Travaux dirigés

Introduction à la Compilation

2-Analyse syntaxique

Jean-Christophe Le Lann, Quentin Ducasse

Janvier 2023

Exercice 1 :

Question 1 : Observer les règles de grammaire précédentes et déterminer les lexèmes du langage et leur type.

Solution : Les lexèmes de Mini-C sont les suivants:

- **Mots-clés :** `int`, `bool`, `float`, `char`, `main`, `if`, `else`, `while`
- **Symboles :** `( ) { } [ ] ; || && == != < <= > >= + - * / % !`
- **Littéraux :** Identifiants, chaînes de caractères, nombres entiers et nombres flottants

Question 2 : Ecrire quelques codes en `Mini-C`, qui soient corrects syntaxiquement (on ne demande pas d'écrire des codes qui aient un véritable sens algorithmique. Ces codes serviront en question 4 pour tester votre parseur.

Solution : Voici deux programmes de difficulté croissante.

```
// Test operateurs
int main() {
    int val1;
    int val2;
    bool result;
    bool is_equal;
    is_equal = false;
    result1 = -2 + 3 * 4;
    result2 = 4 % 5 / 6;
    is_equal = !is_equal;
    result = val1 < val2;
}
```

Le deuxième vérifie les conditions `if`, `else` et `while`.

```
// Test conditions
int main() {
    bool test1;
    int i;
    test1 = true;
    if (test1) {
        i = 0;
    } else {
      while(i != 10) {
        i = i + 1;
      }
    }
}
```

Question 3 : Écrire le lexer de `Mini-C`. À partir d'un texte fourni en entrée (ou du nom du fichier contenant ce texte), le lexer doit générer un tableau de lexèmes.

Solution : Dans votre fichier `constants` fourni :

```
LEXEM_REGEXES = [
    # Comments and whitespaces
    (r"\/\/.*", "COMMENT"),
    (r"[ \t\n]+", None),
    # Keywords
    (r"int", "TYPE_INT"),
    (r"bool", "TYPE_BOOL"),
    (r"float", "TYPE_FLOAT"),
    (r"char", "TYPE_CHAR"),
    (r"main", "KW_MAIN"),
    (r"true", "KW_FALSE"),
    (r"false", "KW_TRUE"),
    (r"if", "KW_IF"),
    (r"else", "KW_ELSE"),
    (r"while", "KW_WHILE"),
    # Comparisons
    (r"\=\=", "EQUALITY"),
    (r"\!\=", "INEQUALITY"),
    (r"\<", "LT"),
    (r"\<\=", "LTE"),
    (r"\>", "GT"),
    (r"\>\=", "GTE"),
    (r"\|\|", "OR"),
    (r"\&\&", "AND"),
    # Operators
    (r"\+", "ADDITION"),
    (r"\-", "SUBTRACTION"),
    (r"\*", "MULTIPLICATION"),
    (r"\/", "DIVISION"),
    (r"\%", "MODULO"),
    (r"\!", "NOT"),
    # Special characters
    (r"\=", "ASSIGN"),  # Put equal here to let the double operators before
    (r"\;", "TERMINATOR"),
    (r"\{", "L_CURL_BRACKET"),
    (r"\}", "R_CURL_BRACKET"),
    (r"\[", "L_SQ_BRACKET"),
    (r"\]", "R_SQ_BRACKET"),
    (r"\(", "L_PAREN"),
    (r"\)", "R_PAREN"),
    # Identifiers & Integers
    (r"[a-z]\w*", "IDENTIFIER"),
    (r"-?\d+", "INT_NUMBER"),
    (r"-?\d+(\.\d*)?", "FLOAT_NUMBER"),
]
```

Note : Attention, le caractère de retour à la ligne dépend des architectures et de votre système d'exploitation. Nous utilisons ici pour Linux mais Windows utilise ˚ et des vieux Mac utilisent ˚.

Question 4 : Compléter le code précédent de manière à parser les codes exemples écrits à la première question

---

```
import logging

logger = logging.getLogger(__name__)


class ParsingException(Exception):
    pass


class Parser:
    def __init__(self, lexems):
        """
        Component in charge of syntaxic analysis.
        """
        self.lexems = lexems

    # ============================
    #      Helper Functions
    # ============================

    def accept(self):
        """
        Pops the lexem out of the lexems list.
        """
        self.show_next()
        return self.lexems.pop(0)

    def show_next(self, n=1):
        """
        Returns the next token in the list WITHOUT popping it.
        """
        try:
            return self.lexems[n - 1]
        except IndexError:
            self.error("No more lexems left.")

    def expect(self, tag):
        """
        Pops the next token from the lexems list and tests its type through the
↪   tag.
        """
        next_lexem = self.show_next()
        if next_lexem.tag != tag:
```

```
            raise ParsingException(
                f"ERROR at {str(self.show_next().position)}: Expected {tag}, got
                ↪  {next_lexem.tag} instead"
            )
        return self.accept()

    def remove_comments(self):
        """
        Removes the comments from the token list by testing their tags.
        """
        self.lexems = [lexem for lexem in self.lexems if lexem.tag != "COMMENT"]

    # ==========================
    #     Parsing Functions
    # ==========================

    def parse(self):
        """
        Main function: launches the parsing operation given a lexem list.
        Note that for the other parsing functions, the BNF is shown with the raw
        characters between single quotes.
        """
        try:
            self.remove_comments()
            self.parse_program()
        except ParsingException as err:
            logger.exception(err)
            raise

    def parse_program(self):
        """
        int main '(' ')' '{' declaration* statement* '}'
        """
        self.expect("TYPE_INT")
        self.expect("KW_MAIN")
        self.expect("L_PAREN")
        self.expect("R_PAREN")
        self.expect("L_CURL_BRACKET")
        while self.show_next().tag != "R_CURL_BRACKET":
            if self.show_next().tag in ["TYPE_INT", "TYPE_FLOAT", "TYPE_BOOL",
            ↪  "TYPE_CHAR"]:
                self.parse_declaration()
            else:
                self.parse_statement()
        self.expect("R_CURL_BRACKET")

    def parse_declaration(self):
        """
        ('int'|'float'|'bool'|'char') identifier ('[' identifier ']')?  ';'
        """
        if self.show_next().tag in ["TYPE_INT", "TYPE_FLOAT", "TYPE_BOOL",
        ↪  "TYPE_CHAR"]:
```

```
            self.accept()
            self.expect("IDENTIFIER")
            if self.show_next().tag == "L_SQ_BRACKET":
                self.expect("L_SQ_BRACKET")
                self.expect("INT_NUM")
                self.expect("R_SQ_BRACKET")
            self.expect("TERMINATOR")
        else:
            self.error("No type specified.")

    def parse_statement(self):
        """
        assignment | if_statement | while_statement
        """
        if self.show_next().tag == "IDENTIFIER":
            self.parse_assignment()
        elif self.show_next().tag == "KW_IF":
            self.parse_if_statement()
        elif self.show_next().tag == "KW_WHILE":
            self.parse_while_statement()
        else:
            self.error("Expecting statement (identifier, if or while)")

    def parse_assignment(self):
        """
        identifier ('[' expression ']')? '=' expression ';'
        """
        self.expect("IDENTIFIER")
        if self.show_next().tag == "L_SQ_BRACKET":
            self.expect("L_SQ_BRACKET")
            self.parse_expression()
            self.expect("R_SQ_BRACKET")
        self.expect("ASSIGN")
        self.parse_expression()
        self.expect("TERMINATOR")

    def parse_if_statement(self):
        """
        'if' '(' expression ')' '{' statement* '}' else_statement?
        """
        self.expect("KW_IF")
        self.expect("L_PAREN")
        self.parse_expression()
        self.expect("R_PAREN")
        self.expect("L_CURL_BRACKET")
        while self.show_next().tag != "R_CURL_BRACKET":
            self.parse_statement()
        self.expect("R_CURL_BRACKET")
        if self.show_next().tag == "KW_ELSE":
            self.parse_else_statement()

    def parse_else_statement(self):
```

```
        """
        'else' '{' statement* '}'
        """
        self.expect("KW_ELSE")
        self.expect("L_CURL_BRACKET")
        while self.show_next().tag != "R_CURL_BRACKET":
            self.parse_statement()
        self.expect("R_CURL_BRACKET")

    def parse_while_statement(self):
        """
        'while' '(' expression ')' '{' statement* '}'
        """
        self.expect("KW_WHILE")
        self.expect("L_PAREN")
        self.parse_expression()
        self.expect("R_PAREN")
        self.expect("L_CURL_BRACKET")
        while self.show_next().tag != "R_CURL_BRACKET":
            self.parse_statement()
        self.expect("R_CURL_BRACKET")

    def parse_expression(self):
        """
        conjunction ('||' conjunction)*
        """
        self.parse_conjunction()
        while self.show_next().tag == "OR":
            self.accept()
            self.parse_conjunction()

    def parse_conjunction(self):
        """
        equality  ('&&' equality)*
        """
        self.parse_equality()
        while self.show_next().tag == "AND":
            self.accept()
            self.parse_equality()

    def parse_equality(self):
        """
        relation (('=='|'!=') relation)*
        """
        self.parse_relation()
        while self.show_next().tag in ["EQUALITY", "INEQUALITY"]:
            self.accept()
            self.parse_relation()

    def parse_relation(self):
        """
        addition (('<'|'<='|'>='|'>') addition)*
```

```
        """
        self.parse_addition()
        while self.show_next().tag in ["LT", "LTE", "GT", "GTE"]:
            self.accept()
            self.parse_addition()

    def parse_addition(self):
        """
        term (('+'|'-') term)*
        """
        self.parse_term()
        while self.show_next().tag in ["ADDITION", "SUBTRACTION"]:
            self.accept()
            self.parse_factor()

    def parse_term(self):
        """
        factor (('*'|'/'|'%') factor)*
        """
        self.parse_factor()
        while self.show_next().tag in ["MULTIPLICATION", "DIVISION", "MODULO"]:
            self.accept()
            self.parse_factor()

    def parse_factor(self):
        """
        ('-'|'!')? primary
        """
        if self.show_next().tag in ["SUBTRACTION", "NOT"]:
            self.accept()
        self.parse_primary()

    def parse_primary(self):
        """
        identifier ('[' expression ']')? | literal | parenth
        """
        if self.show_next().tag == "IDENTIFIER":
            self.expect("IDENTIFIER")
            if self.show_next().tag == "L_SQ_BRACKET":
                self.expect("L_SQ_BRACKET")
                self.parse_expression()
                self.expect("R_SQ_BRACKET")
        elif self.show_next().tag in [
            "STRING",
            "INT_NUMBER",
            "FLOAT_NUMBER",
            "KW_TRUE",
            "KW_FALSE",
        ]:
            self.accept()
        elif self.show_next().tag == "L_PAREN":
            self.parse_parenth()
```

```
def parse_parenth(self):
    """
    '(' expression ')'
    """
    self.expect("L_PAREN")
    self.parse_expression()
    self.expect("R_PAREN")
```
