Semantic Analyzer

xpojez00, xbudin05

December 3, 2021

1 Introduction

Whenever a rule is applied we need to check if semantic attributes match accordingly. Semantic analyzer should be called only upon expressions and operations involving them, i.e. assignment, return values, function calls etc. With creation of semantic action upon a rule applied we are closely tied to generating the final code, which can be done from Abstract Syntax Tree. Tree is generated from expressions currently.

2 Structure

```
NodeType = {OPERATION, ID, FUNCTION, VALUE, VOID, NIL, POP, IF, ELSEIF, ELSE,
    NODE_WHILE, FUNCTION_DEFINITION, FUNCTION_POP, DECLARE, ASSIGN, LVALUES,
    RVALUES, RETURN}

SemanticType = {INTEGER, NUMBER, BOOLEAN, STRING, VOID, NIL}

struct Node {
    NodeType nodeType
    void *data
    vector *sons
    vector *returns
    SemanticType semanticType
    OperationType operation
    int attribute
}
```

Note: attribute can be used by Nodes, which need to remember for example a unique ID assigned to them or in case of function calls[1]

3 Abstract Syntax Tree

We have decided to create a *m-n-tree*. This tree has m sons as incoming parameters and in case of function we have also n sons as function returns. Whenever we want to call Node finalized i.e. ready to be generated into IFJcode21, we must first finalize all m sons before. In case of function we should generate n sons before finalization and before we check the semantic correctness. Meaning if Semantic types our parent expects are the same with our return values. This was special case for functions. In all other operators we simply just check if all our sons are of the same SemanticType and then the operation itself gains a SemanticType for itself. Most of the time it is logical sum i.e.

$$node.semanticType = \sum_{k=1}^{m} node.sons[k].semanticType$$

[1] Presume function call A has as the last argument function call B, we might need more than one return value from function call B as parameters of the function call A

In case of node being function, SemanticType defaultly equals to VOID unless it returns at least one value. Then the first returned value is the SemanticType of function.

Note: Semantic type in casual statements has no meaning this is why we include VOID as semantic type.

4 Creating boolean expression out of non-boolean expression

If we end with an expression of non-boolean type we create new expression, which is of OPERATION type and operator equals to ==. We append as sons original expression and *nil node*. This is default behaviour defined by the Teal language.

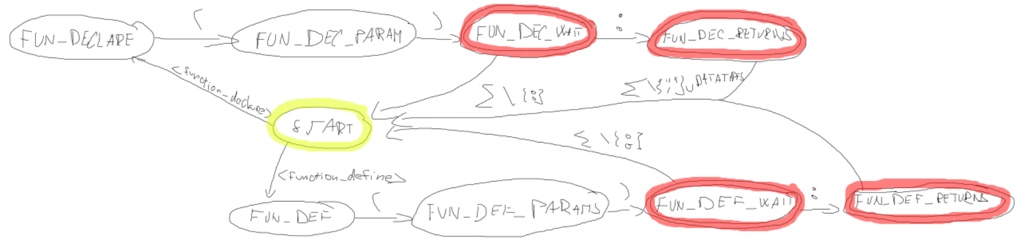
5 Possible Operators

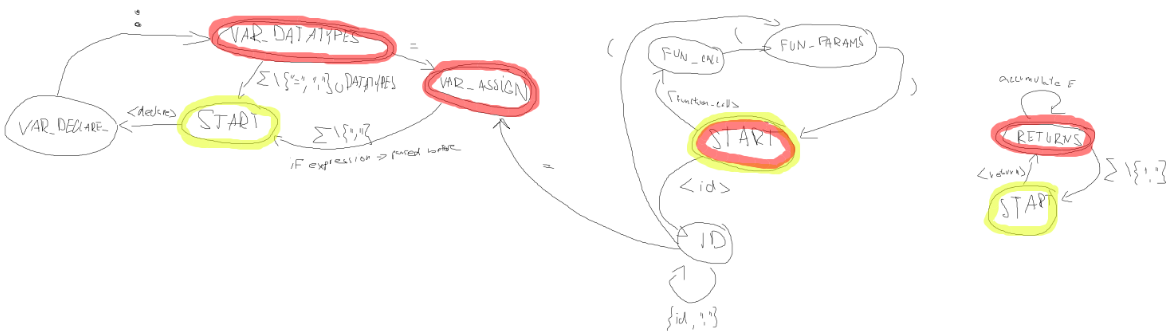
Set of operators	Set of possible semantic values
$\{\#\}$	$\{STRING\}$
$\{\textbf{not}\}$	$\{BOOLEAN\}$
$\{*, /, +, -\}$	$\{INTEGER, NUMBER\}$
$\{//\}$	$\{INTEGER\}$
$\{..\}$	$\{STRING\}$
$\{>, <, >=, <=\}$	$\{INTEGER, NUMBER\}$
$\{==, \sim=\}$	$\{INTEGER, NUMBER, BOOLEAN, STRING\}$
$\{\textbf{and}, \textbf{or}\}$	$\{BOOLEAN\}$

6 Connection with LL-grammar

With Semantic actions in Top-to-Bottom parsing we also need to apply some Semantic rules as for example assignment of correct Semantic expression into variable. We have created new FSM, which will rely on detecting syntactic rules by LL-grammar. This FSM is therefore non-deterministic but non-terminals from FSM will notify us about special occasion as when we should evaluate our current FSM states. This in practice means whenever we go onto another statement we want to evaluate our FSM from its current state and values saved inside. After each new statement and follow up evaluation we go to default starting state of this FSM. We then set inside values to default and continue according to the FSM rules. We can see from START state we decide upon LL-grammar non-terminals and last state also usually gets us back to this starting state.

Note: After we evaluate statement, i.e. we move to START, we decide, whether the statement is semantically correct and if yes we then append this Node to Abstract Syntax Tree.





7 Code Generation

To generate code from AST we recursively descend from the root to leafs in post order. This way we have generated our children before we can use them. We can generate some parts of code before, between and after recursive descend to generate valid code. As header of file we generate *.IFJcode2021* and a JUMP $*function_calls*. As footer of the file we generate LABEL $*function_calls* and then all function calls in the source code. Between the JUMP and LABEL we generate all built-in-functions and defined functions from the source code.

7.1 DEFVAR in loops

To avoid generating this instruction inside loops we firstly generate all DEFVAR instructions before generating the statement Node with all instruction. Then we descend the Node second time but this time we generate all instructions except DEFVAR instructions.