50006 - Compilers - (Dr Dulay) Lecture 3

Oliver Killane

09/04/22

Lecture Recording

Lecture recording is available here

LL Parsing

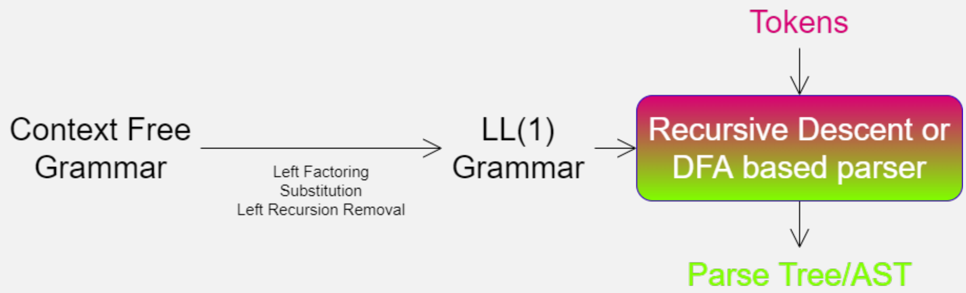
Top-down parsing using either recursive descent (can be hand-coded) or an **LL(1)** pushdown automaton (generated by an **LL(1)** parser-generator (e.g ANTLR)).

Definition: LL(k) Grammar

A grammar is **LL(k)** if a k-token lookahead is sufficient to determine which alternative of a rule to use when parsing.

- **LL(1)** uses the current token only.
- **LL(0)** does not exist (would be deciding based on 0 tokens).
- When using **LL/Top Down Parsing** leaf nodes are constructed from the root.

The parser created for the grammar can be implemented as either a **DFA** or to parse by **recursive descent**.



Definition: LL(1) Grammar

A grammar is **LL(1)** if for a rule $A \to \alpha \mid \beta$ (A is non-terminal).

$$\begin{aligned} &first(\alpha) \cap first(\beta) = \emptyset \\ \wedge\, &\epsilon \in first(\alpha) \Rightarrow (first(\beta) \cap follow(A) = \emptyset) \\ \wedge\, &\epsilon \in first(\beta) \Rightarrow (first(\alpha) \cap follow(A) = \emptyset) \end{aligned}$$

$$A \to \underbrace{\alpha}_{t \in first(\alpha)} \mid \overbrace{\beta}^{t \in first(\beta)}$$

If both could start with t, if we get t as the current token, we cannot determine if we are parsing a or b.

$$A \to \underbrace{\alpha}_{\epsilon \in first(\alpha)} \mid \overbrace{\beta}^{t \in first(\beta)} \qquad follow(A) = \{\ldots, t, \ldots\}$$

If a contains e, then we could potentially skip over this rule to whatever follows A.

Here with token t we cannot determine if we are skipping past this rule, or parsing b.

And the other order, and with many alternatives

This extends to more than two alternatives.

Definition: Extended Backus-Naur Form (BNF)

Used for writing context free grammars, and includes several useful features:

$\{\alpha\}$	0 or more occurrences of α
$[\alpha]$	0 or 1 occurrences of α
$(\ldots)$	A way to group elements together

For example we can left factor rules with alternatives that have intersecting first sets.

$$\begin{gathered} Expr \to Term \text{ '+' } Expr \mid Term \\ \text{then becomes} \\ Expr \to Term \text{ ['+' } Expr] \end{gathered}$$

Or we can remove left recursion:

$$\begin{gathered} Sequence \to Sequence \text{ ';' } Statement \mid Statement \\ \text{then becomes} \\ Sequence \to Statement \text{ \{';' } Statement\} \end{gathered}$$

Definition: Recursive Descent Parser

Consists of a set of parse functions for each rule which take in some tokens, and return remaining tokens and a generated **AST**.

We can use a basic function for matching **terminal** tokens:

```
# Get the next token. If it matches the expected, pop it from
# the list of tokens, else throw an error.
def match(expected: token):
    if lexical_analyser.next_token() == expected:
        lexical_analyser.pop_token()
    else:
        raise error("Unexpected token")
```

We can apply some basic rules for the main patterns:

```
# A B
A()
B()

# A | B (to be an LL(1) grammar, first sets must be disjoint)
if next_token() in first(A):
    A()
elif next_token() in first(B):
    B()

# {A}
while next_token() in first(A):
    A()

#[A]
if next_token() in first(A):
    A()
```

Example: Statements

$$
\begin{aligned}
\mathit{Stat} &\rightarrow \mathit{IfStat} \mid \mathit{BeginStat} \mid \mathit{PrintStat} \\
\mathit{IfStat} &\rightarrow \text{'if'}\ \mathit{Expr}\ \text{'then'}\ \mathit{Stat}\ [\text{'else'}\ \mathit{Stat}]\ \text{'fi'} \\
\mathit{BeginSTat} &\rightarrow \text{'begin'}\ \mathit{Stat}\ \{\text{';'}\ \mathit{Stat}\}\ \text{'end'} \\
\mathit{PrintStat} &\rightarrow \text{'print'}\ \mathit{Expr}
\end{aligned}
$$

```python
def Stat() -> Statement:
    if next_token == IF:
        return IfStat()
    elif next_token == BEGIN:
        return BeginStat()
    elif next_token == PRINT:
        return PrintStat()
    else:
        raise error("Expected Statement Starting Token")

# Parse an if statement, returning the statement if parse succeeds.
def IfStat() -> Statement:
    match(IF)
    cond = Expr()
    match(THEN)
    if_branch = Stat()
    else_branch = None
    if next_token() == ELSE:
        match(ELSE)
        else_branch = Stat()
    match(FI)
    return IfStatement(cond, if_branch, else_branch)

# Parse a block of statements, returning the block statement if successful
def BeginStat() -> Statement:
    stats = []
    match(BEGIN)
    stats.append(Stat())
    while next_token() == SEMICOLON:
        match(SEMICOLON)
        stats.append(Stat())
    match(END)
    return Block(stats)

# Parse a print statement, returning the statement if successful.
def PrintStat() -> Statement:
    match(PRINT)
    return PrintStatement(Expr())
```

AST Construction

Class hierarchies can be used to create organise nodes into variants of a given type (e.g if statements, print statements and assignments are all statements, so implement/inherit from some STatement class/interface).

Other languages

While in languages such as Java or Python, class hierarchies are used to implement ASTs other languages have cleaner representations.

- In **Haskell** the data keyword can be used to build an **ADT** for an **AST**
- Rust supports enums similar to the haskell data keyword.

CFG $\to$ LL(1) Conversion

Transformations must be applied to a non-LL(1) context free grammar, which cannot always be automated.

- Left Recursion Removal and Substitution are usually applied first.
- The semantics of the grammar must be maintained.
- Readability is a key consideration, expecially if the language is to be extended in future.

Definition: Left Factorisation

Where two or more alternatives of a rule have a common prefix (first element to be parsed), factor this to be parsed before determining which alternative to parse.

non-LL(1)	**EBNF LL(1)**	**BNF LL(1)**
$A \to B\ C\ \vert B\ D$	$A \to B\ (C \mid D)$	$A \to B\ X$ $X \to C \mid D$
$A \to B\ C\ \vert B$	$A \to B\ [C]$	$A \to B\ X$ $X \to C \mid \epsilon$

Definition: Substitution

Substituting a rule/non-terminal with its alternatives.

- Can be used to find indirect conflicts that may require left-factoring.
- Can produce grammars ecoding more information in a rule (makes it easier to produce the required AST)

$$
\begin{aligned}
A &\to B \mid C \\
B &\to \text{'hello'} \\
C &\to \text{'hello' 'there'}
\end{aligned}
$$

Here we have an indirect conflict as both alternatives for C start with 'hello'.

$$A \to \text{'hello'} \mid \text{'hello' 'there'}$$

Now we can left-factor.

$$A \to \text{'hello' ['there']}$$

Definition: Left Recursion Removal

Grammars cannot be **LL(1)** with left-recursion. We can use **direct left recursion removal** to eliminate left recursion from rules while leaving the grammar mostly intact.

$$A \to X \mid A\ Y \Rightarrow A \to X\{Y\}$$

For example with left-associative arithmetic expressions:

$$\begin{aligned} Expr &\to Expr\ (\text{'+'} \mid \text{'-'})\ Term \mid Term \\ Term &\to Term\ (\text{'*'} \mid \text{'/'}) \mid Factor \\ Factor &\to \text{'('}\ Expr\ \text{')'} \mid \underline{int} \end{aligned}$$

The **parse tree** may no longer represent the associativity, hence we will need to ensure the arithmetic is still associative when we construct the **AST**:

$$\begin{aligned} Expr &\to Term\ \{(\text{'+'} \mid \text{'-'})\ Term\} \\ Term &\to Factor\ \{(\text{'*'} \mid \text{'/'})\ Factor\} \\ Factor &\to \text{'('}\ Expr\ \text{')'} \mid \underline{int} \end{aligned}$$

Error Recovery

When detecting an error in parsing. . .

- Useful error messages need to contain information collected during parsing.
- Error recovery can be used to allow further parse errors to be detected, with as little code as possible being skipped but avoiding nonsense/spurious error messages being created as a result.
- Error correction can be attempted to allow for semantic checks to be performed. The corrections must attempt to emulate what the semantics of the erroneous code likely was.

Definition: Panic Mode Recovery

Each parse function has a **syncset** of tokens. When an error occur, the parser skips forwards until it encounters one of these tokens.

- Additional tokens can be provided as arguments to the parsing function (e.g add **follow set** of outer non-terminal to inner parse)
- The **follow set** of a rule is often used as the **syncset**
-

```python
# match a token, if necessary report and error, return boolean if the token matched expected.
def match(expected) -> bool:
    if next_token() == expected:
        pop_token()
        return True
    else:
        add_error(next_token(), parser_pos(), [expected], INCORRECT_TOKEN)
        return False

# Skip until at the end of the file, or token matches the sync set
def skipto(syncset):
    while next_token() not in syncset and next_token() != EOF:
        pop_token()


def check(expectset, syncset, error):
    if next_token() not in expectset:
        add_error(next_token(), parser_pos(), expectset, error)
        skipto(expectset + syncset)
```