SWEN430 - Compiler Engineering (2018)

Lecture 5 - Parsing III

Lindsay Groves & David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

Recap: Recursive Descent Parsing

- For each nonterminal *N* in the grammar, define a method *parseN* to recognise an instance of *N* as a prefix of the input, and build an AST for it.
- Logic of the parse methods reflects the structure of the grammar. Can be coded directly from the grammar once it is in LL(1) form.
- Sequence in grammar –> sequence in parser.
- Alternatives in grammar –> if/switch in parser
- Recursion in grammar –> recursion in parser

Recap: When does it work? LL(1) conditions

- Must be able to decide what to do on basis of next input symbol.

- **Choice Condition**:

  Given $N \longrightarrow \alpha_1 \mid \ldots \mid \alpha_n$,

  no symbol that can start both α_i and α_j, for $i \neq j$.

  i.e. $\mathit{first}(\alpha_i) \cap \mathit{first}(\alpha_j) = \emptyset$ for $i \neq j$.

- If a grammar doesn't satisfy this condition, we can sometimes transform it into one that does by *left factoring*.

Eliminating Left Recursion

- Left recursion occurs frequently in PL grammars, e.g.:

  $E \longrightarrow E + n \mid n$

  Defines sequences of numbers (n) separated by plus signs.

- We can't have left recursion in an LL(1) grammar!

  It breaks the Choice Condition: $\mathit{first}(E + n) = \{n\} = \mathit{first}(n)$.

  And would put a parser into an infinite loop!

- Can we remove this by left-factoring?

  First, we need to expand the E on the right-hand side:

  $$\begin{array}{ll} & E + n \mid n \\ = & (E + n \mid n) + n \mid n \\ = & E + n + n \mid n + n \mid n \end{array}$$

  We can never get rid of the E!

Eliminating Left Recursion

- We can avoid left-recursion by using *right-recursion* instead:

  $$E \longrightarrow T\ E'$$
  $$E' \longrightarrow +\ T\ E'\ \mid\ \lambda$$
  $$T \longrightarrow n$$

  This changes the parse tree. Does that matter?

- Or, we can use an *extended BNF grammar*, allowing repetition as in regular expressions:

  $$E \longrightarrow T\ (\ +\ T\ )^*$$
  $$T \longrightarrow n$$

  More on this soon.

When does it work? — Take two

- When else does the parser need to make a decision?
- Consider a grammar of the form:

$$\begin{aligned} S &\longrightarrow a\ T\ U \\ T &\longrightarrow \lambda \mid b \\ U &\longrightarrow b \end{aligned}$$

  Since T can produce the empty string (λ), we need to decide whether a b in the input is part of T or part of U (in which case T is λ).

- If the grammar is:

$$\begin{aligned} S &\longrightarrow a\ T\ U \\ T &\longrightarrow \lambda \mid b \\ U &\longrightarrow c \end{aligned}$$

  There is no problem.

LL(1) — Condition 2

- λ **Condition:**

  If a non-terminal symbol *N* can produce an empty string (formally, $N \Longrightarrow^* \lambda$), then

  No symbol than can start an occurrence of *N* can also follow an occurrence of *N*.

  i.e. $\mathit{first}(N) \cap \mathit{follow}(N) = \emptyset$.

- $\mathit{follow}(N)$ is the set of terminals that can follow an occurrence on *N* in a sentence.

LL(1) — Condition 2

- Examples:

(i)
$$S \longrightarrow \mathtt{a} \mid T\ \mathtt{b}$$
$$T \longrightarrow \mathtt{b} \mid \epsilon$$

(ii)
$$S \longrightarrow \mathtt{(}\ T \mid \epsilon$$
$$T \longrightarrow S\ \mathtt{)}\ S$$

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

Follow sets

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 in a sentence.

- For example:

$$\begin{array}{lll} S \longrightarrow T\ \mathtt{a} \mid U\ \mathtt{c} & & (1,2) \\ T \longrightarrow \mathtt{d}\ T \mid \mathtt{e} & & (3,4) \\ U \longrightarrow \mathtt{c}\ U \mid \lambda & & (5,6) \end{array}$$

follow(U) = { $\mathtt{c}$ }
follow(T) = { $\mathtt{a}$ }

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

Recursive Descent Parsing with Extended BNF Grammars

- Extended BNF grammars allow:
  - Nested alternatives, using (...) for grouping.
  - [A] to mean that A is optional.
  - A^* to mean that A may occur 0 or more times.
  - A^+ to mean that A may occur 1 or more times.
- How can we extend the recursive descent to handle these forms?
- How do we extend the LL(1) conditions?

Recursive Descent Parsing with Extended BNF Grammars

- To parse [A]:

```
if nextSym can start A then parseA;
```

- To parse A^*:

```
while nextSym can start A do parseA;
```

- To parse A^+:
  - Treat as $A\,A^*$

    ```
    parseA; while nextSym can start A do parseA;
    ```

  - Or:

    ```
    do parseA while nextSym can start A
    ```

- [A] and A^* can produce λ, so need to apply λ Condition to them.

Some notes on the While parser

- The grammar given in the While Language Specification is not LL(1).

  ... and is incomplete, and sometimes inconsistent with the compiler.

  E.g. print statements, block statements and union types are missing.

- Some statements can only occur in certain contexts.
  e.g. `break` and `continue` statements can only occur inside a loop.

- And statements like assignments don't always need semicolons.

- We could build that into the grammar, but it's easier (?) to keep a flag saying whether we're inside a loop or need semicolons.

Some notes on the While parser

- Assignment statements, variable declarations and method calls all start with an identifier, so we can't tell from that symbol what kind of statement it is.
- Can handle this by either:
  - Looking in the symbol table to see whether the identifier was declared as a variable name, type name or method name (i.e. using semantic information).

    When will/won't this work?
  - Looking ahead further:
    - If the next symbol is a "(" it's a method call;
    - If it's "=" or "[" it's an assignment.
    - What if it's a "."?
    - What else could it be?

Some notes on the While parser

- Expressions are the most complicated part of the grammar.
- The way we write the grammar rules determines the *precedence* and *associativity* of operators.
- We might define arithmetic expressions as:

  $$E \quad ::= \quad E + E \mid E - E \mid E * E \mid E / E \mid \mathit{number} \mid \mathit{variable} \mid (E)$$

  But this is ambiguous.

  We can construct different parse trees for expressions such as $1 + 2 + 3$.

Some notes on the While parser

- It is common the write the grammar for arithmetic expressions as:

$$\begin{array}{lcl} E & ::= & E + T \mid E - T \mid T \\ T & ::= & T * F \mid T / F \mid F \\ F & ::= & \textit{number} \mid \textit{variable} \mid (E) \end{array}$$

- This means that:
  - $+$, $-$, $*$ and $/$ are all *left-associative*

    e.g. $1 - 2 - 3$ is treated as $(1 - 2) - 3$, and
  - $*$ and $/$ have *higher precedence* than $+$ and 0.

    e.g. $1 + 2 * 3$ is treated as $1 + (2 * 3)$.
- But ... it still has left-recursion.
- and changing to right-recursion makes all the operators right-associative!!
  (if we base order of evaluation on the structure of the parse tree)

Some notes on the While parser

- Using EBNF notation, we can write:

$$
\begin{array}{lcl}
E & ::= & T\ ((+ \mid -)\ T)^* \\
T & ::= & F\ ((* \mid /)\ F)^* \\
F & ::= & \mathit{number} \mid \mathit{variable} \mid \text{"("}\ E\ \text{")"}
\end{array}
$$

- We can now code the parsing algorithms using loops.

- The While parser distinguishes between arithmetic, relational and logical expressions in the grammar and in the parsing methods.

  An alternative is to use a more flexible grammar for the parser, and handle these distinctions in the type checker.