LL(1) Grammars

LL(k)

Predictive parsing that does not need backtracking. The first **L** stands for Left-to-right and the second **L** stands for Leftmost and **k** indicates the maximum number of lookahead symbols needed to take a decision

Most programming constructs can be expressed using an $LL(1)$ grammar. A grammar G is $LL(1)$ iff whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:

1. for no terminal a do both α and β derive strings beginning with a
2. at most one of α and β can derive the empty string
3. if $\beta \rightarrow^* \epsilon$, then α does not derive any string starting with a terminal in $FOLLOW(A)$. Likewise if $\alpha \rightarrow^* \epsilon$, then β does not derive any string starting with a terminal in $FOLLOW(A)$

LL(1) Grammars

LL(k)

Predictive parsing that does not need backtracking. The first **L** stands for Left-to-right and the second **L** stands for Leftmost and **k** indicates the maximum number of lookahead symbols needed to take a decision

Most programming constructs can be expressed using an $LL(1)$ grammar. A grammar G is $LL(1)$ iff whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:

1. for no terminal a do both α and β derive strings beginning with a
2. at most one of α and β can derive the empty string
3. if $\beta \rightarrow^* \epsilon$, then α does not derive any string starting with a terminal in $FOLLOW(A)$. Likewise if $\alpha \rightarrow^* \epsilon$, then β does not derive any string starting with a terminal in $FOLLOW(A)$

Parsing table

Derive *FIRST*, *FOLLOW*, *nullable* sets and parsing table for the following grammar:

$$S \to iEtSS' | a \quad S' \to eS | \epsilon \quad E \to b$$

Parsing table:

	a	b	e	i	t	$\$$
S	$S \to a$			$S \to iEtSS'$		
S'			$S' \to \epsilon$ $S' \to eS$			$S' \to \epsilon$
E		$E \to b$				

Parsing table

Derive *FIRST*, *FOLLOW*, *nullable* sets and parsing table for the following grammar:

$$S \to iEtSS'|a \quad S' \to eS|\epsilon \quad E \to b$$

Parsing table:

	a	b	e	i	t	$
S	$S \to a$			$S \to iEtSS'$		
S'			$S' \to \epsilon$ $S' \to eS$			$S' \to \epsilon$
E		$E \to b$				

Non-recursive predictive parsing

Table-driven predictive parsing

```
**Input:** A string $w$ and a parsing table M for grammar $\mathcal{G}$
**Output:** if $w$ is in $\mathscr{L}(\mathcal{G})$, a leftmost derivation of $w$, otherwise an error indication
set $ip$ to pint to the first symbol of $w$;
set $X$ to the top stack symbol;
while ($X \neq \$$) do
    if ($X$ is $a$) then pop the stack and advnce $ip$;
    else if ($X$ is a terminal) then error();
    else if ($M[X,a]$ is an error entry) then error();
    else if ($M[X,a] = X \to Y_1 Y_2 \cdots Y_k$) then c
        output the production $X \to Y_1 Y_2 \cdots Y_k$;
        pop the stack;
        push $Y_k Y_{k-1} \cdots Y_1$ onto the stack, with $Y_1$ on top;
    end if
    Set $X$ to the top stack symbol;
end while
```

LL(1) parser moves (1/2)

MATCHED	STACK	INPUT	ACTION
	E$	**id** $+$ **id** $*$ **id**$	

LL(1) parser moves (2/2)

MATCHED	STACK	INPUT	ACTION
	S$	ibtibtaea$	

Error Recovery in Predictive Parsing

Error detection

An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol, and $M[A,a]$ is ERROR.

Error Recovery in Predictive Parsing

Error detection

An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol, and $M[A,a]$ is ERROR.

Panic Mode

Based on the idea of skipping symbols on the input until a token in a synchronizing set appears. Strategies:

- place all symbols in $FOLLOW(A)$ into the synchronizing set for nonterminal A.
- symbols starting higher level constructs
- use of ϵ-productions to change the symbol in the stack
- just pop the symbol in the stack and send alert

Error Recovery in Predictive Parsing

Error detection

An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol, and $M[A,a]$ is ERROR.

Phrase-level recovery

Fill the blank entries in the predictive parsing table with entries to recovery routines.

ToC

Bottom-up Parsing

Bottom-up Parsing

The problem of Bottom-up parsing can be viewed as the problem of constructing a parse tree for an input string beginning at the leaves and working up towards the root.
In particular we will consider the problem of finding the rightmost derivation given an input string, through a series of reductions to reach the initial symbol

Let's consider the input string **id** $*$ **id** using the simple grammar for expressions

Bottom-up Parsing

Bottom-up Parsing

The problem of Bottom-up parsing can be viewed as the problem of constructing a parse tree for an input string beginning at the leaves and working up towards the root.
In particular we will consider the problem of finding the rightmost derivation given an input string, through a series of reductions to reach the initial symbol

Let's consider the input string **id** $*$ **id** using the simple grammar for expressions

Tools for Bottom-up Parsing

Reductions

In a bottom-up parser at each step a reduction is applied. A certain string is reduced to the head of the production (non-terminal) applying the production in reverse. The key decision is when to reduce!

Handle Pruning

A handle is a substring of a sentential form that matches the body of a production and whose reduction represents a step along the rightmost derivation of the sentential form in reverse.

Consider the grammar $S \rightarrow 0S1|01$ and the two sentential forms 000111, $00S11$

Tools for Bottom-up Parsing

Reductions

In a bottom-up parser at each step a reduction is applied. A certain string is reduced to the head of the production (non-terminal) applying the production in reverse. The key decision is when to reduce!

Handle Pruning

A handle is a substring of a sentential form that matches the body of a production and whose reduction represents a step along the rightmost derivation of the sentential form in reverse.

Consider the grammar $S \rightarrow 0S1|01$ and the two sentential forms 000111, $00S11$

Shift-reduce parsing

Shift-reduce parsing

A shift-reduce parser is a particular kind of bottom-up parser in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed. Four possible actions are possible:

- shift
- reduce
- accept
- error

Conflicts

- shift/reduce
- reduce/reduce

Shift-reduce parsing

Shift-reduce parsing

A shift-reduce parser is a particular kind of bottom-up parser in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed. Four possible actions are possible:

- shift
- reduce
- accept
- error

Conflicts

- shift/reduce
- reduce/reduce

Shift-reduce parsing

Consider the grammar $S \rightarrow SS + | SS * | a$ and the following sentential forms:
$SSS + a * +$, $SS + a * a+$, $aaa * a + +$

LR Parsing

LR Parsers

LR parsers show interesting good properties:

- all programming languages admit a grammar that can be parsed by an LR parser
- most general non-backtracking shift-reduce parser
- syntactic errors can be detected as soon as it is possible to do so on a left-to right scan of the input
- the class of grammars that can be parsed by an LR is a proper superset of that parsable with a predictive parsing strategy

Items and LR(0) Automaton

Item

An Item is a production in which a dot $\cdot$ has been added in the body. Intuitively, it indicates how much of a production we have seen during parsing.
One collection of sets of $LR(0)$ items, called the canonical $LR(0)$ collection, provides the basis for constructing a DFA that is used to make decisions.
The construction of the canonical $LR(0)$ is based on two functions CLOSURE and GOTO

CLOSURE

If $\mathcal{I}$ is a set of items for a grammqr $\mathcal{G}$, then $\mathrm{CLOSURE}(\mathcal{I})$ is the set of items constructed from $\mathcal{I}$ by the following two rules:

1. Initially, add every item in $\mathcal{I}$ to $\mathrm{CLOSURE}(\mathcal{I})$
2. if $A \to \alpha \cdot B\beta$ is in $\mathrm{CLOSURE}(\mathcal{I})$ and $B \to \gamma$ is a production, then add the item $B \to \cdot\gamma$ to $\mathrm{CLOSURE}(\mathcal{I})$, if is not already there. Apply this rule until no more items can be added to $\mathrm{CLOSURE}(\mathcal{I})$

CLOSURE

Consider the expression grammar:
$E' \to E \quad E \to E + T | T \quad T \to T * F | F \quad F \to (E) | id$
Compute the closure of the item $E' \to \cdot E$

GOTO

GOTO(I,X)

GOTO(I,X) is defined to be the closure of the set of all items $[A \rightarrow \alpha X \cdot \beta]$ such that $[A \rightarrow \alpha \cdot X\beta]$ is in I.

- Intuitively the GOTO function is used to define the transition of the LR(0) automaton for a grammar. The states of the automaton correspond to sets of items, and GOTO(I,X) specifies the transition from the state for I under input X

Build the LR(0) automaton

Build the LR(0)automaton for the expression grammar:
$E' \rightarrow E \quad E \rightarrow E + T | T \quad T \rightarrow T * F | F \quad F \rightarrow (E) | id$