Rapport

Algoritmer och Datastrukturer, 7,5 hp
HT18

Projekt

Lisp-interpreter

av

Anton Lundström(931030-6692)

Akademin för teknik och miljö
Avdelningen för industriell utveckling, IT och samhällsbyggnad

Högskolan i GävleS-801 76 Gävle, Sweden

Datorpost:

anton.lundstroem@hotmail.se

Innehållsförteckning

1. Inledning

Denna rapport kommer behandla skapandet och implementation av en, om än begränsad, interpretator för programmeringsspråket Lisp. Vidare kommer rapporten mer ingående förklara de delar som ingår hos en interpretator samt varje dels specifika uppgift.

1.1. Lisp

Lisp är ett programmeringsspråk som har som har sina rötter så långt tillbaka som år 1958. Det som karaktäriserar Lisp är dess nyttjande av prefixnotation och parenteser [1]. Prefixnotation, även kallad polsk notation, är en av tre sätt att skriva matematik och logik. De andra två notationsformerna är av typen infix och postfix. Prefixnotation innebär att operatorn skrivs först för att sedan följas av dess operander.

1.2. Lexer

Lexern är den första delen av en interpretator eller kompilator. Dennes uppgift är att ta den källkod som ska tolkas och skapa tokens utifrån vad som läses in. Dessa tokens innehåller ett *värde* och en *typ*. Skulle ett heltal läsas in av lexern skapas ett token som exempelvis innehåller siffran som värde och strängen *”NumberLiteral”* som typ. Vidare måste lexern även matcha specifika *keywords* med de tecken som läses in. Läses ett tecken eller sträng in som överensstämmer med ett keyword skapas ett förutbestämt token som innehåller ett redan definierat värde samt typ.

1.3. Parser

Parsern tar den lista av tokens som lexern genererat och skapar ett *Abstract Syntax Tree*. Det finns flera olika typer av parsing men den som nyttjades vid implementeringen var av typen prediktiv *parser*. Med prediktiv parsing menas att parsern har koll på nästkommande värde, i en interpretators fall token, och utifrån detta vet vilken *production* som ska användas. En production är en av delarna i vad som kallas ett *context-free-grammar*.

Ett context-free-grammar är en rekursivt sätt att generera strängar, eller i parserns fall noder. I ett context-free-grammar ingår fyra delar; non-terminals, terminals, productions och en start symbol [2]. Dessa delar definieras av den användare som ska implementera språket. Ett context free grammar för en binär operation kan ses i Figur 1.

G_{binOp} = {{expr, binOp, factor}, {op, number, (,)}, productions, expr}

Productions:

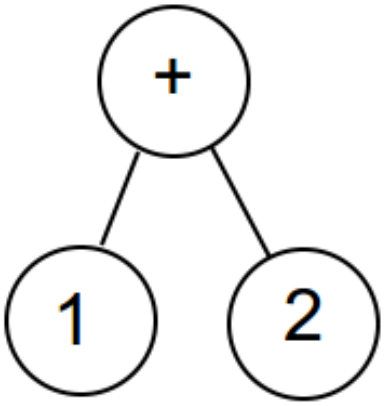
expr :: = (binOp) | ...

binOp :: = op factor factor

factor :: = number | binOp

Figur 1: Context Free Grammar för en binär operation

Parserns uppgift är i grund och botten att, utifrån fördefinierade regler, skapa ett Abstract Syntax Tree. Detta träd är till för att kunna traverseras av evaluatorn eller kompilatorn samt att göra semantiska analyser på. Det abstrakta syntaxträdet för den binära operationen (+ 1 2), som i infixnotation motsvarar 1 + 2 kan ses i Figur 2.



Figur 2: Abstrakt Syntax Träd för operationen 1+2

1.4. Evaluator

Evalutatorns uppgift är att traversera de noder som finns i det abstrakta syntax trädet som skapats av parsern och utvärdera dessa. I fallet för den implementerade evaluatorn traverseras det abstrakta syntax trädet med hjälp av *postorder traversal.* Med detta innebär att, med start från rotnoden, alla löv i trädet utvärderas innan dess förälder utvärderas. I fallet för Figur 2 skulle detta alltså innebära att löven innehållandes siffran ett och två besöks innan dessa evalueras i dess föräldernod.

En annan typ av abstrakt datatyp som hanteras av evaluatorn är ett *Symbol Table*. Denna datatyp håller koll på de variabler och dess typer som definieras av källkoden samt vilket scope dessa hör till.

2. Implementering

2.1. Lexer

Lexern som implementerades har två möjligheter att initieras. Antingen via en tom initiering av konstruktorn eller via att en `Iterator` av typen `Character` passeras in som parameter. För att möjliggöra avläsningen av kod från en fil skapades klassen *FileParser* vars metod `charsInFileIterator()` returnerar just den Iterator som krävs för att kunna iterera över samtliga tecken i källkoden. Vidare implementerades även en klass `PeekableIterator` som implementerar interfacet Iterator. Detta gjordes för att Javas egna implementering ej stödjer *peek*, alltså att titta på nästa element i Iteratorn utan att faktiskt flytta till nästa element.

För att skapa de tokens som ska återspegla källkoden skapades metoden lex(). Detta görs genom att via *regular expressions* matcha varje karaktär i källkoden mot ett fördefinierat regular expression och utifrån detta skapa ett token med korrekt typ och värde (se Figur 3).

```
public List<Token> lex(){

    while(peekIter.hasNext()){
        ch = peekIter.next();

        if (ch == '"' || ch == '\'' ){
            tokenize("StringLiteral", scanString(ch));

        } else if (Pattern.matches("\\>|\\<|\\(|\\)|\\{|\\}|\\,|\\;|\\=|\\:", ch.toString())){
            tokenize(scanComparator(ch), "");

        }  else if (Pattern.matches("\\+|\\-|\\/|\\*", ch.toString())){
            tokenize("Operator", ch.toString());

        } else if (Character.isDigit(ch)){
            tokenize("NumberLiteral", scanInteger(ch));

        } else if (Character.isLetter(ch)){
            tokenize(dict.matchKeyword(scanSymbol(ch)));

        } else {
            // Ignore whitespace and line endings
        }
    }
    tokenize("EOF", "EOF");
    return tokenList;
}
```

Figur 3: Visar metoden lex() i Lexern som ansvarar för att skapa tokens

Det mest intressanta är den näst sista if-satsen som kontrollerar om karaktären som skannas in är en bokstav. Det som inträffar då är att alla efterföljande tecken, fram tills en icke-bokstav stöts på, skannas in till en sträng och sedan matchas mot ett lexikon med fördefinierade keywords. Skulle teckenkombinationen ej matcha ett keyword returneras ett defaulttoken med ”Symbol” som typ och strängen som värde. När samtliga tecken i inputen itererats över läggs ett *end of file*- token på listan för att indikera att datat är slut. Till sist returneras listan med tokens.

2.2. Parser

Parsern implementerades med tidigare nämnda egenskapen av rekursiv parsing i åtanke. De productions som implementerades kan se i Bilaga A. Ett exempel på hur en av dessa rekursiva metoder kan se ut ses i figur 4. Denna figur visar implementationen för *factor*. Denna metod returnerar en Nod som sedan används i det abstrakta syntax trädet baserat på vilka token denne stöter på. För detta skapades en abstrakt klass *Node* som övriga noder sedan ärvde av. Detta möjliggjordes med hjälp av polymorfism eftersom olika typer av noder returneras beroende på vilken production som används.

```
    // factor ::= NumberLiteral | ( binOp ) | op factor | symbol
private Node factor(){
    Node factor = null;

    if (cTypeEquals("Operator")){
        Token cToken = cToken();
        if (cValueEquals("+") || cValueEquals("-")){
            match("Operator");
            factor = new UnaryOpNode(cToken, factor());
        }

    } else if(cTypeEquals("NumberLiteral")){
        factor = new NumberLiteralNode(cToken());
        match("NumberLiteral");

    } else if (cTypeEquals("(")){
        match("(");
        factor = binOp();
        match(")");
    } else {
        factor = symbol();
    }

    return factor;
}
```

Figur 4: Implementationen av metoden factor()

Metoden *match()* är hjärtat för hela parsern. Det är med hjälp av denna metod parsern särskiljer terminals från non-terminals och itererar bland de tokens lexern skapade. För varje förväntad terminal i metoderna ”matchas” detta mot det faktiska nästa värde i iteratorn. Stämmer dessa värden överens, så de matchar, flyttas pekaren ett steg fram så den pekar på nästa token, matchar de inte kastas ett felmeddelande. Detta felmeddelande är ett syntax error, eftersom användaren då har skrivit något tokigt och ordningen för tokens ej matchar någon production (se Figur 5).

```
private void match(String tokenType){
    if (cTypeEquals(tokenType)){
        this.currentToken = peeker.next();
    } else {
        throw new SyntaxErrorException("Expected: " + tokenType + " but got: " + cTokenType());
    }
}
```

Figur 5: Implementationen av metoder match()

2.3. Evaluator

För att möjliggöra evalueringen av Noder implementerades ett Reflective Visitor Pattern [3] i form av klasserna Visitor, NodeVisitor samt interfacet Ivisitable. Klassen Ivisitable är ett tomt interface. Detta interface implementeras av den abstrakta klassen Node för att möjliggöra ”besök” till dess subklasser. NodeVisitor är det som sköter evalueringen av noder med hjälp av *overloading*. På detta sätt kan operationerna för varje enskild nod definieras i egna metoder (se Figur 6).

```
public void evaluate(NumberLiteralNode node) {
    result = Integer.parseInt(node.getValue());
}

public void evaluate(SymbolNode node){
    String symbolName = node.getSymbol();
    int value = st.lookup(scope, symbolName);
    result = value;
}
```

Figur 6: Några av de olika evaluate() metoderna som nyttjar overloading

I vissa metoder i klassen NodeVisitor finns metoden `visit()`. Denna metod nyttjas för att besöka nodens löv. Implementeringen av denna metod gjordes i klassen Visitor.

2.4. Symbol Table

För att kunna mappa variablers namn till dess värde i runtime och samtidigt kunna hålla koll på vilket scope dessa tillhör implementerades ett *Symbol Table.* Detta symbol-table var ej lika fullfjädrat som ett riktigt symbol table. Det som lagrades i implementationen var följande:

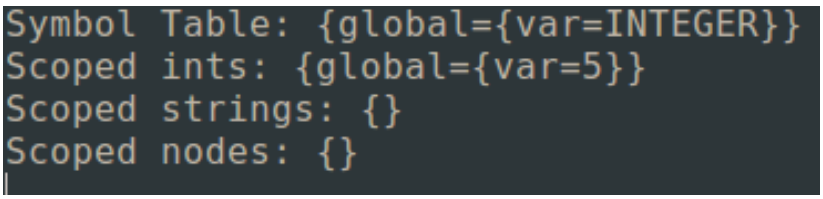
- `HashMap<String, HashMap<String, String>>();`
- `HashMap<String, HashMap<String, Integer>>();`
- `HashMap<String, HashMap<String, String>>();`
- `HashMap<String, HashMap<String, Node>>();`

Den första HashMapen ansvarar för vilket scope, symbolnamn samt typ som deklarerats i koden. De resterande ansvarar för scopet, symbolnamn samt nummer, strängar och noder. När en exempelvis en `NumberLiteralNode` stöts på av evaluatorn nyttjas metoden `lookup(String scope, String symbol)`. Det görs då en kontroll på att denne symbol finns inom det scopet som passerats in, sedan returneras korrekt typ baserat på hur den tidigare definierats. En nackdel med implementationen är att HashMapsen aldrig rensas på värden, har en variabel A tillägnats siffran tre kommer den alltid finnas kvar inom det givna scopet. Det går att ändra värde på denna men det rensas ej.

3. Beskrivning av slutresultat

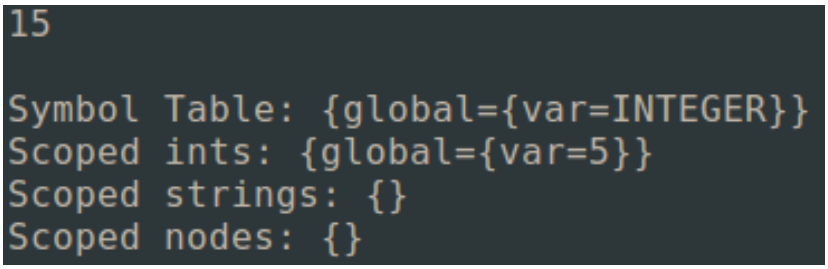
3.1. Definiera variabler

För att definiera variabler används *defvar.* Figur 7 visar den output samt symbol table då funktionen `(defvar var 5)` evaluterats. Eftersom endast en variabel definieras ska ingenting printas, därav syns endast dess symbol table.



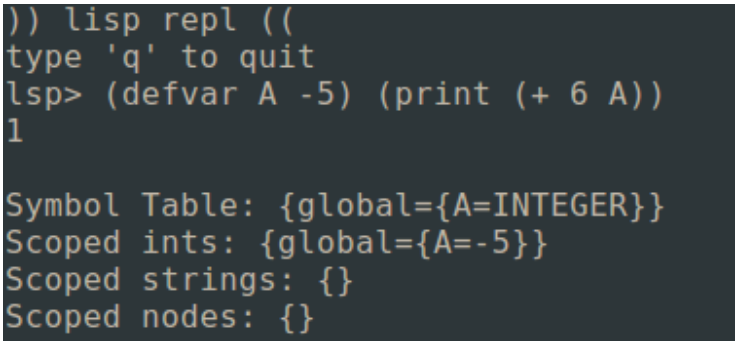
Figur 7: Innehållet i symbol tablet när strängen (defvar var 5) evaluerats

Anges strängen `(defvar var 5) (print (+ var 10))` skrivs dock det förväntade värdet 15 ut (se Figur 8).



Figur 8: Uträkning till det förväntade värdet 15 samt det nuvarande symbol tablet

Även negativa tal stöds, anges till exempel strängen `(defvar A -5) (print (+ 6 A))`. Skrivs värdet 1 ut.



Figur 9: Resultat av input samt det nuvarande symbol tablet

3.2. Definiera funktioner

För att definiera funktioner används *defun*. Figur 10 visar symbol tablet när strängen `(defun func (+ 5 func))` angetts.

```
)) lisp repl ((
type 'q' to quit
lsp> (defun func (+ 5 func))
Symbol Table: {global={func=FUNCTION}}
Scoped ints: {}
Scoped strings: {}
Scoped nodes: {global={func=ast.BlockNode@46fbb2c1}}
```

Figur 10: Innehållet i symbol tablen när funktionen definierats

För att sedan använda funktionen anges `(functionName Number)`. Anges exempelvis strängen `(func 10)` som input resulterar det i 15. Anges `(print (func 10))` skrivs detta även ut (se Figur 11).

```
lsp> (print (func 10))
15

Symbol Table: {func={func=INTEGER}, global={func=FUNCTION}}
Scoped ints: {func={func=10}}
Scoped strings: {}
Scoped nodes: {global={func=ast.BlockNode@1698c449}}
```

Figur 11: Resultatet av utskriften för funktionen func

4. Diskussion

Ett extremt roligt och insiktsfullt projekt som skrapade på ytan för hur en kompilator eller interpreter fungerar och skapas. Det finns dock en del buggar och icke-förväntade resultat i den skapade interpretatorn. Exempelvis vid nyttjande av fler ”print” skrivs samtliga, även tidigare definierade print-statements ut på skärmen. Detta är ett problem som främst uppstår vid användningen av repln. Detta för att det vid skapandet av det abstrakta syntaxträdet läggs en PrintNode som tar ett block som barn. Denna PrintNode försvinner aldrig från trädet vilket resulterar i att om fler print-statements anges läggs det till fler PrintNodes som besöker sina barn gång på gång.

Hade mer tid funnits tillhanda hade interpretatorn utvidgats med en riktigt implementation av symbol-table samt vissa av de productions som fanns skulle skrivas om. Även en större andel av Lisps funktioner hade då implementerats.

5. Innehållsförteckning

[1] ”Lisp (programming language),” [Online]. Available: https://en.wikipedia.org/wiki/Lisp_(programming_language) [Accessed: 10-Jan-2019].

[2] ”Context Free Grammars.” [Online]. Available: https://www.cs.rochester.edu/~nelson/courses/csc_173/grammars/cfg.html. [Accessed: 10-Jan-2019].

[3] Y. Mai and M. Champlain, *”Reflective Visitor Pattern”*, Department of Electrical and Computer Engineering, Concodria University, 2001.

6. Bilaga A

Productions för grammatiken i parsern, liten bokstav innebär non-terminal, stor bokstav betyder terminal.

Program ::= block EOF

block ::= (statements)

statements ::= statement | statement statements

statement ::= block | expr

expr ::= assignment | proc | print | binOp | procCall | symbol | factor | noOp

assignment ::= Function symbol factor

proc ::= Function procSymbol block

print ::= Function block noOp

binOp ::= Operator factor factor

procCall ::= symbol expr

factor ::= NumberLiteral | (binOp) | op factor | symbol

symbol ::= VarSymbol | ProcSymbol | Symbol

noOp ::= NoOp