بسم اللّه الرّحمن الرّحیم

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

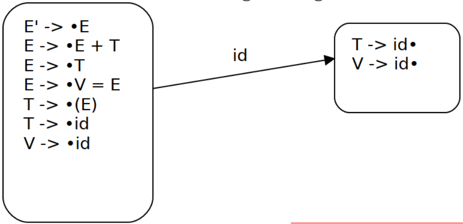
# کامپایلر

حسین فلسفین

## مثالی دیگر از یک گرامر که LR(0) نیست اما SLR(1) هست

```
E' -> E
E  -> E + T  | T | V = E
T  -> (E) | id
V  -> id
```

Here are the first two LR(0) configurating sets entered if id is the first token of the input.

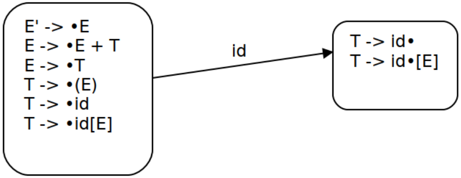


In an LR(0) parser, the set on the right has a reduce-reduce conflict. However, an SLR(1) parser will compute Follow(T) = { + ) $ } and Follow(V) = { = } and thus can distinguish which reduction to apply depending on the next input token. The modified grammar is SLR(1).

## مثالی دیگر از یک گرامر که LR(0) نیست اما SLR(1) هست

```
E' -> E
E  -> E + T  | T
T  -> (E) | id | id[E]
```

Here are the first two LR(0) configurating sets entered if id is the first token of the input.



In an LR(0) parser, the set on the right has a shift-reduce conflict. However, an SLR(1) will compute Follow(T) = { + ) ] $ } and only enter the reduce action on those tokens. The input [ will shift and there is no conflict. Thus this grammar is SLR(1) even though it is not LR(0).

## *More Powerful LR Parsers*

*We shall extend the previous LR parsing techniques to use one symbol of lookahead on the input. There are two different methods:*

*1. The "canonical-LR" or just "LR" method, which makes full use of the lookahead symbol(s). This method uses a large set of items, called the* $\mathrm{LR}(1)$ *items.*

*2. The "lookahead-LR" or "*$\mathrm{LALR}$*" method, which is based on the* $\mathrm{LR}(0)$ *sets of items, and has many fewer states than typical parsers based on the* $\mathrm{LR}(1)$ *items. By carefully introducing lookaheads into the* $\mathrm{LR}(0)$ *items, we can handle many more grammars with the* $\mathrm{LALR}$ *method than with the* $\mathrm{SLR}$ *method, and build parsing tables that are no bigger than the* $\mathrm{SLR}$ *tables.* $\mathrm{LALR}$ *is the method of choice in most situations.*

## $\mathrm{LR}(1)$ *items;* $\mathrm{LR}(1)$ *parsing table*

***Even more powerful than $\mathrm{SLR}$ is the $\mathrm{LR}(1)$ parsing algorithm. Most programming languages whose syntax is describable by a context-free grammar have an $\mathrm{LR}(1)$ grammar. The algorithm for constructing an $\mathrm{LR}(1)$ parsing table is similar to that for $\mathrm{LR}(0)$, but the notion of an item is more sophisticated.***

***An $\mathrm{LR}(1)$ item consists of a grammar production, a right-hand-side position (represented by the dot), and a lookahead symbol.***

***The idea is that an item $[A \to \alpha \bullet \beta, x]$ indicates that the sequence $\alpha$ is on top of the stack, and at the head of the input is a string derivable from $\beta x$.***

## *Canonical* $\mathrm{LR}(1)$ *Items*

***We shall now present the most general technique for constructing an LR parsing table from a grammar. Recall that in the*** $\mathrm{SLR}$ ***method, state*** $i$ ***calls for reduction by*** $A \to \alpha$ ***if the set of items*** $I_i$ ***contains item*** $[A \to \alpha\bullet]$ ***and input symbol*** $a$ ***is in*** $\mathrm{FOLLOW}(A)$***. In some situations, however, when state*** $i$ ***appears on top of the stack, the viable prefix*** $\beta\alpha$ ***on the stack is such that*** $\beta A$ ***cannot be followed by*** $a$ ***in any right-sentential form. Thus, the reduction by*** $A \to \alpha$ ***should be invalid on input*** $a$***.***

*It is possible to carry more information in the state that will allow us to rule out some of these invalid reductions by $A \to \alpha$. By splitting states when necessary, we can arrange to have each state of an LR parser indicate exactly which input symbols can follow a handle $\alpha$ for which there is a possible reduction to $A$.*

*The extra information is incorporated into the state by redefining items to include a terminal symbol as a second component. The general form of an item becomes $[A \to \alpha \bullet \beta, a]$, where $A \to \alpha\beta$ is a production and $a$ is a terminal or the right endmarker $\$$. We call such an object an $\mathrm{LR}(1)$ item. The 1 refers to the length of the second component, called the lookahead of the item.*

*Lookaheads that are strings of length greater than one are possible, of course, but we shall not consider such lookaheads here.*

***The $\mathrm{LR}(1)$ technique does not rely on $\mathrm{FOLLOW}$ sets, but rather keeps the specific lookahead with each item. We will write an $\mathrm{LR}(1)$ item thus: $[A \to \alpha \bullet \beta, S]$, in which $S$ is the set of tokens that can follow this specific item. When the dot has reached the end of the item, as in $[A \to \alpha\beta\bullet, S]$, the item is an acceptable reduce item only if the lookahead at that moment is in $S$; otherwise the item is ignored.***

## *The* $\mathrm{LR}(1)$ *Canonical Collection*

***The*** $\mathrm{LR}(1)$ ***parsing tables,*** $\mathrm{ACTION}$ ***and*** $\mathrm{GOTO}$***, for a grammar*** $G$ ***are derived from a DFA for recognizing the possible handles for a parse in*** $G$***. This DFA is constructed from what is called an*** $\mathrm{LR}(1)$ ***canonical collection, in turn a collection of sets of items of the form***

$$[Y \to \alpha \bullet \beta, a]$$

***where*** $Y \to \alpha\beta$ ***is a production rule in the set of productions*** $P$***,*** $\alpha$ ***and*** $\beta$ ***are (possibly empty) strings of symbols, and*** $a$ ***is a lookahead. The item represents a potential handle. The*** $\bullet$ ***is a position marker that marks the top of the stack, indicating that we have parsed the*** $\alpha$ ***and still have the*** $\beta$ ***ahead of us in satisfying the*** $Y$***. The lookahead symbol,*** $a$***, is a token that can follow*** $Y$ ***(and so,*** $\alpha\beta$***) in a legal right-most derivation of some sentence.***

☞ *If the position marker comes at the start of the right-hand side in an item, $[Y \to \bullet\alpha\beta, a]$ the item is called a possibility. One way of parsing the $Y$ is to first parse the $\alpha$ and then parse the $\beta$, after which point the next incoming token will be an $a$. The parse might be in the following configuration:*

*Stack:* $\$\gamma$ *Input:* $ua\ldots$

*where $\alpha\beta \Rightarrow^* u$, where $u$ is a string of terminals.*

☞ *If the position marker comes after a string of symbols $\alpha$ but before a string of symbols $\beta$ in the right-hand side in an item, $[Y \to \alpha \bullet \beta, a]$ the item indicates that $\alpha$ has been parsed (and so is on the stack) but that there is still $\beta$ to parse from the input:*

*Stack:* $\$\gamma\alpha$ *Input:* $va\ldots$

*where $\beta \Rightarrow^* v$, where $v$ is a string of terminals.*

☞ *If the position marker comes at the end of the right-hand side in an item,* $[Y \to \alpha\beta \bullet, a]$ *the item indicates that the parser has successfully parsed* $\alpha\beta$ *in a context where* $Ya$ *would be valid, the* $\alpha\beta$ *can be reduced to a* $Y$*, and so* $\alpha\beta$ *is a handle. That is, the parse is in the configuration*

**Stack:** $\$\gamma\alpha\beta$ **Input:** $a \ldots$

*and the reduction of* $\alpha\beta$ *would cause the parser to go into the configuration*

**Stack:** $\$\gamma Y$ **Input:** $a \ldots$

## مجموعهٔ $a$های نظیر آیتم‌های $[A \to \alpha\bullet, a]$ همواره زیرمجموعهٔ $\mathrm{FOLLOW}(A)$ است

***The lookahead has no effect in an item of the form $[A \to \alpha \bullet \beta, a]$, where $\beta$ is not $\varepsilon$, but an item of the form $[A \to \alpha\bullet, a]$ calls for a reduction by $A \to \alpha$ only if the next input symbol is $a$. Thus, we are compelled to reduce by $A \to \alpha$ only on those input symbols $a$ for which $[A \to \alpha\bullet, a]$ is an $\mathrm{LR}(1)$ item in the state on top of the stack. The set of such $a$'s will always be a subset of $\mathrm{FOLLOW}(A)$, but it could be a proper subset, as in Example 4.51.***

## *Computing the Closure of a Set of Items*

```
SetOfItems CLOSURE($I$) {
    repeat
        for ( each item $[A \to \alpha \cdot B\beta, a]$ in $I$ )
            for ( each production $B \to \gamma$ in $G'$ )
                for ( each terminal $b$ in $\text{FIRST}(\beta a)$ )
                    add $[B \to \cdot\gamma, b]$ to set $I$;
    until no more items are added to $I$;
    return $I$;
}
```

```
Closure($I$) =
  repeat
    for any item $(A \to \alpha.X\beta, z)$ in $I$
        for any production $X \to \gamma$
            for any $w \in \text{FIRST}(\beta z)$
                $I \leftarrow I \cup \{(X \to .\gamma,\ w)\}$
  until $I$ does not change
  return $I$
```

## *Computing* GOTO

```
SetOfItems GOTO($I, X$) {
        initialize $J$ to be the empty set;
        for ( each item $[A \to \alpha \cdot X\beta, a]$ in $I$ )
                add item $[A \to \alpha X \cdot \beta, a]$ to set $J$;
        return CLOSURE($J$);
}
```

```
Goto($I, X$) =
  $J \leftarrow \{\}$
  for any item $(A \to \alpha.X\beta,\ z)$ in $I$
      add $(A \to \alpha X.\beta,\ z)$ to $J$
  return Closure($J$).
```

## *Computing the* $\mathrm{LR}(1)$ *Collection*

```
void items(G') {
    initialize C to {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;
}
```