# Visitors - Switches

## Τάξεις Java που παράγονται από το SableCC

Στη λεκτική ανάλυση, στην οποία δεν έγινε αναφορά εδώ, παράγονται διάφορες τάξεις μεταξύ των οποίων μία για κάθε token με όνομα TOnomaToken και η τάξη του λεκτικού αναλυτή Lexer. Δεν θα αναφερθούμε διεξοδικά στο Lexer ή στον Parser, που κάνει τη συντακτική ανάλυση, απλά αναφέρονται για σύνδεση με τα παρακάτω.

Αυτό που ισχύει γενικά στα ονόματα των τάξεων που παράγει το SableCC είναι ότι οι κάτω παύλες φεύγουν, αλλά έχουν ως αποτέλεσμα το γράμμα που τις ακολουθεί να γίνει κεφαλαίο, ενώ όλα τα ονόματα έχουν ένα πρόθεμα:

T για token κουπόνι

P για production παραγωγή

A για alternative εναλλακτική.

Ακόμα, τα ονόματα κάθε εναλλακτικής, ακολουθούνται από το όνομα της παραγωγής τους.

Κορυφή (σχεδόν) της ιεραρχίας είναι η τάξη Node, δηλαδή ένας κόμβος.

Όλα τα tokens είναι υποσύνολα της τάξης Token, που με τη σειρά της είναι Node.

Κάθε Alternative είναι υποτάξη της αντίστοιχης Production.

Η τάξη Node, και συνεπώς όλες οι υποτάξεις της, υλοποιούν το interface Switchable που περιέχει τη συνάρτηση apply. Αργότερα θα εξηγηθεί ποιος ακριβώς είναι ο ρόλος αυτής της συνάρτησης, προς το παρόν όμως αυτό που μπορεί να ειπωθεί είναι ότι αναλαμβάνει την επεξεργασία του κόμβου, χωρίς στην πραγματικότητα να περιέχει η ίδια κάποιο ειδικό κώδικα. Έτσι κάθε κόμβος είτε πρόκειται για T είτε για P είτε για A θα περιέχει τη δική του apply.

## Switches

Το SableCC κατασκευάζει αυτόματα τάξεις που αναλαμβάνουν τη διάσχιση ενός συντακτικού δέντρου, είτε πρόκειται για αφηρημένο είτε για συγκεκριμένο. Τις τοποθετεί μέσα στο φάκελο analysis. Αυτές βασίζονται στη μέθοδο / σχέδιο προγραμματισμού (design pattern) που λέγεται «Επισκέπτης» (Visitor). Στο SableCC αυτό το ονομάζει Switch, επηρεασμένος από τη δομή ελέγχου switch με τα αντίστοιχα case, έτσι όπως είναι γνωστή από το δομημένο προγραμματισμό. Προς αυτό είναι αναγκαία και η ύπαρξη του interface Switchable και της μεθόδου apply.

Η ιεραρχία των τάξεων στο πακέτο analysis συνοψίζεται ως εξής:

*Switch* > *Analysis* > AnalysisAdapter > DepthFirstAdapter

Τα δύο πρώτα είναι interfaces δηλαδή – απλουστευμένα - δεν έχουν υλοποιήσεις στις μεθόδους.

Παρέχονται και άλλες τάξεις για να υπάρχει εύρος επιλογών, αλλά τελικά η μόνη που θα χρειαστούμε είναι η DepthFirstAdapter η οποία, όπως δηλώνει το όνομα της, αναλαμβάνει την κατά βάθος διάσχιση ενός δέντρου. Θα φτιάχνουμε υποτάξεις αυτής, με στόχο να υποσκελίζουμε, αντικαθιστούμε, αναιρούμε, ακυρώνουμε (override) τις μεθόδους που θέλουμε να κάνουν συγκεκριμένα πράγματα. Αυτό όμως θα αναλυθεί αργότερα. Προς το παρόν θα μείνουμε στη DepthFirstAdapter καθώς είναι σημαντική η μελέτη της.

Ο τρόπος που δουλεύει η DepthFirstAdapter αλλά και κάθε *Analysis* είναι ο εξής:

Μέσα στην Analysis υπάρχει μια μέθοδος caseZZZ(ZZZ node) για κάθε είδος (υποτάξη) κόμβου που μπορούμε να έχουμε, υπό το όνομα ΖΖΖ. Αυτή, όπως εύστοχα υπονοεί το όνομα της λειτουργεί όπως το “case:”, δηλαδή καθορίζει το τι ακριβώς θα συμβεί όταν επεξεργαζόμαστε κόμβο τύπου ΖΖΖ. Ακόμα, υπάρχουν οι in… out… τις οποίες η case καλεί κατά την είσοδο και την έξοδο αντίστοιχα.

Αρχικά καλείται η μέθοδος apply για τον κόμβο ρίζα, έστω ονομαζόμενο Root. Αυτή η μέθοδος αναλαμβάνει να καλεί την αντίστοιχη μέθοδο caseRoot(Root node). Μέσα, αυτή η μέθοδος θα έχει κώδικα που θα κάνει τη διάσχιση του δέντρου, με τον εξής τρόπο. Για κάθε παιδί του κόμβου ρίζα καλεί τη συνάρτησή του visit(Switch). Φαίνεται να μην ξέρουμε ποιος μπορεί να είναι ακριβώς ο τύπος κάθε παιδιού. Επειδή όμως εφαρμόζεται ο πολυμορφισμός, κατά τη στιγμή της εκτέλεση καλείται η συνάρτηση του πραγματικού τύπου του κάθε κόμβου. Εκεί μέσα προφανώς, είναι γνωστό τι τύπου είναι ο κόμβος, έστω ΥΥΥ, και καλείται η ορθή μέθοδος caseYYY(…). Αυτό επαναλαμβάνεται μέχρι να φτάσουμε στα φύλλα του δέντρου.

Για να αναφερθούμε σε ένα συγκεκριμένο παράδειγμα:

Έστω η τάξη AandExpression. Μέσα σε αυτήν θα δούμε τη συνάρτηση

    public void apply(Switch sw)  
    {  
        ((Analysis) sw).caseAAndExpression(this);

    }

Αυτή η μέθοδος παίρνει ως παράμετρο ένα Switch( που μπορεί να είναι DepthFirstAdapter) και καλεί τη μέθοδο caseAAndExpression… που αντιστοιχεί σε αυτήν, περνώντας τον εαυτό της ως παράμετρο, αναγνωρίζοντας έτσι τον εαυτό της.

interface Switch{

    //...

    public void caseAAndExpression( AAndExpression e);

    //....

}

Η μέθοδος που θα κληθεί θα ξέρει τον τύπο του αντικειμένου που δέχεται ως παράμετρο, αφού μόνο έναν τύπο δέχεται ως παράμετρο. Έτσι όταν έρθει η ώρα της υλοποίησης αυτών των συναρτήσεων, θα καλείται μέσα στη visit του ΑAndExpression για παράδειγμα το

e.getLeft().apply( this);

Τι σημαίνει αυτό: σημαίνει ότι εμείς μεν δεν ξέρουμε τι τύπου είναι η αριστερή υποέκφραση. Όμως όλες οι τάξεις των εκφράσεων υλοποιούν το κοινό interface Switchable που απλά έχει τη συνάρτηση apply. Εδώ είναι που μπαίνει ο πολυμορφισμός και επειδή η αριστερή υποέκφραση ξέρει τι τύπος είναι ο εαυτός της και η Java ξέρει τι τύπου είναι η αριστερή υποέκφραση, καλείται η σωστή συνάρτηση apply. Δηλαδή, η συνάρτηση apply του πραγματικού τύπου – που βρίσκεται κάτω κάτω στην ιεραρχία των τάξεων. Έστω, χάριν του παραδείγματος, ότι ο τύπος είναι στην πραγματικότητα IdentifierExpression. Ο έλεγχος του προγράμματος θα μεταβεί συνεπώς στην συνάρτηση apply της IdentifierExpression. Αυτή δε θα κάνει τίποτα άλλο, παρά να επαναφέρει τον έλεγχο στην τάξη Switch( DepthFirstAdapter ή ό,τι άλλο) αυτή τη φορά όμως, καλώντας τη σωστή, για το δικό της τύπο, συνάρτηση caseAIdentifierExpression. Αν όχι για κανένα άλλο λόγο, επειδή περνάει τον εαυτό της ως παράμετρο. Έτσι η Java αποφασίζει να καλέσει τη συνάρτηση της οποίας η ταυτότητα ταιριάζει με την πραγματική παράμετρο.

Όλα αυτά για τη θεωρητική εξήγηση των τάξεων που μας δίνει το SableCC. Στην πράξη τώρα, αυτό που μας ενδιαφέρει είναι τα φτιάξουμε μια δική μας τάξη που θα κληρονομεί την τάξη DepthFirstAdapter. (class MyClass extends DepthFirstAdapter).Μέσα σε αυτή την τάξη και για όσους κόμβους μας ενδιαφέρει θα υλοποιήσουμε τις δικές μας μεθόδους case, in, out που θα κάνουν κατά περίπτωση συλλογή συμβόλων, σημασιολογική ανάλυση, παραγωγή κώδικα ή ό,τι άλλο μπορεί να θέλουμε.

Για παράδειγμα μπορεί να θέλουμε να ελέγξουμε ότι οι δύο υποεκφράσεις μιας έκφρασης πρόσθεσης είναι ακεραίου τύπου, έτσι ώστε η πρόσθεση να έχει νόημα. Επισκεπτόμενοι τον κόμβο της πρόσθεσης θα επισκεφτούμε τον κάθε ένα κόμβο παιδί και θα διαπιστώσουμε τι τύπου στη (Mini)Java είναι. Μετά θα επιστρέψουμε στη μέθοδο case για την πρόσθεση και βγάλουμε ένα μήνυμα σφάλματος αν δεν είναι και οι δύο τύποι ακέραιοι. Τέλος θα ενημερώσουμε και τον γονέα του κόμβου της πρόσθεσης, ότι αυτός ο κόμβος είναι ακεραίου τύπου.