Shift-Reduce Parsers

Shift-reduce parsers require the following data structures.

1. a buffer for holding the input string to be parsed.

Shift-Reduce Parsers

Shift-reduce parsers require the following data structures.

1. a buffer for holding the input string to be parsed.
2. a data structure for detecting handles (a stack happens to be adequate)

Shift-Reduce Parsers

Shift-reduce parsers require the following data structures.

1. a buffer for holding the input string to be parsed.
2. a data structure for detecting handles (a stack happens to be adequate)
3. a data structure for storing and accessing the lhs and rhs of rules.

Shift-Reduce Parsers

Basic actions of the shift-reduce parser are:

Shift: Moving a single token from the input buffer onto the stack till a handle appears on the stack.

Shift-Reduce Parsers

Basic actions of the shift-reduce parser are:

Shift: Moving a single token from the input buffer onto the stack till a handle appears on the stack.

Reduce: When a handle appears on the stack, it is popped and replaced by the left hand side of the corresponding production.

Shift-Reduce Parsers

Basic actions of the shift-reduce parser are:

Shift: Moving a single token from the input buffer onto the stack till a handle appears on the stack.

Reduce: When a handle appears on the stack, it is popped and replaced by the left hand side of the corresponding production.

Accept: When the stack contains only the start symbol and input buffer is empty, the parser halts announcing a *successful* parse.

Shift-Reduce Parsers

Basic actions of the shift-reduce parser are:

Shift: Moving a single token from the input buffer onto the stack till a handle appears on the stack.

Reduce: When a handle appears on the stack, it is popped and replaced by the left hand side of the corresponding production.

Accept: When the stack contains only the start symbol and input buffer is empty, the parser halts announcing a *successful* parse.

Error: When the parser can neither shift nor reduce nor accept. Halts announcing an error.

Properties of shift-reduce parsers

Is the following situation possible?

- $\alpha \; \beta \; \gamma$ is the stack contents and $A \rightarrow \gamma$ is the handle.
- The stack contents reduces to $\alpha \; \beta \; A$
- Now $B \rightarrow \beta$ is the next handle.

Implication: The handle is buried in the stack. The search for the handle can be expensive.

Properties of shift-reduce parsers

Is the following situation possible?

- $\alpha\ \beta\ \gamma$ is the stack contents and $A \rightarrow \gamma$ is the handle.
- The stack contents reduces to $\alpha\ \beta\ A$
- Now $B \rightarrow \beta$ is the next handle.

Implication: The handle is buried in the stack. The search for the handle can be expensive.

Assume that this is true. Then, by the definition of a handle, there is a sequence of rightmost derivations:

$$S \overset{*rm}{\Rightarrow} \alpha BAxyz \overset{rm}{\Rightarrow} \alpha\beta Axyz \overset{rm}{\Rightarrow} \alpha\beta\gamma xyz$$

But in the right sentential form $\alpha BAxyz$, B is not the rightmost non-terminal, and thus $\overset{rm}{\Rightarrow}$ is not a rightmost derivation. Therefore the above scenario is not possible.

Properties of shift-reduce parsers

So what scenarios are possible after a reduction?

$$\alpha\beta\gamma xyz$$

Properties of shift-reduce parsers

So what scenarios are possible after a reduction?

$$\begin{tikzcd}\alpha\beta A \;\; xyz \arrow[d, Rightarrow, "rm"] \\ \alpha\beta\gamma xyz\end{tikzcd}$$

Production used is $A \rightarrow \gamma$

Properties of shift-reduce parsers

So what scenarios are possible after a reduction?

$$\begin{tikzcd}
\alpha C \;\; xyz \arrow[dr, Rightarrow, "rm"] & \\
& \alpha\beta A \;\; xyz \arrow[d, Rightarrow, "rm"] \\
& \alpha\beta\gamma xyz
\end{tikzcd}$$

Production used is $C \rightarrow \beta A$

Properties of shift-reduce parsers

So what scenarios are possible after a reduction?

$$\begin{tikzcd}
\alpha C \;\; xyz \arrow[dr, Rightarrow, "rm"] & \alpha D \;\; yz \arrow[d, Rightarrow, "rm"] \\
& \alpha\beta A \;\; xyz \arrow[d, Rightarrow, "rm"] \\
& \alpha\beta\gamma xyz
\end{tikzcd}$$

Production used is $D \to \beta A x$

Properties of shift-reduce parsers

So what scenarios are possible after a reduction?

$$\begin{tikzcd}
\alpha C \quad xyz \arrow[dr, Rightarrow, "rm"] & \alpha D \quad yz \arrow[d, Rightarrow, "rm"] & \alpha\beta AxE \quad z \arrow[dl, Rightarrow, "rm"'] \\
 & \alpha\beta A \quad xyz \arrow[d, Rightarrow, "rm"] & \\
 & \alpha\beta\gamma xyz &
\end{tikzcd}$$

Production used is $E \rightarrow y$

Example of Shift-Reduce Parsing

Ambiguous grammar of expressions

Conflicts in a Shift-Reduce Parser

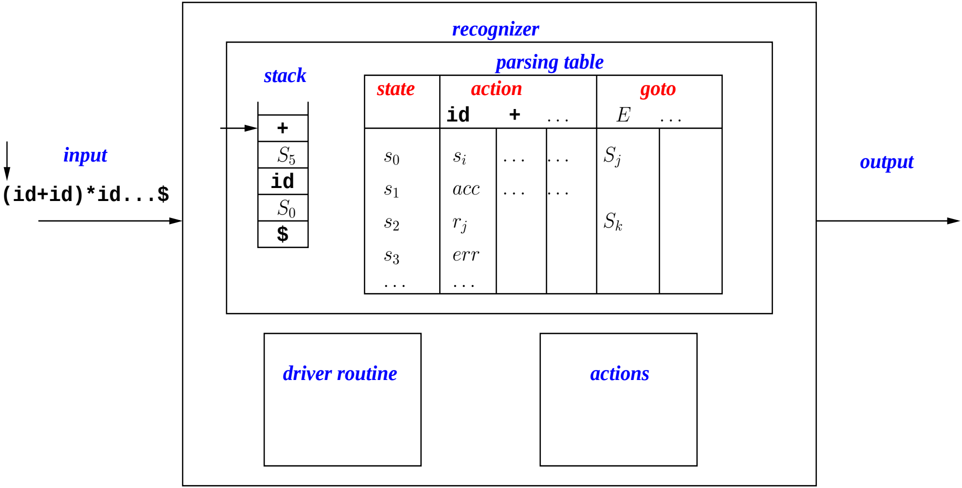
For some grammars, the shift-reduce parser may get into the following conflicting situations.

- *Shift-reduce conflict* A handle β occurs at *tos*; the *nexttoken* a is such that $\beta \mathtt{a} \gamma$ happens to be another handle. The parser has two options
  - reduce the handle using $A \rightarrow \beta$
  - ignore the handle; shift a and continue parsing and eventually reduce using some rule $B \rightarrow \beta \mathtt{a} \gamma$.
- *Reduce-reduce conflict* the stack contents are $\alpha\beta\gamma$ and both $\beta\gamma$ and γ are handles with $A \rightarrow \beta\gamma$ and $B \rightarrow \gamma$ as the corresponding rules. Then the parser has two reduce possibilities.

To handle such conflicts, the nexttoken could be used to prefer one move over the other.

- choose shift (or reduce) in a shift-reduce conflict
- prefer one reduce (over others) in a reduce-reduce conflict.

LR Parser Model



state	action			goto	
	id	+	...	E	...
s_0	s_i	...	...	S_j	
s_1	acc	...	...		
s_2	r_j			S_k	
s_3	err				
...	...				

LR Parsers

Consist of

- a stack which contains strings of the form $s_0 X_1 s_1 X_2 \ldots X_m s_m$, where X_i is a grammar symbol and s_i is a special symbol called a *state*.
- a parsing table which comprises two parts, usually named as *Action* and *Goto*.

The entries in the Action part are:

- s_i which means shift to state i
- r_j which stands for reduce by the j^{th} rule,
- *accept*
- *error*

The Goto part contains blank entries or state symbols.

The Driver Routine

- Initializes stack with *start* state. Calls scanner to get next token.
- Consults the parsing table and performs the action specified there.
- Parsing continues till either an error or accept entry is encountered.

top of stack	*nexttoken*	*action*	*parsing action*
state j	a	si	push a; push state i
	a	rj	$rj : A \rightarrow \alpha$; $length(\alpha) = r$; pop 2r symbols from stack; top of stack contains state k; goto[k, a] $= cl$; push A; push state l;
state j	\$	acc	successful parse; halt
state j	a	err	error handling

SLR(1) Parser

1.	E	$\rightarrow$	$E + T$	2.	E	$\rightarrow$	T
3.	T	$\rightarrow$	$T * F$	4.	T	$\rightarrow$	F
5.	F	$\rightarrow$	(E)	6.	F	$\rightarrow$	id

	action						*goto*		
state	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			c1	c2	c3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			c8	c2	c3
5		r6	r6		r6	r6			
6	s5			s4				c9	c3
7	s5			s4					c10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			