(1)$ parse table for ***Infix with parens with*** $\{\vee_L\} < \{\wedge_L\} < \{\neg\}$ ***without immediate left recursion***.

	var	$\neg$	$\wedge$	$\vee$	**(**	**)**	**$**
S	$S \rightarrow P$ **$**	$S \rightarrow P$ **$**			$S \rightarrow P$ **$**		
P	$P \rightarrow O$	$P \rightarrow O$			$P \rightarrow O$		
O	$O \rightarrow A\ O'$	$O \rightarrow A\ O'$			$O \rightarrow A\ O'$		
O'				$O' \rightarrow \vee A\ O'$		$O' \rightarrow \epsilon$	$O' \rightarrow \epsilon$
A	$A \rightarrow Z\ A'$	$A \rightarrow Z\ A'$			$O \rightarrow Z\ A'$		
A'			$A' \rightarrow \wedge Z\ A'$	$A' \rightarrow \epsilon$		$A' \rightarrow \epsilon$	$A' \rightarrow \epsilon$
Z	$Z \rightarrow$ *var*	$Z \rightarrow \neg Z$			$Z \rightarrow$ **(** P **)**		

Consider the $LL(1)$ parse table for ***Infix with parens without immediate left recursion***.

	var	$\neg$	$\wedge$	$\vee$	**(**	**)**	**$**
S	$S \rightarrow P$ **$**	$S \rightarrow P$ **$**			$S \rightarrow P$ **$**		
P	$P \rightarrow$ *var* P'	$P \rightarrow \neg P'$			$P \rightarrow$ **(** P **)**		
P'			$P' \rightarrow \epsilon$ $P' \rightarrow \wedge\ P\ P'$	$P' \rightarrow \epsilon$ $P' \rightarrow \vee\ P\ P'$	$P' \rightarrow \epsilon$	$P' \rightarrow \epsilon$	

Consider the $LL(1)$ parse table for ***Scheme-style prefix***.

	var	$\neg$	$\wedge$	$\vee$	**(**	**)**	**$**
S							
P							

4.4 $LL(1)$ Parsing Algorithm

$stack \leftarrow []$ (* *empty stack* *)
$push(S, stack)$
while not $empty(stack)$ **do**
  $X \leftarrow pop(stack)$
  if $X \in \mathcal{T}$ **then**
    if $X == curTok()$ **then** $advanceTok()$ **else** $error()$
  else if $M[X, curTok()] = \{A \rightarrow Y_1 \cdots Y_n\}$ **then**
    $push(Y_n, stack)$ **;** $\cdots$ **;** $push(Y_1, stack)$
  else $error()$
end
$accept()$

4.5 $LL(1)$ Parsing Example

Suppose we wish to parse $a \wedge b \vee c$ in the ***Infix with parens with*** $\{\vee_L\} < \{\wedge_L\} < \{\neg\}$ ***without immediate left recursion*** grammar using the $LL(1)$ parsing algorithm.

Stack	Input	Action
S	$\dot{a} \wedge b \vee c$ **\$**	parse *S* → *P* **\$**
\$ *P*	$\dot{a} \wedge b \vee c$ **\$**	parse *P* → *O*
\$ *O*	$\dot{a} \wedge b \vee c$ **\$**	parse *O* → *A O'*
\$ *O' A*	$\dot{a} \wedge b \vee c$ **\$**	parse *A* → *Z A'*
\$ *O' A' Z*	$\dot{a} \wedge b \vee c$ **\$**	parse *Z* → *var*
\$ *O' A' var*	$\dot{a} \wedge b \vee c$ **\$**	consume *var* token
\$ *O' A'*	$\dot{\wedge}\, b \vee c$ **\$**	parse *A'* → ∧ *Z A'*
\$ *O' A' Z* ∧	$\dot{\wedge}\, b \vee c$ **\$**	consume ∧ token
\$ *O' A' Z*	$\dot{b} \vee c$ **\$**	parse *Z* → *var*
\$ *O' A' var*	$\dot{b} \vee c$ **\$**	consume *var* token
\$ *O' A'*	$\dot{\vee}\, c$ **\$**	parse *A'* → ϵ
\$ *O'*	$\dot{\vee}\, c$ **\$**	parse *O'* → ∨ *A O'*
\$ *O' A* ∨	$\dot{\vee}\, c$ **\$**	consume ∨ token
\$ *O' A*	$\dot{c}$ **\$**	parse *A* → *Z A'*
\$ *O' A' Z*	$\dot{c}$ **\$**	parse *Z* → *var*
\$ *O' A' var*	$\dot{c}$ **\$**	consume *var* token
\$ *O' A'*	$\dot{\$}$	parse *A'* → ϵ
\$ *O'*	$\dot{\$}$	parse *O'* → ϵ
\$	$\dot{\$}$	consume **\$** token
		accept

Note that the stack records what is to be parsed in the future.

4.6 $LL(k)$ for $k > 1$

To use more symbols of lookahead, we extend the definition of $First$ to $First_k$:

$$
\begin{array}{llll}
A \in \mathcal{N} & First_k(A) & = & \{\underline{a}_1 \cdots \underline{a}_k \mid \{\underline{a}_1, \ldots, \underline{a}_k\} \subseteq \mathcal{T} \text{ and } A \Rightarrow^* \underline{a}_1 \cdots \underline{a}_k \beta\} \\
 & & \cup & \{\underline{a}_1 \cdots \underline{a}_j \mid j < k \text{ and } \{\underline{a}_1, \ldots, \underline{a}_j\} \subseteq \mathcal{T} \text{ and } A \Rightarrow^* \underline{a}_1 \cdots \underline{a}_j\}
\end{array}
$$

Consider $First_2$ for ***Scheme-style prefix***.

	$First_2$
S	$\{var_1,\ \mathtt{(}\neg_1,\ \mathtt{(}\wedge_1,\ \mathtt{(}\vee_1\}$
P	$\{var_1,\ \mathtt{(}\neg_1,\ \mathtt{(}\wedge_1,\ \mathtt{(}\vee_1\}$

Consider the $LL(2)$ parse table for ***Scheme-style prefix***.

	var	$\mathtt{(}\neg$	$\mathtt{(}\wedge$	$\mathtt{(}\vee$	otherwise
S	$S \rightarrow P\ \mathtt{\$}$	$S \rightarrow P\ \mathtt{\$}$	$S \rightarrow P\ \mathtt{\$}$	$S \rightarrow P\ \mathtt{\$}$	
P	$P \rightarrow var$	$P \rightarrow \mathtt{(}\ \neg\ P\ \mathtt{)}$	$P \rightarrow \mathtt{(}\ \wedge\ P\ P\ \mathtt{)}$	$P \rightarrow \mathtt{(}\ \vee\ P\ P\ \mathtt{)}$	

Consider $First_2$ for ***Scheme-style postfix with parens***.

	$First_2$
S	$\{var_1,\ \mathtt{(\,(}_1\}$
P	$\{var_1,\ \mathtt{(\,(}_1\}$

Consider the $LL(2)$ parse table for ***Scheme-style postfix with parens***.

	var	$\mathtt{(\,(}$	otherwise
S	$S \rightarrow P\ \mathtt{\$}$	$S \rightarrow P\ \mathtt{\$}$	
P	$P \rightarrow var$	$P \rightarrow \mathtt{(}\ P\ \neg\ \mathtt{)}$ $P \rightarrow \mathtt{(}\ P\ P\ \wedge\ \mathtt{)}$ $P \rightarrow \mathtt{(}\ P\ P\ \vee\ \mathtt{)}$ $P \rightarrow \mathtt{(}\ P\ \mathtt{)}$	