SWEN430 - Compiler Engineering

Lecture 4 - Parsing II

David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

Language Theory

Language Theory

- Context-Free Grammar (CFG) is a set of rules of the form

$$N \longrightarrow \alpha_1 \mid \ldots \mid \alpha_2$$

Here, N is a *non-terminal*, and $\alpha_1, \ldots, \alpha_2$ are strings of *terminals* and non-terminals.

$$\begin{aligned} E &\longrightarrow N \mid V \mid \texttt{(}\, E \,\texttt{+}\, E \,\texttt{)} \\ N &\longrightarrow \texttt{[0-9]}^{+} \\ V &\longrightarrow \texttt{[a-zA-Z\_]}^{+}\texttt{[a-zA-Z0-9\_]}^{*} \end{aligned}$$

- Valid words in this language include:

```
1, (2 + 3), ((_x + 3) + 103282)
```

- Invalid words in this language include:

```
0x, 2 + x, ((_x + 3))
```

Predictive Parsing

- **Predictive parsers** choose production based on *next input symbol(s)*. They are nice and easy to implement

$$S \longrightarrow \texttt{a}\ S \mid \texttt{b}\ T \qquad (1,2)$$
$$T \longrightarrow \texttt{c} \mid \texttt{b}\ T \qquad (3,4)$$

- One symbol of lookahead required above, so it's an LL(1) grammar

$$S \longrightarrow \texttt{a}\ \texttt{a}\ S \mid \texttt{a}\ \texttt{b}\ T \qquad (1,2)$$
$$T \longrightarrow \texttt{c} \mid \texttt{b}\ T \qquad (3,4)$$

- Above requires two symbols of lookahead, so it's LL(2)

LL(1) — first() sets

first(γ) set

Let γ be a sequence of terminal and non-terminal symbols. Then *first*(γ) is the set of all terminal symbols which begin a string derived from γ.

$$S \longrightarrow T\ \texttt{a} \mid U\ \texttt{b} \qquad (1,2)$$
$$T \longrightarrow \texttt{d}\ T \mid \texttt{e} \qquad (3,4)$$
$$U \longrightarrow \texttt{c}\ U \mid \texttt{f} \qquad (5,6)$$

first(T) = { d, e }

first(U) = { c, f }

- What does this tell us about production rules 1 and 2 ?

LL(1) — Condition 1

Choice Condition

Consider an $LL(1)$ Context-Free Grammar. For all productions $N \longrightarrow \alpha$ and $N \longrightarrow \beta$, it must hold that $\mathit{first}(\alpha) \cap \mathit{first}(\beta) = \emptyset$.

- For example:

(**i**) $S \longrightarrow \texttt{a}\ S \mid \texttt{a}$

(**ii**)
$$S \longrightarrow T\ \texttt{a} \mid U\ \texttt{b}$$
$$T \longrightarrow \texttt{c}\ T \mid \texttt{d}$$
$$U \longrightarrow \texttt{c}\ U \mid \texttt{e}$$

(**iii**)
$$S \longrightarrow T\ \texttt{a} \mid U\ \texttt{c}$$
$$T \longrightarrow \texttt{d}\ T \mid \texttt{e}$$
$$U \longrightarrow \texttt{c}\ U \mid \epsilon$$

- What are first() sets here? (note: ϵ is *empty token*)

LL(1) — follow() sets

follow(γ) set

Let γ be a non-terminal symbol. Then *follow*(γ) is the set of all terminal symbols which may follow a string derived from γ.

- For example:

$$S \longrightarrow T\ \boxed{\texttt{a}} \mid U\ \boxed{\texttt{c}} \qquad (1,2)$$
$$T \longrightarrow \boxed{\texttt{d}}\ T \mid \boxed{\texttt{e}} \qquad (3,4)$$
$$U \longrightarrow \boxed{\texttt{c}}\ U \mid \epsilon \qquad (5,6)$$

follow(U) = { $\boxed{\texttt{c}}$ }

follow(T) = { $\boxed{\texttt{a}}$ }

- What does this tell us about production rules 5 and 6 ?
- Do we need *follow* sets if there are no ϵ productions ?

LL(1) — Condition 2

ϵ Condition

Consider an $LL(1)$ Context-Free Grammar. For all non-terminal symbols N where $N \longrightarrow \epsilon$, it must hold that $first(N) \cap follow(N) = \emptyset$.

- For example:

$$(\mathbf{i}) \qquad \begin{aligned} S &\longrightarrow \texttt{a} \mid T\ \texttt{b} \\ T &\longrightarrow \texttt{b} \mid \epsilon \end{aligned}$$

$$(\mathbf{ii}) \qquad \begin{aligned} S &\longrightarrow \texttt{(}\ T \mid \epsilon \\ T &\longrightarrow S\ \texttt{)}\ S \end{aligned}$$

- What are the follow() sets for these grammars?

Left factoring

Left factoring

Suppose we have productions $N \longrightarrow x\ \alpha$ and $N \longrightarrow x\ \beta$, where x is either a terminal or non-terminal. Then, we can rewrite this into $N \longrightarrow x\ M$ and $M \longrightarrow \alpha \mid \beta$.

- Example:

$$\begin{aligned} \mathit{List} &\longrightarrow \texttt{(}\ \texttt{)} \mid \texttt{(}\ \mathit{ListBody}\ \texttt{)} \\ \mathit{ListBody} &\longrightarrow \mathit{ListElt} \mid \mathit{ListElt}\ \texttt{,}\ \mathit{ListBody} \\ \mathit{ListElt} &\longrightarrow N \mid \mathit{List} \end{aligned}$$

Becomes [N=List, x=(, α=), β=ListBody)] :

$$\begin{aligned} \mathit{List} &\longrightarrow \texttt{(}\ \mathit{RestOfList} \\ \mathit{RestOfList} &\longrightarrow \texttt{)} \mid \mathit{ListBody}\ \texttt{)} \end{aligned}$$

Eliminating Left Recursion

- Consider this grammar:

$$E \longrightarrow E \boxed{+} T \mid T$$
$$T \longrightarrow [\boxed{0} - \boxed{9}]^{+}$$

This grammar is **left-recursive**. Why is it not $LL(1)$?

- Can (sometimes) eliminate left-recursion by using **right-recursion** instead. Consider this alternative to the above:

$$E \longrightarrow T\ E'$$
$$E' \longrightarrow \boxed{+}\ T\ E' \mid \epsilon$$
$$T \longrightarrow [\boxed{0} - \boxed{9}]^{+}$$

Is this equivalent to the left-recursive grammar?

Parser Generators

Parser Generators

Yacc

Yet Another Compiler-Compiler generates *bottom up* parses for LALR parser grammars. It was originally developed for UNIX by Stephen C. Johnson.

JavaCC

The **Java Compiler Compiler** generates top-down recursive descent parsers for $LL(k)$ grammars.

ANTLR

Another Tool for Language Recognition generates bottom up parses for $LL(*)$ grammars. It was originally developed by Terrance Parr (see `antlr.oorg`).

Example of Java Grammar in ANTLR

```
grammar Expressions;

expr: expr op=('*' | '/') expr
    | expr op=('+' | '-') expr
    | INT
    | IDENTIFIER
    ;

INT: [0-9]+;
IDENTIFIER: [a-zA-Z][a-zA-Z0-9]+;
WS: [\t\r\n]+ -> skip;
```

- ANTLR grammar for simple expressions