Today

- Syntax Analysis: Recursive Descent Parsing

Assignments

- HW3 out (due Tues)
Parsing: An example grammar

Simple list of assignment statements

\[ stmt \_list \rightarrow stmt \mid stmt \; ; \; stmt \_list \]
\[ stmt \rightarrow var \; = \; expr \]
\[ var \rightarrow \texttt{`A'} \mid \texttt{`B'} \mid \texttt{`C'} \]
\[ expr \rightarrow var \mid var \; + \; var \mid var \; - \; var \]

– quotes used here to help distinguish terminals from non-terminals
– Note: many possible grammars for our language!

Recall from last time:

• This grammar is not left-recursive (which is a good thing!)
• But, it is \( LL(6) \) (e.g., because of \( \texttt{A} = \texttt{B} + \texttt{C}; \texttt{B} = \texttt{A} \))
• We can reduce \( k \) by left factoring

Give it a try ...
A more standard way to write the grammar (with problems fixed)

\[
\begin{align*}
\text{stmt}\_\text{list} & \rightarrow \text{var} \ 'N' \ \text{expr} \ \text{stmt}\_\text{list} \_\text{tail} \\
\text{stmt}\_\text{list}\_\text{tail} & \rightarrow \ 'L' \ \text{stmt}\_\text{list} \mid \epsilon \\
\text{var} & \rightarrow \ 'A' \mid 'B' \mid 'C' \\
\text{expr} & \rightarrow \ \text{var} \ \text{expr}\_\text{tail} \\
\text{expr}\_\text{tail} & \rightarrow \ '+' \ \text{var} \mid '-' \ \text{var} \mid \epsilon
\end{align*}
\]

Note that for parsing, it is convenient to rewrite using token types ...

\[
\begin{align*}
\langle \text{stmt}\_\text{list}\rangle & \ ::= \ \text{VAR} \ \text{ASSIGN} \ \langle \text{expr}\rangle \ \langle \text{stmt}\_\text{list}\_\text{tail}\rangle \\
\langle \text{stmt}\_\text{list}\_\text{tail}\rangle & \ ::= \ \text{SEMICOLON} \ \langle \text{stmt}\_\text{list}\rangle \mid \epsilon \\
\langle \text{expr}\rangle & \ ::= \ \text{VAR} \ \langle \text{expr}\_\text{tail}\rangle \\
\langle \text{expr}\_\text{tail}\rangle & \ ::= \ \text{PLUS} \ \text{VAR} \mid \text{MINUS} \ \text{VAR} \mid \epsilon
\end{align*}
\]

Where ::= is used in place of \( \rightarrow \) (read as “becomes”)
Recursive Descent Parsing

A simple approach for *ad hoc* parsing

- use **recursion** to parse the token stream
- divide parse into separate methods ... roughly one for each *non terminal*
  - corresponding grammar rule(s) encoded in the non-terminals method
- “descend” the parse tree using method calls (possibly recursion)

We’ll use a **Parser** class with basic methods and member variables:

```python
class Parser(object):
    def __init__(self, lexer):
        self.lexer = lexer  # for next_token function
        self.current_token = None  # used during parsing

    def parse(self): ...  # succeeds if correct syntax

    # helper functions
    def __advance(self): ...  # move forward in token stream
    def __eat(self, tokentype, err_msg): ...  # check curr token's type, advance
    def __error(self, err_msg): ...  # print an error

    # recursive descent functions
    def __stmt_list(self): ...
    def __stmt_list_tail(self): ...
    def __expr(self): ...
    def __expr_tail(self): ...
```
The Helper Functions

# get and store the next token
def __advance(self):
    self.current_token = self.lexer.next_token()

# ensure current token in stream is of given type, and advance
def __eat(self, tokentype, err_msg):
    if self.current_token.tokentype == tokentype:
        self.__advance()
    else:
        self.__error(err_msg)

# handle errors
def __error(self, err_msg):
    s = err_msg + ' found "' + self.current_token.lexeme + '"';
    line = self.current_token.line
    column = self.current_token.column
    raise error.Error(s, line, column)  # MyPLError for HW-3

Note that raising an exception stops parser at first error

• as opposed to printing error, and then continuing
The Recursive Descent Functions

The example:

\[
\langle stmt\_list \rangle ::= \text{VAR}\ A\SSIGN\ \langle expr \rangle\ \langle stmt\_list\_tail \rangle \\
\langle expr \rangle ::= \text{VAR}\ \langle expr\_tail \rangle \\
\langle stmt\_list\_tail \rangle ::= \text{SEMICOLON}\ \langle stmt\_list \rangle|\epsilon
\]

How it works:

```python
def parse(self):
    self.__advance() # set current_token
    self.__stmt_list() # check stmt list rule
    self.__eat(token.EOS, 'expecting end-of-file')

def __stmt_list(self):
    self.__eat(token.VAR, 'expecting variable') # check if var, advance
    self.__eat(token.ASSIGN, 'expecting "="') # check if =, advance
    self.__expr() # check expr rule
    self.__stmt_list_tail() # check tail rule

def __expr(self):
    self.__eat(token.VAR, 'expecting variable')
    self.__expr_tail()
```

Watch out for aligning calls to advance in each recursive descent function

- assume the current token is already advanced (i.e., 1 look ahead)
- or the function starts by advancing the current token

Exercise: Finish the recursive descent functions for this language
How to handle rule decisions (which version of a rule to take)

- assuming grammar is in $LL(1)$, only need to look at the current token
- as an example, consider this MyPL rule (for Boolean expressions):

$$\texttt{<bexpr> ::= <expr> <bexpr>}$$
$$\quad \text{ | NOT <bexpr> <bexpr>}$$
$$\quad \text{ | LPAREN <bexpr> RPAREN <bconnct>}$$

Current token is either NOT, LPAREN, or else the first case ...

```python
def __bexpr(self):
    if self.current_token tokentype == token.NOT:
        self.__advance()
        self.__bexpr()
        self.__bexpr()
    elif self.current_token tokentype == token.LPAREN:
        self.__advance()
        self.__bexpr()
        self.__bexpr()
        self.__eat(token.RPAREN, 'expecting ""')
        self.__bconnct()
    else:
        self.__expr()
        self.__bexpr()
```

**Note:** don’t always need one recursive descent function per non-terminal

- this is more of a rule of thumb ...
- sometimes you can condense ... do what makes sense
- when we add the AST, we may also need to reorganize a bit
Some "trickier" cases ...

\[
\langle expr \rangle ::= ( \langle rvalue \rangle \ | \ LPAREN \langle expr \rangle \ RPAREN \ ) \ ( \langle mathrel \rangle \langle expr \rangle \ | \ \epsilon \ )
\]

Note that this rule is really four separate rules:

- The general structure of the rule is:

  \[ e \to ( v_1 \ | \ v_2 ) ( r_1 \ | \ r_2 ) \]

- Which corresponds to:

  \[
  e \to v_1 \ r_1 \\
  e \to v_1 \ r_2 \\
  e \to v_2 \ r_1 \\
  e \to v_2 \ r_2 
  \]

We can handle the entire thing in one recursive descent function:

```python
def __expr(self):
    if self.current_type.token_type == token.LPAREN:
        self.__advance()
        self.__expr()
        self.__eat(token.RPAREN, 'expecting ")"')
    else:
        self.__rvalue()
        mathrels = [token.PLUS, token.MINUS, ...]
        if self.current_token.token_type in mathrels:
            self.__advance()
            self.__expr()
```

Note that here you have to “dig in” to \( \langle mathrel \rangle \) to see if we should apply the rule
As another “tricky” case ... use (while) loops for Kleene star

- For example, consider this rule:

\[
\langle exprlist \rangle \ ::= \langle expr \rangle ( \text{COMMA} \langle expr \rangle )^* \mid \epsilon
\]

Again, we can do this in a single function call ...

```python
def __exprlist(self):
    # tokens that can start an expression ...
    types = [token.STRINGVAL, token.INTVAL, ... token.ID, token.LPAREN]
    if self.current_token.tokentype in types:
        self.__expr()
        while self.current_token.tokentype == token.COMMA:
            self.__advance()
        self.__expr()
```

Again, here we have to “dig in” to \langle expr \rangle to determine whether to apply it