West Chester University
Dr. Richard Burns
Fall 2023

CSC 416/565: DESIGN AND CONSTRUCTION OF COMPILERS

LAST CLASS

Predictive Parsing
Algorithm: Recursive Descent

TODAY

- Left Recursion

- *Motivation:* Recursive Descent parsing algorithm does not work with grammars that are left recursive.

Recursive Descent parsing algorithm does not work with grammars that are left recursive.

Unable to use this grammar
w/ recursive descent:

```
S → E$
E → E + T | E - T | T
T → T * F | T / F | F
F → id | num | (E)
```

Why? Infinite loop

```
boolean E_1() { return E() && advance(PLUS) && T(); }
boolean E() { ... ; return E_1() || ... }
```

Definition: A left-recursive grammar has some non-terminal S, such that $S \rightarrow^{+} S\alpha$, where α is any combination of zero or more terminals or non-terminals.

- A production is left recursive if its LHS symbol is also the first symbol of its RHS.
- $\rightarrow^{+}$ means *one or more* derivation steps...
- "...eventually derives to”

It is still possible to make derivations with a grammar that is left recursive.

- Often easier to understand left-recursive grammars than those modified to not be left-recursive.

Example

Grammar:

$S \rightarrow S\alpha \mid \beta$

- What is the language defined by this left-recursive grammar?
- Rewrite <u>left-recursive grammar</u> so that it uses <u>right recursion</u> instead.

A general rule to eliminate left recursion

- A left-recursive grammar has the form:

$$X \rightarrow X\gamma$$
$$X \rightarrow \alpha$$

- where α does not start with X

- All strings derived from grammar will start with α, and continue with zero or more γ.

- Can rewrite using right-recursion as:

$$X \rightarrow \alpha X'$$
$$X' \rightarrow \gamma X' \mid \varepsilon$$

Example: *expression-term* grammar

Left-recursive grammar:

$E \rightarrow E + T$

$E \rightarrow T$

Right-recursive grammar:

Can work for *multiple* productions

- <u>Left recursive:</u>

$N \rightarrow N\ \alpha_1\ |\ N\ \alpha_2 \ldots\ |\ N\ \alpha_m\ |\ \beta_1\ |\ \beta_2\ |\ \ldots\ |\ \beta_n$

- <u>Transformation, introducing new nonterminal N’</u>

$N\ \ \rightarrow \beta_1\ N'\ |\ \beta_2\ N'\ |\ \ldots\ |\ \beta_n\ N'$
$N' \rightarrow \alpha_1\ N'\ |\ \alpha_2\ N'\ |\ \ldots\ |\ \alpha_m\ N'\ |\ \varepsilon$

- More examples in [Thain]

Example

- Removing left-recursion that spans *multiple* productions is trickier.

$\mathtt{S} \rightarrow \mathtt{A}\ \alpha \mid \gamma$
$\mathtt{A} \rightarrow \mathtt{S}\ \beta$

- left-recursive because: $\mathtt{S} \rightarrow^{+} \mathtt{S}\beta\alpha$
- ... more complicated general algorithms exist

Predictive Parsing

- *New Idea:* in predictive parsing, parser can *predict* which production to use, simply by looking at the next few tokens:
- No backtracking! (as we had in Recursive Descent in the last class)
- Works only when the next few tokens provide *enough information* about which production to use.
  - no more choices to make
  - bad/incorrect choices not possible

Predictive Parsing

Predictive parsers accept `LL(k)` class of grammars:

Will parse the input:

- left-to-right ("first L")
- leftmost derivation ("second L")
- using k "tokens of lookahead"
  - *k=1*: if we can look at next token and always know which production to use

Example

Grammar:

```
E → T | T + E
T → int | int * T | (E)
```

Some issues:

- With 1 symbol of lookahead, we may know that our next token is `<int>`, but not enough information to choose which `T` production rule to apply.
- Sometimes referred to as <u>prediction conflicts</u> caused by *common prefixes*
- *Idea:* <u>Left factor</u> the grammar when two productions for the same nonterminal start with the same symbols.

Left Factoring

- "Take the allowable 'endings' and make a new nonterminal to stand for them.”
- "Re-write the productions to defer the decision about which production to use in predictive parsing until enough of the input has been seem that we can make the right choice."

Example

Grammar:

```
E → T | T + E
T → int | int * T | (E)
```

Apply left factoring.

Another Example

Grammar:

```
S → if E then S else S
S → if E then S
```

Apply left factoring.

Note: Can't simply just increase the lookahead. Rewriting the grammar is necessary.

LL(1) Parsing Tables

- From a *left-factored grammar*, going to create a parsing table.
- Parser will use this table to create parse tree, by performing one left-to-right pass on the input, with a leftmost derivation, using 1 symbol of lookahead.
- Components:
  - left column: current leftmost nonterminal in parse tree
  - top row: next input token
  - each table cell entry: information about which production to use to expand the current nonterminal

1 $\mathsf{S} \rightarrow \mathsf{A}\ \mathsf{C}\ \$$
2 $\mathsf{C} \rightarrow \mathsf{c}$
3 $\quad\ \mid\ \lambda$
4 $\mathsf{A} \rightarrow \mathsf{a}\ \mathsf{B}\ \mathsf{C}\ \mathsf{d}$
5 $\quad\ \mid\ \mathsf{B}\ \mathsf{Q}$
6 $\mathsf{B} \rightarrow \mathsf{b}\ \mathsf{B}$
7 $\quad\ \mid\ \lambda$
8 $\mathsf{Q} \rightarrow \mathsf{q}$
9 $\quad\ \mid\ \lambda$

	Lookahead					
Nonterminal	a	b	c	d	q	$
S	1	1	1		1	1
C			2	3		3
A	4	5	5		5	5
B		6	7	7	7	7
Q			9		8	9

Figure 5.10: LL(1) table. The blank entries should trigger error actions in the parser.

Grammar is augmented with the `S → E$` production.

$$
\begin{array}{lll}
S \rightarrow E\,\$ & & \\
 & T \rightarrow F\,T' & \\
E \rightarrow T\,E' & & F \rightarrow \mathrm{id} \\
 & T' \rightarrow *\,F\,T' & F \rightarrow \mathrm{num} \\
E' \rightarrow +\,T\,E' & T' \rightarrow /\,F\,T' & F \rightarrow (\,E\,) \\
E' \rightarrow -\,T\,E' & T' \rightarrow & \\
E' \rightarrow & &
\end{array}
$$

GRAMMAR 3.15.

	+	*	id	(	)	$
S			$S \rightarrow E\$$	$S \rightarrow E\$$		
E			$E \rightarrow TE'$	$E \rightarrow TE'$		
E'	$E' \rightarrow +TE'$				$E' \rightarrow$	$E' \rightarrow$
T			$T \rightarrow FT'$	$T \rightarrow FT'$		
T'	$T' \rightarrow$	$T' \rightarrow *FT'$			$T' \rightarrow$	$T' \rightarrow$
F			$F \rightarrow \mathrm{id}$	$F \rightarrow (E)$		

TABLE 3.17. Predictive parsing table for Grammar 3.15. We omit the columns for `num`, `/`, and `-`, as they are similar to others in the table.

Alg. for processing input stream using parsing table

- Maintain frontier (fringe) of parse tree as a stack.
  - stack will contain non-terminals that have yet to be expanded and terminals yet to be matched against the input
  - top of stack: leftmost terminal or nonterminal
- Initialize stack to start symbol.
- Iterate until:
  - reject on error state
  - accept on end of input and empty stack

Example

Input String:

`id+id$`

$$
\begin{array}{lll}
S \to E\,\$ & & \\
 & T \to F\,T' & \\
E \to T\,E' & & F \to \text{id} \\
 & T' \to *\,F\,T' & F \to \text{num} \\
E' \to +\,T\,E' & T' \to /\,F\,T' & F \to (\,E\,) \\
E' \to -\,T\,E' & T' \to & \\
E' \to & &
\end{array}
$$

GRAMMAR 3.15.

	+	*	id	(	)	$
S			$S \to E\$$	$S \to E\$$		
E			$E \to TE'$	$E \to TE'$		
E'	$E' \to +TE'$				$E' \to$	$E' \to$
T			$T \to FT'$	$T \to FT'$		
T'	$T' \to$	$T' \to *FT'$			$T' \to$	$T' \to$
F			$F \to \text{id}$	$F \to (E)$		

TABLE 3.17. Predictive parsing table for Grammar 3.15. We omit the columns for `num`, `/`, and `-`, as they are similar to others in the table.

- Grammars parsable with LL(1) parsing tables are called LL(1) grammars.
- Grammars parsable with LL(2) parsing tables are called LL(2) grammars, and so on...
- for LL(k), columns are every sequence of k terminals.
  - tables grow very large
  - LL(1) only table used in practice

- Duplicate entries in a parsing table ⇔ grammar is ambiguous.
- No ambiguous grammar is LL(K) for any k.

- How to construct a parsing table for a LL(1) grammar.

After the Break

BEFORE THE BREAK

- finished with example that used a LL(1) parsing table to perform derivation on input token stream

NOW

- How can we construct LL(1) parsing table?
- *Intuition:* during the parsing process, we have:
  1. the leftmost nonterminal on the fringe that we're ready to expand
  2. the next token

Parsing Table Construction

Motivation: FIRST and FOLLOW Sets

- Example table entry in LL(1) Parsing Table: `T[X, t] = Y`
- Under what conditions should the parser use the production rule: $X \rightarrow Y$

Scenario #1

- Y can derive t in the first position.

$$Y \rightarrow^* tZ$$

- Using this move would be a good idea ... eventually the parser could match the t.

$$t \in \text{FIRST}(Y)$$

- The terminal/symbol t is one of the things that Y can produce in the very first position.
- There may be others that Y could also produce.

Scenario #2

- Y reduces to ε, t follows Y in a derivation.

$$Y \not\rightarrow^* t$$

$$t \notin \text{FIRST}(Y)$$

$$Y \rightarrow^* \varepsilon$$

- This production rule is still useful if parser can get rid of X by deriving Y (and by deriving ε).

$$t \in \text{FOLLOW}(X)$$

- Not talking about X deriving t, just that t appears in a derivation *after* X.

$$S \rightarrow^* W\ X\ t\ Z$$

Formal Definition: FIRST

`FIRST(X)` - the set of all terminals, potentially including ε, that can begin strings derived from X

$$FIRST(X) = \{t | X \rightarrow^* tY\} \cup \{\epsilon | X \rightarrow^* \epsilon\}$$

Alg. for computing FIRST sets

1. FIRST(t) = {t}

- first set of a terminal includes itself
- for each terminal symbol t

2. $\varepsilon \in$ FIRST(X)

- if $X \rightarrow \varepsilon$
- if $X \rightarrow Y_1, \ldots Y_k$ and $\varepsilon \in$ FIRST(Y_i) for $1 \le i \le k$
  - // $Y_1, \ldots Y_k$ are all “nullable” and each can derive the empty string

3. FIRST(Z) $\subseteq$ FIRST(X)

- if $X \rightarrow Y_1 \ldots Y_k$ Z and $\varepsilon \in$ FIRST(Y_i) for $1 \le i \le k$
  - $Y_1 \ldots Y_k$ can all derive empty string, therefore $X \rightarrow^* Z$

Example

Compute FIRST sets for the left-factored grammar from before the break:

Grammar:

```
E → TX
X → + E | ε
T → int Y | ( E )
Y → * T | ε
```

Formal Definition: FOLLOW

- FOLLOW(X) - set of *terminals* that can immediately follow X

$$FOLLOW(X) = \{t | S \rightarrow^* WXtZ\}$$

- t ∈ FOLLOW(X) if there is any derivation containing Xt
  - tricky: this can also occur if the derivation contains XYZt where both Y and Z derive ε

FOLLOW Intuition #1

Grammar:

```
X → YZ
Z → ab
```

<u>*Intuition #1:*</u>

- if X → YZ, then FIRST(Z) - {ε} ⊆ FOLLOW(Y)
  - *(ε never appears in FOLLOW sets)*
  - if two symbols are adjacent somewhere in the grammar, the *"FIRST of the second symbol is in the FOLLOW of the first symbol"*

```
X → Yab
```

a ∈ FIRST(Z)
a ∈ FOLLOW(Y)

FOLLOW Intuition #2

- *Intuition #2:*
- if X → YZ, then FOLLOW(X) ⊆ FOLLOW(Z)
  - anything that occurs at the end of a production, its FOLLOW set will include the FOLLOW of the symbol on the LHS.

$\boxed{S \rightarrow \texttt{Xa} \rightarrow \texttt{YZa}}$

a ∈ FOLLOW(X)
a ∈ FOLLOW(Z)

FOLLOW Intuition #3

$S \rightarrow Xa \rightarrow YZa$

- ***Intuition #3:***
- *Generalizing:* What is the end of the production?
- What if $Z \rightarrow^{*} \varepsilon$?
- if $Z \rightarrow^{*} \varepsilon$, then FOLLOW(X) $\subseteq$ FOLLOW(Y)
  - Need to recursively check rightmost symbols for potential ε reduction

$\boxed{S \rightarrow Xa \rightarrow YZa \rightarrow Ya}$

a $\in$ FOLLOW(X)
a $\in$ FOLLOW(Y)

Alg. for computing FOLLOW sets

Rules from before:

1. $\$ \in FOLLOW(S_1)$

- After we reduce start symbol, expect to see EOF.

2. $FIRST(Y) - \{\varepsilon\} \subseteq FOLLOW(X)$

- Not interested in ε: FOLLOW sets are sets of terminals.
- For each production: $\underline{S \rightarrow aXY}$
  - Optional a terminal(s)
  - Y = terminal or nonterminal

3. $FOLLOW(S) \subseteq FOLLOW(X)$

- For each production: $\underline{S \rightarrow aXY}$, where $\varepsilon \in FIRST(Y)$
  - Y "falls out" of the derivation

4. $FOLLOW(S) \subseteq FOLLOW(Y)$
  - If Y does not "fall out"

Example

Compute FOLLOW sets for the left-factored grammar from before the break:

Grammar:

```
E → TX
X → + E | ε
T → int Y | ( E )
Y → * T | ε
```

… now going to <u>construct</u> a parsing table for our left-factored LL(1) grammar!

Grammar:

```
E → TX
X → + E | ε
T → int Y | ( E )
Y → * T | ε
```

Parsing Table Construction Alg.

For each production in G: X → Y

- For each terminal in t ∈ FIRST(Y), add `T[X, t] = Y`
- If ε ∈ FIRST(Y), for each t ∈ FOLLOW(X), add `T[X, t] = Y`
- If ε ∈ FIRST(Y) and $ ∈ FOLLOW(X), add `T[X, $] = Y`

Example

Grammar:

```
E → TX
X → + E | ε
T → int Y | ( E )
Y → * T | ε
```

Another Example

Grammar:

`S → Sa|b`

- Demonstrating that left-recursive grammars are not LL(1)!
- Recall that the language for this grammar is strings of a `b` followed by zero or more `a`'s
- Try building a parse table.

Conclusion

- One way to check if grammar is LL(1) is to build parsing table and see if there are no duplicate entries.

- Classes of grammars that are *not* LL(1):
  - Not left factored
  - Left recursive
  - Ambiguous
  - Plus some others (i.e. those that require more than 1 token of lookahead)

Next Class

Bottom-Up Parsing