CSCI 3130: Formal Languages and Automata Theory — Lecture 15
The Chinese University of Hong Kong, Fall 2010 — Andrej Bogdanov

In Lecture 9 we introduced the Cocke-Younger-Kasami algorithm for parsing strings in CFGs. The advantage of the CYK algorithm is that it works for every CFG (once we have eliminates loops, nullable variables, and converted the CFG to Chomsky Normal Form). However, the CYK algorithm is too slow to be used for parsing the code of computer programs.

In the last lecture we showed a much faster family of parsing algorithms: LR(0) parsers. These parsers work extremely fast, but only apply to a restricted class of CFGs which we call LR(0) grammars. Unfortunately, very few interesting CFGs are LR(0). In particular, most CFGs of programming languages are not LR(0). To illustrate this point, we look at the CFG of the java programming language, which contains the following productions:

$$\text{Statement} \to \texttt{if}\ \text{ParExpression Statement}$$
$$\text{Statement} \to \texttt{if}\ \text{ParExpression Statement}\ \texttt{else}\ \text{Statement}.$$

After processing a segment of code like "`if (n == 0) break;`", and LR(0) parser would encounter a shift/reduce conflict: the first production suggests a reduce, while the second one suggests a shift. A simple engineering solution immediately comes to mind: If the parser could peek at the next token and see if it is the keyword `else`, it could resolve this particular shift/reduce conflict.

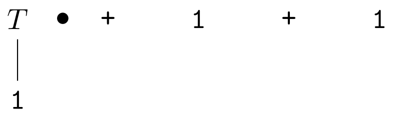
For simplicity, we will assume that the CFG does not contain any nullable variables. (If nullable variables exist, we can eliminate them first.)

1 LR(1) items

An LR(1) parser works in a similar way as an LR(0) parser, except that it is allowed to peek at the next input symbol to resolve pending shift/reduce and reduce/reduce conflicts. Before we explain LR(1) parsing, let's see an example where peeking ahead can help. Consider the following CFG:

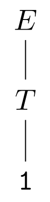
$$E \to T \mid T\ \texttt{+}\ E$$
$$T \to \texttt{1}.$$

This CFG generates expressions like `1+1+1`. Let's see what happens when we try to parse this using an LR(0) parser. Initially, the valid LR(0) items are $E \to \bullet T, E \to \bullet T\texttt{+}E, T \to \bullet\texttt{1}$. All these are shift items, so after reading the first `1`, the input becomes $\texttt{1} \bullet \texttt{+1+1}$, and the only valid item left is $T \to \texttt{1}\bullet$, so we reduce the input to $\bullet T\texttt{+1+1}$. After shifting on T we construct part of the parse tree

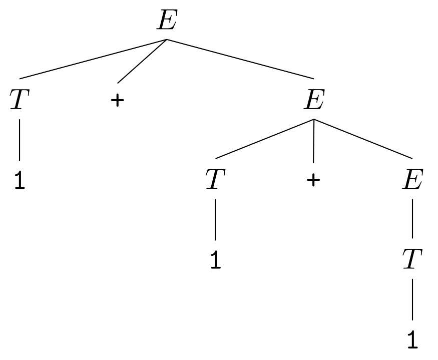


and we are left with two valid items: $E \to T\bullet, E \to T \bullet \texttt{+}E$. This is a shift/reduce conflict.

Peeking at the next item in the input, we see that it is a `+`. This is inconsistent with the first item: If this item was the correct one, it would have led to the following parse tree:



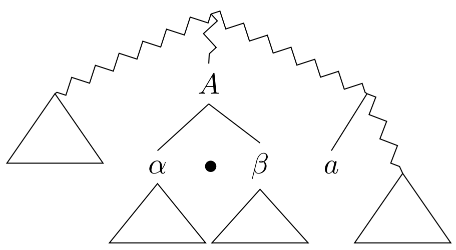
Clearly this is the wrong parse tree, since the first terminal `1` is not followed by a `+`. So by peeking one symbol ahead and a little bit of reasoning, we managed to discard the item $E \to T\bullet$ and we are left with the unique shift item $E \to T \bullet +E$. Continuing in this manner, we can construct the unique parse tree:

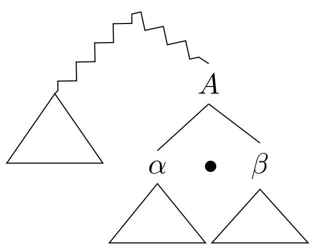


To carry out this parsing procedure more systematically, it will help to extend the definition of "item". An LR(0) item describes a portion of the parse tree together with the position of the input within this portion. In LR(1) parsing, we sometimes we need to look one symbol ahead. For this purpose, it helps to record the symbol that comes after the current portion of the parse tree inside the item.

Definition 1. An *LR(1) item* of a CFG G is a string of the form $A \to [\alpha \bullet \beta, a]$, where $A \to \alpha\beta$ is a production in G, and a is a terminal of G or the special symbol ε.

The items $A \to [\alpha \bullet \beta, a]$ and $A \to [\alpha \bullet \beta, \varepsilon]$ indicate the following position of the $\bullet$ in the parse tree, respectively:





2 LR(1) update rules

Just like in the LR(0) algorithm, we first want to construct an NFA N_G that tells us how items will be updated on shift transitions. The states of N_G are all LR(1) items of G, plus a special start state q_0. We now describe the transitions. The initial setup and the "shift update" rule are essentially the same as for LR(0) parsers:

Initial setup: For every production of the form $S \to \alpha$, where S is the start variable, put an ε-transition from q_0 to the state $S \to [\bullet\alpha, \varepsilon]$ in N_G.

NFA shift update For every variable or terminal X, every production of the form $B \to \alpha a \beta$ in G, and every b that is a terminal or ε, put a transition from $B \to [\alpha \bullet a \beta, b]$ to $B \to [\alpha a \bullet \beta, b]$ with label a.

Recall that in the LR(0) updates, whenever we encounter an item of the form $B \to \alpha \bullet C\beta$, we also need to provide ε-transitions to all items of the form $C \to \bullet\delta$. In the LR(1) setting, what are the analogous ε-transitions from items of the form $B \to [\alpha \bullet C\beta, b]$?

To answer this question, let's go back to our example CFG $E \to T \mid T \texttt{ + } E, T \to \texttt{1}$. Consider the LR(1) item $E \to [\bullet T\texttt{+}E, \varepsilon]$. Looking at the variable T on the right hand side, we see that the derivation has to then use the production: $T \to \bullet\texttt{1}$. However, this is an LR(0) item, not an LR(1) item. To turn it into an LR(1) item, we need to figure out which terminal in the input can appear *after* we have completed the subtree that comes from the production $T \to \texttt{1}$. Since the next symbol after T is a +, the following symbol will always be a +, so we put an ε-transition from $E \to [\bullet T\texttt{+}E, \varepsilon]$ to the LR(1) item $T \to [\bullet\texttt{1}, \texttt{+}]$ in N_G.

In general, things can be more complicated. Consider, for instance an item of the form $A \to [\bullet BC, a]$. To figure out which terminal comes after we have completed the subtree rooted at B, we need to know what is the first symbol in C! There can be many such possibilities, and we need to include them all.

Definition 2. Let A be a variable in CFG G that is not nullable. The *first set* $F(A)$ of A is the set of all terminals that can occur as the leftmost leaf in a parse tree rooted at A.

In our example, the first set of E is $F(E) = \{\texttt{1}\}$, and the first set of T is also $F(T) = \{\texttt{1}\}$, since the leftmost symbol in all strings derived from E or T is 1. It will be useful to extend the definition of "first set" not only to variables, but also to arbitrary strings with variables and terminals in them.

Definition 3. Assume G does not have any nullable variables. Let $X_1 \dots X_k$ be a string of variables and terminals in G. The *first set* $F(X_1 \dots X_k)$ of $X_1 \dots X_k$ is

$$F(X_1 \dots X_k) = \begin{cases} F(X_1), & \text{if } X_1 \text{ is a variable} \\ \{X_1\}, & \text{if } X_1 \text{ is a terminal} \\ \{\varepsilon\}, & \text{if } X = \varepsilon. \end{cases}$$

We can now describe the last rule for LR(1) updates:

NFA branch update For every variable or terminal X, every production of the form $B \to \alpha C\beta$ in G, and every b that is a terminal or ε, put a transition from $B \to [\alpha \bullet C\beta, b]$ to $B \to [\alpha C \bullet \beta, x]$ with label ε for every $x \in F(\beta b)$.

Applying the construction to the CFG G, we obtain the following NFA N_G:

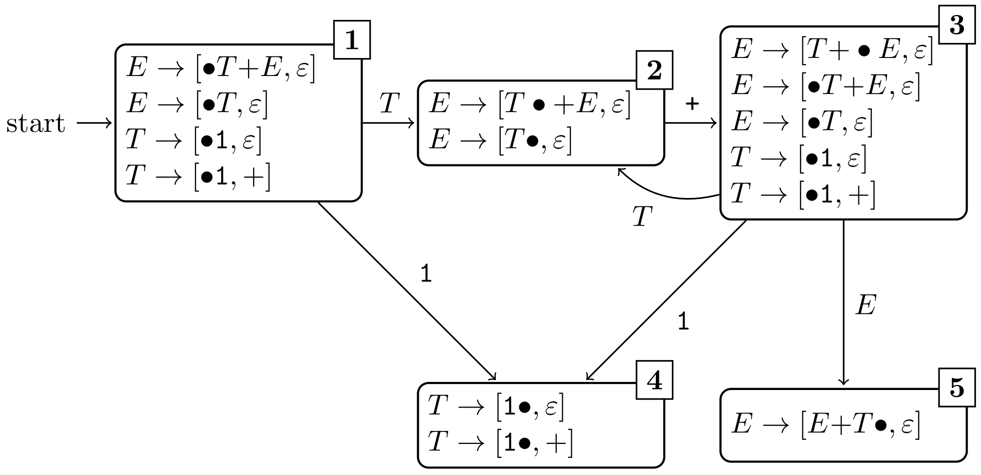
$$\begin{tikzcd}
 & \boxed{E \to [T\bullet, \varepsilon]} & \boxed{E \to [\bullet T, \varepsilon]} \arrow[l, "T"'] \arrow[r, "\varepsilon"] & \boxed{T \to [\bullet\mathtt{1}, \varepsilon]} \arrow[r, "+"] & \boxed{T \to [\mathtt{1}\bullet, \varepsilon]} \\
\text{start} \arrow[r] & q_0 \arrow[ur, "\varepsilon"] \arrow[d, "\varepsilon"] & & & \\
 & \boxed{E \to [\bullet T\texttt{+}E, \varepsilon]} \arrow[r, "T"] \arrow[d, "\varepsilon"] & \boxed{E \to [T \bullet \texttt{+}E, \varepsilon]} \arrow[r, "\texttt{+}"] & \boxed{E \to [T\texttt{+} \bullet E, \varepsilon]} \arrow[r, "E"] \arrow[uu, "\varepsilon"] \arrow[ll, bend right, "\varepsilon"'] & \boxed{E \to [T\texttt{+}E\bullet, \varepsilon]} \\
 & \boxed{T \to [\bullet\mathtt{1}, \texttt{+}]} \arrow[r, "+"] & \boxed{T \to [\mathtt{1}\bullet, \texttt{+}]} & &
\end{tikzcd}$$

Computing the first sets In working out the above examples, the first sets were easy to compute by hand. For more complicated CFGs, it is useful to have an algorithm for computing these sets. The idea of the algorithm is simple: If a production starts with a terminal, like $A \to \mathtt{a}B$, we include the terminal in the first set of the left-hand side variable. If a production starts with a variable, let $A \to B\mathtt{c}$, then we include every terminal that is in the first set of B also in the first set of A. When there are no more terminals to include, the first sets are completed.

```
On input $G$, where $G$ is a CFG without nullable variables:
    Initially, for every variable $A$, set $F(A) := \varnothing$.
    For every variable $A$ and production $A \to a\alpha$, where $a$ is a terminal:
        Set $F(A) := F(A) \cup \{a\}$.
    While there exists a variable $A$ and production $A \to B\alpha$
    and a terminal $a$ such that $a \in F(B)$ but $a \notin F(A)$:
        Set $F(A) := F(A) \cup \{a\}$.
```

3 Converting the NFA to a DFA: Shift/reduce conflicts

Now that we have the LR(1) NFA N_G set up, we can convert it to a DFA D_G, just like we did for LR(0) grammars. Applying the NFA to DFA conversion rules to N_G, we obtain the following DFA D_G. For clarity we omit the state $\varnothing$ from the diagram.



start
1
$E \to [\bullet T{+}E, \varepsilon]$
$E \to [\bullet T, \varepsilon]$
$T \to [\bullet 1, \varepsilon]$
$T \to [\bullet 1, +]$
T
2
$E \to [T \bullet {+}E, \varepsilon]$
$E \to [T\bullet, \varepsilon]$
+
3
$E \to [T{+} \bullet E, \varepsilon]$
$E \to [\bullet T{+}E, \varepsilon]$
$E \to [\bullet T, \varepsilon]$
$T \to [\bullet 1, \varepsilon]$
$T \to [\bullet 1, +]$
T
1
1
E
4
$T \to [1\bullet, \varepsilon]$
$T \to [1\bullet, +]$
5
$E \to [E{+}T\bullet, \varepsilon]$

Unlike the LR(0) DFAs we saw in the last lecture, this DFA contains a shift/reduce conflict. State 2 contains both a shift item and a reduce item, causing a possible conflict. However, this conflict can be resolved by looking at the next symbol in the input: In state 2 if the next symbol is a `+`, we shift to state 3; if the end of the string has been reached, we reduce according to the item $E \to [T\bullet, \varepsilon]$.

4 Parsing LR(1) grammars

We will say that an item $A \to [\alpha \bullet \beta, a]$ is *consistent* with x if either $\beta \neq \varepsilon$ and x is the first symbol in β, or $\beta = \varepsilon$ and $x = a$.

Definition 4. We say G is an LR(1) grammar if in every state q of the DFA D_G (except for the dead state) and every x that is either a terminal or ε, either all items in q that are consistent with x are incomplete, or there is at most one item in q consistent with x and it is complete.

If G is an LR(1) grammar, looking at the next input symbol can always resolve shift/reduce conflicts. The PDA that performs the LR(1) parsing is similar as for LR(0) grammars, except that it employs lookahead to resolve potential conflicts:

States: The states of P_G are the same as the states of D_G. In addition, P_G has a special start state $\bullet S$ and a special final state $S\bullet$. There are no other final states.

Transitions: The PDA D_G has the following transitions:

- For every state q and every transition $q \xrightarrow{X} q'$ of D_G (except those going to the dead state of D_G), P_G has a transition from q to q' with the action `read` a, `push` q.
- For every state q, if the next input symbol is x (we set $x = \varepsilon$ if the end of input is reached) and q has a unique reduce item of the form $A \to [X_1 \dots X_k\bullet, a]$ that is consistent with x, P_G has a transition going out of q with the following sequence of actions: `pop` q_k, ..., `pop` q_1, `go to state` q_1. Replace the portion of the input containing the pattern $X_1 \dots X_k\bullet$ with $\bullet A$ and construct part of the parse tree by connecting A to $X_1, \dots, X_k$.

- In addition, P_G has the special transitions `push $` from state $\bullet S$ to the start state of D_G and `pop $` from the start state of D_G to the final state $S\bullet$.

The parser generator To summarize our discussion of LR(1) parsing, we give a description of the LR(1) parser generator, which combines all the steps we discussed:

```
On input $G$, where $G$ is a CFG without nullable variables:
    Compute the first sets $F(A)$ for every variable $A$ in $G$.
    Construct the NFA $N_G$.
    Convert $N_G$ to a DFA $D_G$.
    If for some $x$ that is a terminal or $\varepsilon$, some state of $D_G$ has items
    that are consistent with $x$ but have a shift/reduce or reduce/reduce conflict,
        Output the error message "$G$ is not an LR(1) grammar" and halt.
    Otherwise, construct and output the PDA $P_G$.
```