Front End: Syntax Analysis

Bottom-Up Parsing

Parsers

- Top-down
  Construct leftmost derivations starting from the start symbol.
- Bottom-up
  Construct (reverse) rightmost derivations starting from the input string by reducing it to the start symbol.

Both parsers are guided by the input string in the search of a derivation.

Bottom-up Parsing

Intuition: construct the parse tree from the leaves to the root.

Example:

Grammar:

$S \rightarrow AB$

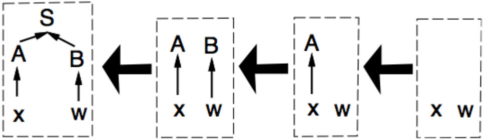
$A \rightarrow x \mid Y$

$B \rightarrow w \mid Z$

$Y \rightarrow xb$

$Z \rightarrow wp$

Input xw**.**



Bottom-up parsing

Constructing a parse tree for an input string starting from the leaves towards the root.

Example II

$$\begin{array}{rcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

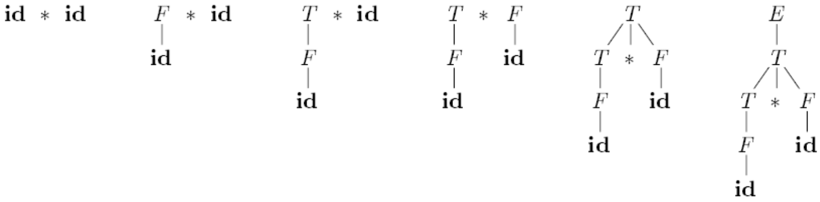


Figure 4.25: A bottom-up parse for $\mathbf{id} * \mathbf{id}$

Shift-Reduce Parsing

A shift-reduce parser is a form of bottom-up parser whose primary operations are

- **Shift**: shift the next input symbol
- **Reduce**: identify the *handle* and replace it with the head of the appropriate production.

A *reduction step* is the reverse of a derivation step (= a non-terminal is replaced by the body of one of its productions). Thus, reducing corresponds to constructing a derivation in reverse.

Example: The parse in Fig. 4.25 corresponds to the rightmost derivation

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

Handle

A handle is a substring that matches the body of a production.

Right Sentential Form	Handle	Reducing Production
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$T \to T \ * \ F$
T	T	$E \to T$

Figure 4.26: Handles during a parse of $\mathbf{id}_1 * \mathbf{id}_2$

Handle: Formal Definition

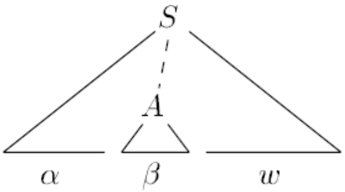


Figure 4.27: A handle $A \to \beta$ in the parse tree for $\alpha\beta w$

Definition

A handle for $\gamma = \alpha\beta w$ s.t. $S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \gamma$ is

1. a production rule $A \to \beta$, and
2. a position p in γ where β can be located.

Stack Implementation

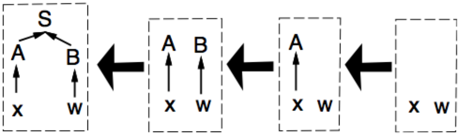
A stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.

Stack	Input	Action
$\$$	$\mathbf{id}_1 * \mathbf{id}_2\ \$$	shift
$\$\ \mathbf{id}_1$	$* \mathbf{id}_2\ \$$	reduce by $F \to \mathbf{id}$
$\$\ F$	$* \mathbf{id}_2\ \$$	reduce by $T \to F$
$\$\ T$	$* \mathbf{id}_2\ \$$	shift
$\$\ T *$	$\mathbf{id}_2\ \$$	shift
$\$\ T * \mathbf{id}_2$	$\$$	reduce by $F \to \mathbf{id}$
$\$\ T * F$	$\$$	reduce by $T \to T * F$
$\$\ T$	$\$$	reduce by $E \to T$
$\$\ E$	$\$$	accept

Figure 4.28: Configurations of a shift-reduce parser on input $\mathbf{id}_1 * \mathbf{id}_2$

Example

STACK	INPUT	ACTION
$\$$	$xw\$$	shift
$\$x$	$w\$$	reduce by $A \to x$
$\$A$	$w\$$	shift
$\$Aw$	$\$$	reduce by $B \to w$
$\$AB$	$\$$	reduce by $S \to AB$
$\$S$	$\$$	accept



$$S \underset{rm}{\Longrightarrow} AB \underset{rm}{\Longrightarrow} Aw \underset{rm}{\Longrightarrow} xw.$$

Conflict I: Shift-Reduce

There are CF grammars for which the shift-reduce parsing does not work.

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

With the following configuration we cannot decide whether to shift or to reduce.

STACK	**INPUT**
$\$ \ldots$ **if** *expr* **then** *stmt*	**else** $\ldots$

Conflict II: Reduce-Reduce

Consider a language where procedures and arrays share the same syntax.

(1)	$stmt$	$\rightarrow$	$\textbf{id}$ ($parameter\_list$)
(2)	$stmt$	$\rightarrow$	$expr$:= $expr$
(3)	$parameter\_list$	$\rightarrow$	$parameter\_list$, $parameter$
(4)	$parameter\_list$	$\rightarrow$	$parameter$
(5)	$parameter$	$\rightarrow$	$\textbf{id}$
(6)	$expr$	$\rightarrow$	$\textbf{id}$ ($expr\_list$)
(7)	$expr$	$\rightarrow$	$\textbf{id}$
(8)	$expr\_list$	$\rightarrow$	$expr\_list$, $expr$
(9)	$expr\_list$	$\rightarrow$	$expr$

Figure 4.30: Productions involving procedure calls and array references

Which production should we choose with configuration

STACK	INPUT
\$ . . . **id (id**	**id, id)** . . .

LR Parsing

LR(k) parsing, introduce by D. Knuth in 1965, is today the most prevalent type of bottom-up parsing.
L is for *L*eft-to-right scanning of the input,
R is for reverse *R*ightmost derivation,
k is the number of lookahead tokens.

Different types:

- Simple LR or SLR, the easiest method for constructing shift-reduce parsers,
- Canonical LR,
- LALR.

The last two types are used in the majority of LR parsers.

The LR-Parsing Model

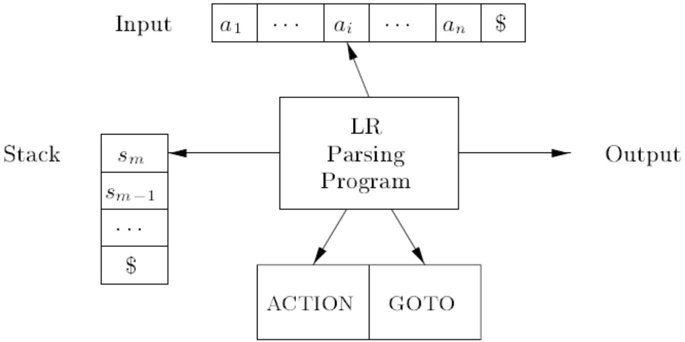


Figure 4.35: Model of an LR parser

The parsing table, consisting of the ACTION and GOTO functions, is the only variable part. The stack content is a sequence of states, corresponding each to a grammar symbol.

The LR-Parsing Algorithm

All LR-parsers behave as summarised below: the only difference is the info held by the parsing table.

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

Figure 4.36: LR-parsing program

Conflict Resolution

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$\,\mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \to \mathbf{id}$
$\$\,F$	$* \mathbf{id}_2 \$$	reduce by $T \to F$
$\$\,T$	$* \mathbf{id}_2 \$$	shift
$\$\,T *$	$\mathbf{id}_2 \$$	shift
$\$\,T * \mathbf{id}_2$	$\$$	reduce by $F \to \mathbf{id}$
$\$\,T * F$	$\$$	reduce by $T \to T * F$
$\$\,T$	$\$$	reduce by $E \to T$
$\$\,E$	$\$$	accept

Figure 4.28: Configurations of a shift-reduce parser on input $\mathbf{id}_1 * \mathbf{id}_2$

How does a shift-reduce parser know when to shift and when to reduce?

Example:

In Fig. 4.28, how does the parser know that T is not yet a handle and that the appropriate action is a *shift*?

Constructing LR-Parsing Table

LR parsers are table-driven, similarly to the non-recursive LL parsers.

In order to recognise the right-hand side of a production, an LR parser must be able to recognise handles of right sentential forms when they appear on top of the stack.

Idea: maintaining states to keep track of where we are in a parse can help an LR parser to decide when to shift and when to reduce.

Construct a Finite Automaton.

The SLR method constructs a parsing table on the base of LR(0) items and LR(0) automata.

Items

Definition

An LR(0) item (or simply item) of a grammar G is a production of G with a dot at some position of the body.

Example:
For the production $A \to XYZ$ we get the items

$$\begin{array}{lcl} A & \to & \bullet XYZ \\ A & \to & X \bullet YZ \\ A & \to & XY \bullet Z \\ A & \to & XYZ \bullet \end{array}$$

A state in our FA is a set of items.

Closure

Given a set of items I, the closure of I is computed as follows:

```
SetOfItems CLOSURE(I) {
        J = I;
        repeat
                for ( each item A → α·Bβ in J )
                        for ( each production B → γ of G )
                                if ( B → ·γ is not in J )
                                        add B → ·γ to J;
        until no more items are added to J on one round;
        return J;
}
```

Figure 4.32: Computation of CLOSURE

Intuition: if item $A \to \alpha \bullet B\beta$ is in CLOSURE(I), then at some point the parser might see a substring derivable from $B\beta$ as input.

Example

E' → E
E → E + T | T
T → T* F | F
F → (E) | **id**

If **I = { E' → •E }**,
then CLOSURE(I) is → ?

Example

$E' \to E$
$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to (E) \mid \textbf{id}$

If $I = \{ E' \to \cdot E \}$,
then CLOSURE(I) is

$E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \textbf{id}$

The GOTO Function

If $A \to \alpha \bullet X\beta$ is in I, GOTO(I, X) contains CLOSURE$(A \to \alpha \bullet X\beta)$.

```
void items(G') {
    C = {CLOSURE({[S' → ·S]})};
    repeat
        for ( each set of items I in C )
            for ( each grammar symbol X )
                if ( GOTO(I, X) is not empty and not in C )
                    add GOTO(I, X) to C;
    until no new sets of items are added to C on a round;
}
```

Figure 4.33: Computation of the canonical collection of sets of LR(0) items

$$G = \begin{array}{l} E' \rightarrow E \\ E \rightarrow E + T \mid T \\ T \rightarrow T^* F \mid F \\ F \rightarrow (E) \mid \textbf{id} \end{array}$$

$$\mathbf{I} = \begin{array}{l} \mathbf{E' \rightarrow E \bullet} \\ \mathbf{E \rightarrow E \bullet + T} \end{array}$$

GOTO(I,+) = ?

$$G = \begin{array}{l} E' \to E \\ E \to E + T \mid T \\ T \to T^* F \mid F \\ F \to (E) \mid \mathbf{id} \end{array}$$

$$\mathbf{I} = \begin{array}{l} \mathbf{E' \to E \bullet} \\ \mathbf{E \to E \bullet + T} \end{array}$$

If $[A \to \boldsymbol{\alpha} \bullet X\ \boldsymbol{\beta}\ ] \in \mathbf{I}$,
GOTO(I, X) contains
CLOSURE($A \to \boldsymbol{\alpha} X \bullet \boldsymbol{\beta}$)

$$\mathbf{GOTO(I,+)} = \begin{array}{|l|} \hline E \to E + .\ T \\ T \to .\ T * F \\ T \to .\ F \\ F \to .\ (E) \\ F \to .\ \mathbf{id} \\ \hline \end{array}$$

The LR(0) Automaton

G' : augmented grammar

LR(0) automaton for G'

$\langle \mathbf{Q}, \mathbf{q_0}, \mathbf{GOTO}: Q \times (T_{G'} \cup N_{G'}) \rightarrow Q, \mathbf{F} \rangle$

where:

$\mathbf{Q} = \mathbf{F} = \text{items}(G')$,

$\mathbf{q_0} = \text{CLOSURE}(\{S' \rightarrow \bullet S\})$

Example

Construction of the LR(0) automaton for the augmented grammar:

$$E' \rightarrow E$$
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \mathbf{id}$$

E' → E

E → E + T | T

T → T* F | F

F → (E) | **id**