بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۱۲)

کامپایلر

حسین فلسفین

Shift-Reduce Parsing

☞ *Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed. As we shall see, the handle always appears at the top of the stack just before it is identified as the handle.*

☞ *There are many different approaches to bottom-up parsing but most of the practical and widely used implementations are based on shift-reduce parsers. These parsers perform a single left-to-right pass over the input without any backtracking.*

☞ *The parser has a stack and an input. The first k tokens of the input are the lookahead. Based on the contents of the stack and the lookahead, the parser performs two kinds of actions: Shift: Move the first input token to the top of the stack. Reduce: Choose a grammar rule $X \rightarrow ABC$; pop C, B, A from the top of the stack; push X onto the stack.*

STACK	INPUT
$	w $

During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string β of grammar symbols on top of the stack. It then reduces β to the head of the appropriate production. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty:

STACK	INPUT
$ S	$

Upon entering this configuration, the parser halts and announces successful completion of parsing.

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

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

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

While the primary operations are shift and reduce, there are actually four possible actions a shift-reduce parser can make: (1) shift, (2) reduce, (3) accept, and (4) error.

1. *Shift. Shift the next input symbol onto the top of the stack.*
2. *Reduce. The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.*
3. *Accept. Announce successful completion of parsing.*
4. *Error. Discover a syntax error and call an error recovery routine.*

The use of a stack in shift-reduce parsing is justified by an important fact: the handle will always eventually appear on top of the stack, never inside. The parser never had to go into the stack to find the handle.

Conflicts During Shift-Reduce Parsing

There are context-free grammars for which shift-reduce parsing cannot be used. Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack and also the next k input symbols (the symbols of lookahead on the input), cannot decide whether to shift or to reduce (a shift/reduce conflict), or cannot decide which of several reductions to make (a reduce/reduce conflict).

Some examples of syntactic constructs for which shift-reduce parsing cannot be used: Example 4.38 and Example 4.39

Introduction to LR Parsing: Simple LR

Here, we introduce the basic concepts of LR parsing, and the easiest method for constructing shift-reduce parsers, called "simple LR" (or SLR, for short). Some familiarity with the basic concepts is helpful even if the LR parser itself is constructed using an automatic parser generator.

Why LR Parsers?

LR parsers are table-driven, much like the nonrecursive LL parsers of Section 4.4.4. A grammar for which we can construct a parsing table is said to be an LR grammar. Intuitively, for a grammar to be LR it is sufficient that a left-to-right shift-reduce parser be able to recognize handles of right-sentential forms when they appear on top of the stack.

LR parsing is attractive for a variety of reasons:

1. *LR parsers can be constructed to recognize virtually all programming language constructs for which CFGs can be written. Non-LR CFGs exist, but these can generally be avoided for typical programming language constructs.*
2. *The LR-parsing method is the most general nonbacktracking shift-reduce parsing method known, yet it can be implemented as efficiently as other, more primitive shift-reduce methods.*
3. *An LR parser can detect a syntactic error as soon as it is possible*

to do so on a left-to-right scan of the input.

4. The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive or LL methods. For a grammar to be $\mathrm{LR}(k)$*, we must be able to recognize the occurrence of the right side of a production in a right-sentential form, with* k *input symbols of lookahead. This requirement is far less stringent than that for* $\mathrm{LL}(k)$ *grammars where we must be able to recognize the use of a production seeing only the first* k *symbols of what its right side derives. Thus, it should not be surprising that LR grammars can describe more languages than LL grammars.*

Items and the $\mathrm{LR}(0)$ Automaton

How does a shift-reduce parser know when to shift and when to reduce? For example, with stack contents $\$T$ and next input symbol $*$ in Fig. 4.28, how does the parser know that T on the top of the stack is not a handle, so the appropriate action is to shift and not to reduce T to E?

An LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in a parse. *States represent sets of "items."*

$\mathrm{LR}(0)$ *items*

An $\mathrm{LR}(0)$ ***item (item for short) of a grammar*** G ***is a production of*** G ***with a dot at some position of the body. Thus, production*** $A \to XYZ$ ***yields the four items***

$$A \to \bullet XYZ$$
$$A \to X \bullet YZ$$
$$A \to XY \bullet Z$$
$$A \to XYZ \bullet$$

The production $A \to \varepsilon$ ***generates only one item,*** $A \to \bullet$.

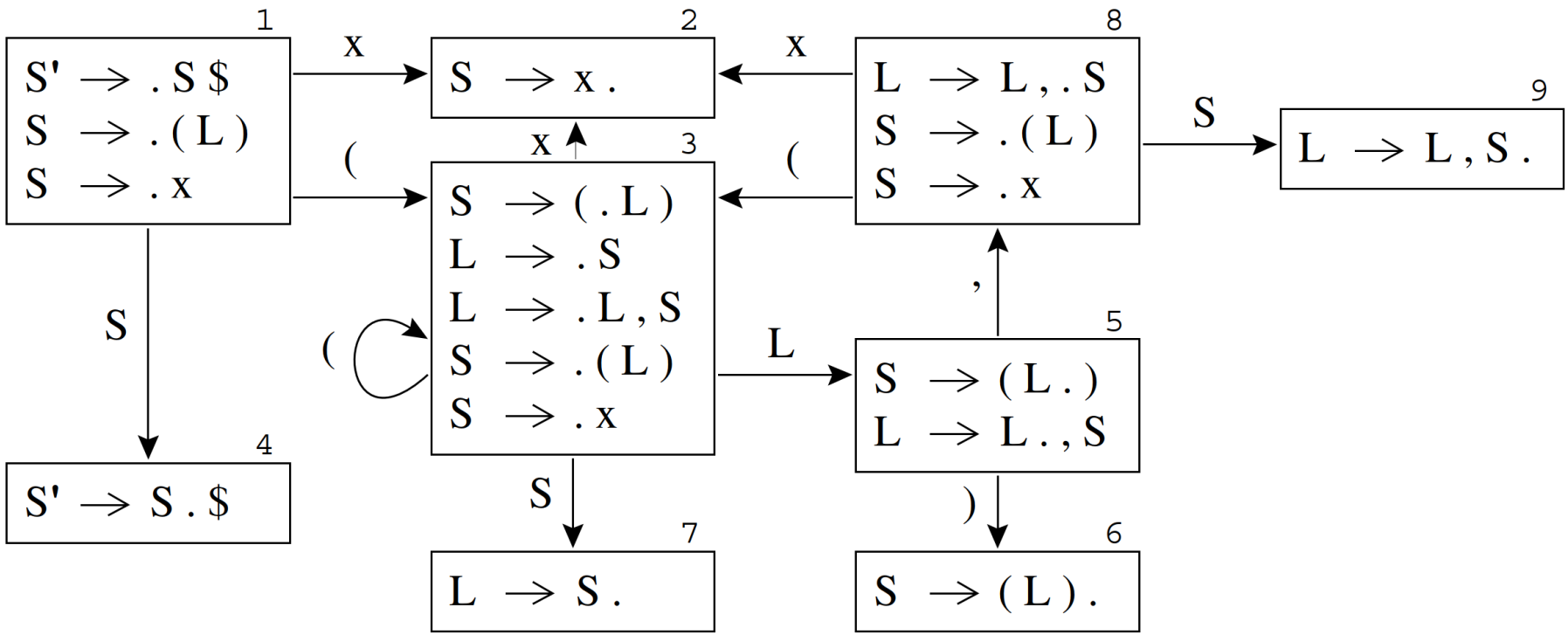
Intuitively, an item indicates how much of a production we have seen at a given point in the parsing process.

A grammar rule, combined with the dot that indicates a position in its right-hand side, is called an item (specifically, an $\mathrm{LR}(0)$ *item). A state is just a set of items.*

Example

The item $A \to \bullet XYZ$ ***indicates that we hope to see a string derivable from*** XYZ ***next on the input. Item*** $A \to X \bullet YZ$ ***indicates that we have just seen on the input a string derivable from*** X ***and that we hope next to see a string derivable from*** YZ***. Item*** $A \to XYZ \bullet$ ***indicates that we have seen the body*** XYZ ***and that it may be time to reduce*** XYZ ***to*** A***.***

One collection of sets of $\mathrm{LR}(0)$ ***items, called the canonical*** $\mathrm{LR}(0)$ ***collection, provides the basis for constructing a deterministic finite automaton that is used to make parsing decisions. Such an automaton is called an*** $\mathrm{LR}(0)$ ***automaton. In particular, each state of the*** $\mathrm{LR}(0)$ ***automaton represents a set of items in the canonical*** $\mathrm{LR}(0)$ ***collection.***



☞ ***The central idea behind "Simple LR," or SLR, parsing is the construction from the grammar of the*** $\mathrm{LR}(0)$ ***automaton. The states of this automaton are the sets of items from the canonical*** $\mathrm{LR}(0)$ ***collection, and the transitions are given by the*** GOTO ***function.***

☞ ***The start state of the*** $\mathrm{LR}(0)$ ***automaton is***

$$\mathrm{CLOSURE}(\{[S' \to \bullet S]\}),$$

where S' ***is the start symbol of the augmented grammar.***

☞ ***All states are accepting states.***

CLOSURE *and* GOTO

To construct the canonical $\mathrm{LR}(0)$ ***collection for a grammar, we define an augmented grammar and two functions,***

CLOSURE *and* GOTO.

If G ***is a grammar with start symbol*** S***, then*** G'***, the augmented grammar for*** G***, is*** G ***with a new start symbol*** S' ***and production*** $S' \rightarrow S$***. The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input. That is, acceptance occurs when and only when the parser is about to reduce by*** $S' \rightarrow S$***.***

$\mathrm{CLOSURE}$ *of Item Sets*

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

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

☞ ***Intuitively,*** $A \to \alpha \bullet B\beta$ ***in*** $\mathrm{CLOSURE}(I)$ ***indicates that, at some point in the parsing process, we think we might next see a substring derivable from*** B ***as input.***

☞ ***The substring derivable from*** B ***will have a prefix derivable from*** B ***by applying one of the*** B***-productions. We therefore add items for all the*** B***-productions; that is, if*** $B \to \gamma$ ***is a production, we also include*** $B \to \bullet\gamma$ ***in*** $\mathrm{CLOSURE}(I)$.

☞ $\mathrm{CLOSURE}$ ***adds more items to a set of items when there is a dot to the left of a nonterminal;***

Example: *If I is the set of one item $\{[E' \to \bullet E]\}$, then $\text{CLOSURE}(I)$ contains the set of items I_0 in Fig. 4.31.*

I_0

$E' \to \cdot E$

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (\ E \ )$

$F \to \cdot \mathbf{id}$

$$
\begin{array}{lcl}
E' & \to & E \\
E & \to & E + T \mid T \\
T & \to & T * F \mid F \\
F & \to & (E) \mid \mathbf{id}
\end{array}
$$

SetOfItems CLOSURE(I) {
    $J = I$;
    repeat
        for (each item $A \to \alpha \cdot B \beta$ in J)
            for (each production $B \to \gamma$ of G)
                if ($B \to \cdot\gamma$ is not in J)
                    add $B \to \cdot\gamma$ to J;
    until no more items are added to J on one round;
    return J;
}

Closure(I) =

repeat

for any item $A \rightarrow \alpha.X\beta$ in I

for any production $X \rightarrow \gamma$

$I \leftarrow I \cup \{X \rightarrow .\gamma\}$

until I does not change.

return I