Semantic Analyzer

xpojez00, xbudin05

November 28, 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 Semantic Tree. Tree is generated from expressions currently.

2 Structure

```
NodeType = {OPERATION, ID, FUNCTION, VALUE, VOID, NIL, POP}

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

struct Node {
    NodeType nodeType
    void *data
    vector *sons
    SemanticType semanticType
}
```

Note: POP type is used for function calls as a special Node operation. This way we know we should expect this Node from a returned function call.

3 Abstract Semantic 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.

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

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.

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

Considering all operands must have the same semantic value, i.e. for example

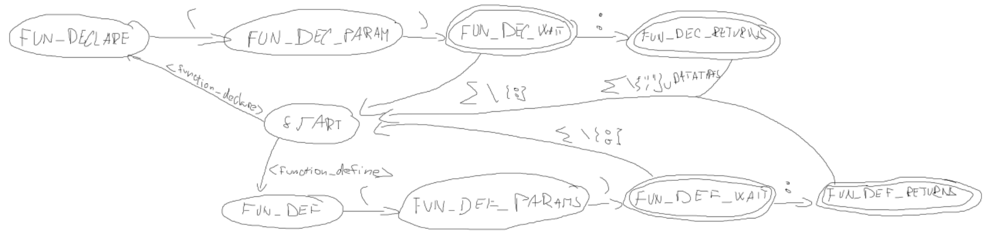
$$INTEGER < NUMBER$$

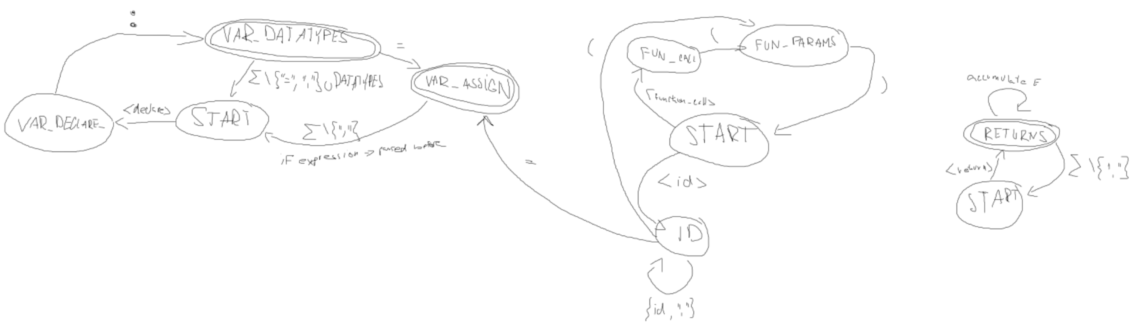
is not valid.

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

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 be 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.


FUN_DECLARE
(
FUN_DEC_PARAM
)
FUN_DEC_WAIT
:
FUN_DEC_RETURNS
<function_declare>
START
Σ \ {:}
Σ \ {","} ∪ DATATYPES
Σ \ {:}
<function_define>
FUN_DEF
(
FUN_DEF_PARAMS
)
FUN_DEF_WAIT
:
FUN_DEF_RETURNS


:
VAR_DATATYPES
=
Σ \ {"=", ","} ∪ DATATYPES
VAR_ASSIGN
VAR_DECLARE
<declare>
START
Σ \ {","}
if expression => parsed before
(
FUN_CALL
(
FUN_PARAMS
)
<function_call>
START
=
<id>
ID
{id, ","}
accumulate E
RETURNS
<return>
Σ \ {","}
START