```
1  public int compare (int a,int b) {
2     if (a > b) return a;
3     else return b;
4  }
```

Fig. 8. A Java code snippet s_1.

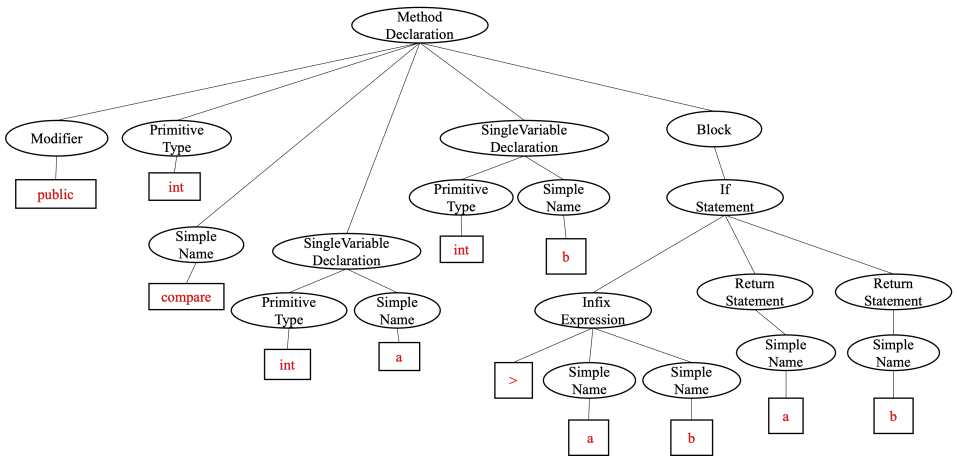

Fig. 9. The AST generated by JDT for code snippet s_1.

A APPENDIX

A.1 Example of AST.

Fig. 9 shows an example of AST, which is generated by JDT (an AST parsing method detailed in Section 3.2) for the Java code snippet s_1 shown in Fig. 8. In the AST shown in Fig. 9, nodes like `MethodDeclaration`, `SingleVariableDeclaration` and `IfStatement` are non-terminals; nodes like `Modifier`, `PrimitiveType` and `SimpleName` are terminals; and leaf nodes like `public`, `int` and `compare` are the corresponding values of the terminals. It is evident that, compared to the source code of s_1, its AST is significantly more complex. In addition, as mentioned in Section 1, different AST parsing methods (e.g., ANTLR and Tree-sitter) would generate distinct ASTs for the same source code due to different lexical rules and grammar rules. Therefore, this paper focuses on investigating the current progress of feature engineering and application of AST, aiming to provide guidance for subsequent researchers on how to effectively leverage complex ASTs to enhance code representation and subsequent code-related tasks.

A.2 Example of data produced by AST preprocessing methods.

Fig. 10 shows a piece of Java code snippet s_2. Fig. 11 shows the BFS, SBT, and AST Path of code snippet s_2 and Fig. 12 shows the Raw AST, Binary Tree, and Split AST of s_2. The complete information contained in an AST consists of its nodes and structures. Different AST preprocessing methods differ in retaining the node and structure information of the AST. As illustrated in Fig. 11 and Fig. 12, SBT and Raw AST retain all nodes and complete structure information. BFS also preserves every node of the AST while disregarding much of the structure information. Binary tree, converting the raw AST into a binary tree and merging nodes with only one child node with their child nodes,

```
1 private String postXml() {
2     try {
3         URLConnection conn = new URL(url).openConnection();
4     }
5 }
```

Fig. 10. A Java code snippet s_2

Fig. 11. BFS, SBT, AST Path of the code snippet s_2. Using JDT as the AST parser.

removes redundant intermediate nodes, and retains complete structure information. AST Path and Split AST divide the complete AST into smaller components, facilitating model learning, but it comes at the cost of losing partial node or structure information of the AST. It is worth noting that AST Path, due to constraints on the input size of the encoding models, is often limited in width, length, and quantity. Consequently, a significant portion of node and structure information tends to be discarded. Similarly, during the process of converting the source code into the Split AST [68], some statements in the source code (e.g., `catch` clause) are not added to the split code set, leading to the loss of node information.

A.3 Technical details of AST encoding methods.

A.3.1 Sequence models.

(i) Bidirectional Long Short Term Memory (BiLSTM).

The LSTM architecture [35] addresses the problem of learning long-term dependencies of RNN by introducing a memory cell that can preserve state over long periods of time [77]. In the work [77], the LSTM unit at each time step t is defined to be a collection of vectors in $\mathbb{R}^d$ ($\mathbb{R}$ is the set of real numbers, and d is the memory dimension of the LSTM): an input gate i_t, a forget gate f_t, an output

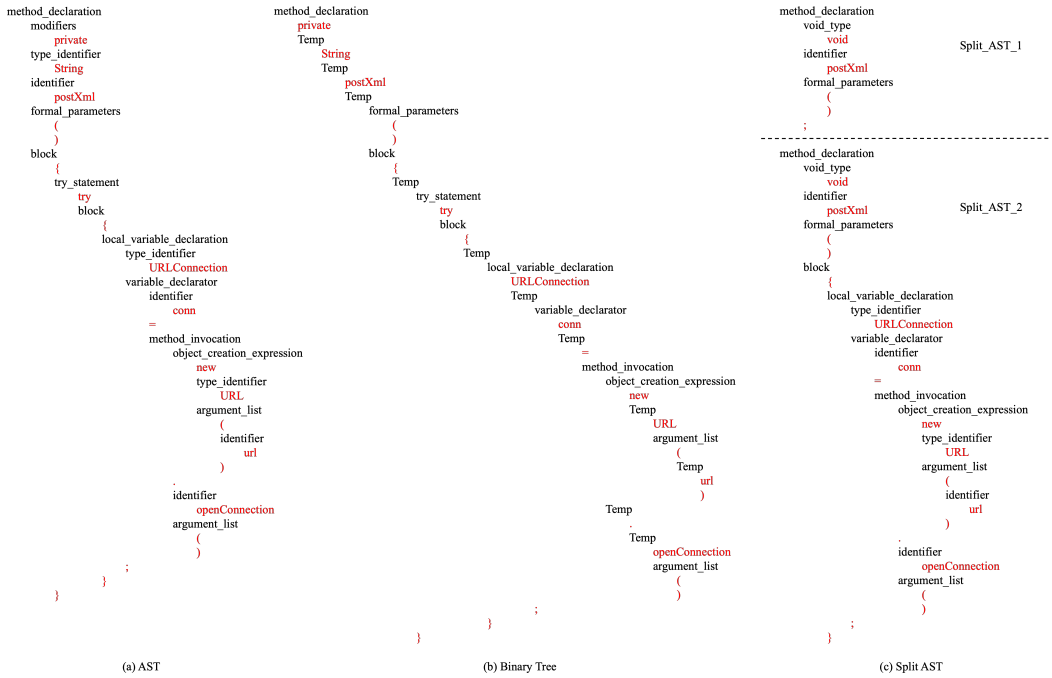

Fig. 12. Raw AST, Binary Tree, and Split AST of the code snippet s_2. Using Tree-sitter as the AST parser.

gate o_t, a memory cell c_t and a hidden state h_t. The LSTM transition equations are the following:

$$\begin{aligned} i_t &= \sigma\left(W^{(i)}x_t + U^{(i)}h_{t-1} + b^{(i)}\right), & f_t &= \sigma\left(W^{(f)}x_t + U^{(f)}h_{t-1} + b^{(f)}\right), \\ o_t &= \sigma\left(W^{(o)}x_t + U^{(o)}h_{t-1} + b^{(o)}\right), & u_t &= tanh\left(W^{(u)}x_t + U^{(u)}h_{t-1} + b^{(u)}\right), \\ c_t &= i_t \odot u_t + f_t \odot c_{t-1}, & h_t &= o_t \odot tanh(c_t) \end{aligned} \tag{14}$$

where x_t is the input at the current time step; W, U are the weighted metrics; b is the bias vector; σ denotes the logistic sigmoid function; $tanh$ denotes the hyperbolic tangent function; $\odot$ denotes element-wise multiplication.

Bidirectional LSTM (BiLSTM) [66] uses two LSTMs at each layer. One LSTM takes the original sequence as input and the other takes the reversed sequence as input. So it is capable of modeling the sequential dependencies between words and phrases in both directions of the sequence. The h_t at the final timestamp can be used as the code representation denoted $\mathbf{z} = h_{t_{final}}$.

(ii) Transformer.

Transformer [81] follows an encoder-decoder structure, as well as utilizing a multi-headed self-attention mechanism and positional encoding to draw global dependencies between input and output. For the code clone detection and the code search task, the decoder part of Transformer is not used since a decoder is not needed. For the code summarization task, both encoder and decoder are used.

Encoder [67]: Encoder combines multiple identical layers where each layer consists of two sub-layers. The first sub-layer forms a multi-headed self-attention structure while the other one is a fully connected layer. Both sub-layers are followed by another layer which normalizes the output

of each sub-layer. The encoder maps an input sequence of symbol representations $\mathbf{x} = (x_1, \ldots, x_n)$ to a sequence of continuous representations $\mathbf{z} = (z_1, \ldots, z_n)$.

Decoder [67]: Decoder has a similar structure as encoder except that it includes one additional sub-layer. This extra sub-layer conducts multi-head attention on the encoder's output. Moreover, the self-attention sub-layer is reformed to avoid attending to subsequent positions. Given $\mathbf{z}$, the decoder generates an output sequence $\mathbf{y} = (y_1, \ldots, y_m)$ of symbols one element at a time.

Multi-head attention mechanism [67]: Transformer's self-attention aims to map a query (Q) and a collection of key (K) - value (V) pairs to vectors which are calculated as a weighted sum of the values, which is shown in the following formula:

$$Attention(Q, K, V) = softmax(QK^T/\sqrt{d_k})V. \tag{15}$$

where d_k is the dimension of the key vector.

Multi-head attention conducts self-attention process multiple times separately, with different weight matrices. All the results are concatenated and multiplied by an additional weight matrix W, as shown in the following formula:

$$\begin{aligned} MultiHead(Q, K, V) &= Concat(head_1, ..., head_h)W^O \\ \text{where} \quad head_i &= Attention(QW_i^Q, KW_i^K, VW_i^V), \end{aligned} \tag{16}$$

where d_{model} is the input dimension, d_k is the dimension of the key vector, d_v is the dimension of the value vector, $h = 8$ is the number of parallel attention layers, $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^v \in \mathbb{R}^{d_{model} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{model}}$.

Positional Encoding [67]: To retain information regarding relative and absolute tokens' positions, a positional encoding layer is added at the lowest part of the encoder and decoder stacks. There are many choices of positional encodings. In this work, we use sine and cosine functions of different frequencies:

$$\begin{aligned} PE_{(pos,2i)} &= sin(pos/10000^{2i/d_{model}}), \\ PE_{(pos,2i+1)} &= cos(pos/10000^{2i/d_{model}}), \end{aligned} \tag{17}$$

where pos is the position and i is the dimension.

A.3.2 Tree-structured models.

(i) Tree-Structured Long Short-Term Memory Networks (TreeLSTM)

TreeLSTM is first proposed by Tai [77] to capture the syntactic properties of natural language. TreeLSTM is a generalization of LSTMs to model tree-structured topologies. Given a tree, let $C(j)$ denote the set of children of node j. The TreeLSTM unit at each node j is defined to be a collection of vectors in $\mathbb{R}^d$, where $\mathbb{R}$ is the set of real numbers, d is the memory dimension of the TreeLSTM: an input gate i_j, forget gates f_{jk} where $k \in C(j)$, an output gate o_j, a memory cell c_j and a hidden state h_j. The entries of the gating vectors i_j, f_j and o_j are in $[0, 1]$.

Child-Sum TreeLSTMs [77]: The Child-Sum TreeLSTM can be used on tree structures where the number of children is arbitrary and children's orders are not considered. The Child-Sum

TreeLSTM transition equations are the following:

$$\begin{aligned}
&\tilde{h}_j = \sum_{k \in C(j)} h_k, && i_j = \sigma\left(W^{(i)} x_j + U^{(i)} \tilde{h}_j + b^{(i)}\right), \\
&f_{jk} = \sigma\left(W^{(f)} x_j + U^{(f)} h_k + b^{(f)}\right), && o_j = \sigma\left(W^{(o)} x_j + U^{(o)} \tilde{h}_j + b^{(o)}\right), \\
&u_j = tanh\left(W^{(u)} x_j + U^{(u)} \tilde{h}_j + b^{(u)}\right), && c_j = i_j \odot u_j + \sum_{k \in C(j)} f_{jk} \odot c_k,\ h_j = o_j \odot tanh(c_j),
\end{aligned} \tag{18}$$

where x_j is the input at node j, W, U are the weighted metrics, b is the bias vector, σ denotes the logistic sigmoid function, $tanh$ denotes the hyperbolic tangent function, $\odot$ denotes element-wise multiplication. The hidden state h_j at the root node is considered the representation of the source code, that is code representation $\mathbf{z} = h_{root}$.

N-ary TreeLSTMs [77]: The N-ary TreeLSTM can be used on tree structures where the branching factor is at most N and where children are ordered, i.e., they can be indexed from 1 to N. For any node j, write the hidden state and memory cell of its kth child as h_{jk} and c_{jk} respectively. The N-ary TreeLSTM transition equations are the following:

$$\begin{aligned}
&i_j = \sigma\left(W^{(i)} x_j + \sum_{\ell=1}^{N} U_\ell^{(i)} h_{j\ell} + b^{(i)}\right), && f_{jk} = \sigma\left(W^{(f)} x_j + \sum_{\ell=1}^{N} U_{k\ell}^{(f)} h_{j\ell} + b^{(f)}\right), \\
&o_j = \sigma\left(W^{(o)} x_j + \sum_{\ell=1}^{N} U_\ell^{(o)} h_{j\ell} + b^{(o)}\right), && u_j = tanh\left(W^{(u)} x_j + \sum_{\ell=1}^{N} U_\ell^{(u)} h_{j\ell} + b^{(u)}\right), \\
&c_j = i_j \odot u_j + \sum_{\ell=1}^{N} f_{j\ell} \odot c_{j\ell}, && h_j = o_j \odot tanh(c_j),
\end{aligned} \tag{19}$$

where $k = 1, 2, \ldots, N$, x_j is the input at node j, W, U are the weighted metrics, b is the bias vector, σ denotes the logistic sigmoid function, $tanh$ denotes the hyperbolic tangent function, $\odot$ denotes element-wise multiplication. Similar to Child-Sum TreeLSTM, code representation $\mathbf{z} = h_{root}$.

(ii) AST-Trans.

AST-Trans [78] is a simple variant of the Transformer model to efficiently handle the tree-structured AST. AST-Trans exploits ancestor-descendant and sibling relationship matrices to represent the tree structure, and uses these matrices to dynamically exclude irrelevant nodes.

AST-Trans has the same encoder and decoder structure as the Transformer, while replacing the singe-head self-attention with tree-structured attention. The absolute position embedding from the original Transformer is replaced with relative position embeddings defined by the two relationship matrices to better model the dependency.

For an AST, it will be firstly linearized into a sequence, which means being transformed into SBT [37] in our experiment. Then the ancestor-descendent and sibling relationships among its nodes will be denoted through two specific matrices. Based on the matrices, tree-structured attention is adopted to better model these two relationships. In the following part, we will introduce the construction of relationship matrices and tree-structured attention.

Construction of relationship matrices. Two kinds of relationships are defined between AST nodes: ancestor-descendant (A) and sibling (S) relationships. And we use two position matrices $A_{N\times N}$ and $S_{N\times N}$ to represent the ancestor-descendent and sibling relationships respectively, where N is the total number of nodes in AST. The i-th node in the linearized AST is denoted as n_i. A_{ij} is the distance of the shortest path between n_i and n_j in the AST. S_{ij} is horizontal sibling distance between n_i and n_j if they satisfy the sibling relationship. If one relationship is not satisfied, its

value in the matrix will be infinity. Note that we consider the relative relationship between two nodes, which means $A_{ij} = -A_{ji}$ and $S_{ij} = -S_{ji}$ if a relationship exists between n_i and n_j.

Formally, we use $SPD(i, j)$ and $SID(i, j)$ to denote the Shorted Path Distance and horizontal Sibling Distance between n_i and n_j. The values in the relationship matrices are defined as:

$$A_{ij} = \begin{cases} SPD(i,j) & \text{if } |SPD(i,j)| \leq P \\ \infty & \text{otherwise} \end{cases} \tag{20}$$

$$S_{ij} = \begin{cases} SID(i,j) & \text{if } |SID(i,j)| \leq P \\ \infty & \text{otherwise} \end{cases} \tag{21}$$

P is a pre-defined threshold and nodes with relative distance beyond P will be ignored. We set $P = 7$ according to the code provided by the original paper [78].

Tree-structured Attention. Tree-structured attention is built on standard self-attention with relative position embeddings and disentangled attention. Tree-structured attention transforms an input sequence $\mathbf{x} = (x_1, ..., x_n)$ ($x_i \in \mathbb{R}^d$ which stands for the embedding of n_i) into a sequence of output vectors $\mathbf{o} = (o_1, ..., o_n)$ ($o_i \in \mathbb{R}^d$).

The relative distance defined under the linear relationship is replaced with $\delta_R(i, j)$ where R stands for either the ancestor-descendent relationship A or the sibling relationship B in the tree structure. $\delta_R(i, j)$ reflects the pairwise distance between n_i and n_j in relationship R. Denote P as the max relative distance, $\delta_R(i, j)$ is defined as:

$$\delta_R(i,j) = \begin{cases} R_{ij} + P + 1 & \text{if } R_{ij} \in [-P, P] \\ 0 & \text{if } R_{ij} = \infty \end{cases} \tag{22}$$

R_{ij} refers to either A_{ij} defined in Eq 20 or S_{ij} defined in Eq 21.

As there are two kinds of relationships, each head only considers one relationship so that it will not add any additional parameter on top of the standard Transformer. h_A heads will use $\delta_A(i, j)$ and the rest h_S heads will use $\delta_S(i, j)$. Information from the two relationships will be merged together through multi-head attention. The output vector $\mathbf{o} = (o_1, ..., o_n)$ is computed as below:

$$\alpha_{i,j} = Q(x_i)K(x_j)^T + Q(x_i){K^P_{\delta_R(i,j)}}^T + Q^P_{\delta_R(j,i)}K(x_j)^T \tag{23}$$

$$o_i = \sum_{j}^{j \in \{j | \delta_R(i,j) > 0\}} \sigma(\frac{\alpha_{i,j}}{\sqrt{3d}})(V(x_j) + V^P_{R_{ij}}) \tag{24}$$

where $Q, K : \mathbb{R}^d \to \mathbb{R}^m$ are query and key functions respectively, $V : \mathbb{R}^d \to \mathbb{R}^d$ is a value function, σ is a scoring function (e.g. softmax or hardmax), $Q^P, K^P \in \mathbb{R}^{(2P+1)\times m}$ represent the query and key projection matrices of relative positions,V^P represents the value project matrix of relative distances, $K^P_{\delta_R(i,j)}$is the $\delta_R(i, j)$-th row of K^P and $Q^P_{\delta_R(i,j)}$is the $\delta_R(i, j)$-th row of Q^P, $V^P_{R_{ij}}$is the R_{ij}-th row of V^P. Note that only the attention weights for node pairs where $\delta_{R(i,j)} > 0$ are computed.