COS341 Practical 4 : Academic Year 2023
TOPIC: *"Variable-has-Value" Analysis*

Motivation.
Assume we are given an SPL program which *contains* the following syntactically correct line:

⋮

n12 := a(**n**14,**n**17) ;

⋮

Though syntactically correct, this addition instruction "makes sense" *only if* the two numeric variables n14 and n17 *already have* some value at this point − otherwise no value could be transferred to the variable n12 by this assignment instruction.

Even though we *cannot* know at compile-time *what* value it is, we *can* find out by way of semantic analysis at compile-time *that* there is *some* value − or that there is none.

For this purpose we need a symbol table (which we already made in the previous Practical 3) and some suitable Semantic Evaluation Rules which get "attached" to the Grammar of **SPL** (which we know already from Practical 2).

Now, each named variable that is listed in the symbol table gets an additional binary flag −"has value", or "has no value"− which is *initialised to "has no value"* for all variables at the beginning of the semantic analysis. As the analysis proceeds (by crawling up and down a given SPL program's AST), the value-less variables' flags in the symbol table can possibly get "flipped" to "has value". As soon as a variable has received some value at some position in the program, this variable can never again lose this property of having a value.

ERROR Messages arise *when/where the analysis detects a value-less variable that ought to have a value* for the sake of a meaningful computation. For example: in the Assignment Instruction given above, an error message would arise if the semantic analysis would detect that variable **n**14 has no value at this specific "place" in the AST which corresponds to the given program line.

The Semantic Rules, which must be attached to the SPL Grammar for this purpose, are given below. Your task it is to implement the Analyser accordingly.

Semantic "Attribution" Rules to the Grammar's Syntax Rules.

1.
ALGO ::= **INSTR** COMMENT **SEQ**
SEQ ::= **;** ALGO

"For every named variable *v* that occurs under **ALGO** :
if *v* gets (or has) a value in **INSTR**,
then *v* has a value in **SEQ**"

*// **Comment:** Three special cases will be treated further down below, namely: when the instruction is a Procedure Call, when the instruction is a Loop, or when the instruction is a Boolean Branching*

2.

ASSIGN ::= NUMVAR ALGO ::= INSTR COMMENT SEQ

SEQ ::= ; ALGO := NUMEXPR

"For every named variable *v* that occurs under ASSIGN :
if *v* has a value in NUMEXPR,
then the variable of NUMVAR has (or gets) a value,
else Semantic Error"

3.

ASSIGN ::= BOOLVAR := BOOLEXPR

// ***Comment:*** *Very similar to Rule 2 of above:*
Please formulate this rule for yourself.

4.

ASSIGN ::= STRINGV := STRI

"**Fact**: STRI *is* a constant value.
Consequence: STRINGV has (or gets) a value here"

5.

NUMEXPR ::= **a (NUMEXPR , NUMEXPR)**

"For every named variable *v* that occurs *under* **NUMEXPR** :
if *v* has a value in **NUMEXPR**,
and if *v* has a value in **NUMEXPR**,
then *the variable bound to* **NUMEXPR** has (or gets) a value,
else Semantic Error"

6., 7.

NUMEXPR ::= **m (** NUMEXPR **,** NUMEXPR **)**
NUMEXPR ::= **d (** NUMEXPR **,** NUMEXPR **)**

// ***Comment:*** *Same as Rule 5 of above:*
Please formulate these two semantic rules for yourself.

8.

NUMEXPR ::= NUMVAR

"**if** the variable of NUMVAR has a value,
then the variable bound to NUMEXPR has (or gets) a value,
else Semantic Error"

9.

NUMEXPR ::= DECNUM

"**Fact**: DECNUM *is* a constant value.
Consequence: NUMEXPR has (or gets) a value here"

10., 11.

BOOLEXPR ::= LOGIC
BOOLEXPR ::= CMPR

// ***Comment:*** *Quite similar to Rule 8 of above:*
Please formulate these two semantic rules for yourself.

12.

LOGIC ::= BOOLVAR

// ***Comment:*** *very similar to Rule 8 of above:*
Please formulate this semantic rule for yourself.

Turn the page →

13.

LOGIC ::= **T** | **F**

"**Fact**: T,F *are* constant values.
Consequence: LOGIC has (or gets) a value here"

14.,...,19.

LOGIC ::= **^ (** BOOLEXPR **,** BOOLEXPR **)**
LOGIC ::= **v (** BOOLEXPR **,** BOOLEXPR **)**
LOGIC ::= **! (** BOOLEXPR **)**
CMPR ::= **E (** NUMEXPR **,** NUMEXPR **)**
CMPR ::= **< (** NUMEXPR **,** NUMEXPR **)**
CMPR ::= **> (** NUMEXPR **,** NUMEXPR **)**

*// **Comment:** Very similar to Rule 5 of above:*
Please formulate these six semantic rules for yourself.

20.

OUTPUT ::= TEXT | VALUE
VALUE ::= **o** NUMVAR
TEXT ::= **r** STRINGV

"For every named variable *v* that occurs in the sub-tree *under* OUTPUT :
if *v* does **not** have a value,
then Semantic Error"

21.

INPUT ::= **g** NUMVAR

"**Fact**: NUMVAR gets (or has) a value here"

22, **23**.

LOOP ::= **w(BOOLEXPR){**ALGO**}**
BRANCH ::= **i(BOOLEXPR)t{**ALGO**}**ELSE

"For every named variable *v* that occurs in the sub-tree *under* **BOOLEXPR** :
if *v* does **not** have a value,
then Semantic Error"

All the 23 Semantic Rules of above relied to some extent on **Rule 1** which implies that the semantic analysis will "flow" sequentially through the given SPL program (respectively its AST) from the 1st line of program code to its last line. However, because of the "*jumps*" that are implied by procedure calls or Boolean-conditioned instructions, we must also know more precisely what *actually* is the sequentially "next" instruction to which the "Has-Value" property of some variable gets transferred; otherwise our semantic value-flow-analysis would be flawed. For this purpose we additionally need the following higher-level "meta rules" under the guidance of which the 23 basic rules of above are applicable:

Turn the page →

Meta-Rule 24.
For any SPL *sub-AST* of the shape

ALGO ::= **INSTR** COMMENT |SEQ|
|SEQ| ::= **;** ALGO

in which **INSTR** has as its child node the SPL command **h** *// halt*

the *sub-AST* of node SEQ is **dead code** (because of the run-time meaning of "halt")
in which any further Semantic Analysis is irrelevant (obsolete)

Meta-Rule 25.
For any SPL *sub-AST* of the shape

ALGO ::= **INSTR** COMMENT |SEQ|
|SEQ| ::= **;** |ALGO|
|ALGO| ::= **INSTR** COMMENT |SEQ|
|SEQ| ::= **;** ALGO

in which **INSTR** has as its child node a procedure call command CALL ::= **c p**DIGITS,
for which there exists elsewhere in the AST a valid declaration **PROC** ::= **p** DIGITS **{ PROGR }**
(as analysed in the foregoing Practical 3),

the Semantic Analysis algorithm "jumps" from the ***End*** of **INSTR** to the ***Start*** of **PROGR**.

Meta-Rule 26.
For any SPL *sub-AST* of the shape

ALGO ::= **INSTR** COMMENT |SEQ|
|SEQ| ::= **;** |ALGO|
|ALGO| ::= **INSTR** COMMENT |SEQ|
|SEQ| ::= **;** ALGO

in which **INSTR** has as its child node a procedure call command CALL ::= **c p**DIGITS,
for which there exists elsewhere in the AST a valid declaration **PROC** ::= **p** DIGITS **{ PROGR }**
(as analysed in the foregoing Practical 3),

the Semantic Analysis algorithm "jumps" from the ***End*** of **PROGR** to the ***Start*** of **INSTR**.

Meta-Rule 27.
For any SPL *sub-AST* of the shape

ALGO ::= **INSTR** COMMENT |SEQ|
SEQ| ::= **;** |ALGO|
|ALGO| ::= **INSTR** COMMENT |SEQ|
|SEQ| ::= **;** |ALGO|
|ALGO| ::= **INSTR** COMMENT |SEQ|

in which **INSTR** has as its child node a LOOP ::= **w (** BOOLEXPR **) {** ALGO **}** node, we stipulate:
Any named variable *v*, which happens to have ***no*** value at the *End* of **INSTR** (green),
also has ***no*** value at the Start of **INSTR** (yellow)!

*// **Comment:** This is because we cannot know at compile-time whether the ALGO of LOOP will really be carried out at run-time (which depends on the truth or the falsity of BOOLEXPR).*

Meta-Rule 28.
For any SPL *sub-AST* of the shape

ALGO ::= **INSTR** COMMENT |SEQ|
SEQ| ::= **;** |ALGO|
|ALGO| ::= **INSTR** COMMENT |SEQ|
|SEQ| ::= **;** |ALGO|
|ALGO| ::= **INSTR** COMMENT |SEQ|

in which **INSTR** has as its child node a BRANCH ::= **i(**BOOLEXPR**)t{**ALGO**}**ELSE node,
in which the ELSE case is ***empty*** (ELSE ::= **ε**),
we stipulate, as above:
Any named variable *v*, which happens to have ***no*** value at the *End* of **INSTR** (green),
also has ***no*** value at the Start of **INSTR** (yellow)!

*// **Comment:** This is because we cannot know at compile-time whether the ALGO of BRANCH will really be carried out at run-time (which depends on the truth or the falsity of BOOLEXPR).*

Meta-Rule 29.
For any SPL *sub-AST* of the shape

ALGO ::= **INSTR** COMMENT |SEQ|
|SEQ| ::= **;** |ALGO|
|ALGO| ::= **INSTR** COMMENT |SEQ|

in which **INSTR** has as its child node a BRANCH ::= **i(**BOOLEXPR**)t{**ALGO**}**ELSE node,
in which the ELSE case is ***non-empty*** (i.e., ELSE ::= **e{**ALGO**}**),
we stipulate:
Any named variable *v*,
which does **not** have (or get) a value in the sub-AST under **t{**ALGO**}**
and which *also* does **not** have (or get) a value in the sub-AST under **e{**ALGO**}**,
also has ***no*** value at the Start of **INSTR** !

*// **Comment:** This is because we cannot know at compile-time whether the ALGO of "**then**" or the ALGO of "**else**" will actually be carried out at run-time (which depends on the truth or the falsity of BOOLEXPR). Only in case that a variable is (or gets) valued in **both** branches we can be **sure** that it is valued at the End of the entire If-then-else Command.*

Meta-Rule 30.
For any SPL *sub-AST* of the shape

ALGO ::= **INSTR** COMMENT |SEQ|
|SEQ| ::= **;** |ALGO|
|ALGO| ::= **INSTR** COMMENT |SEQ|

in which **INSTR** has as its child node a BRANCH ::= **i(**BOOLEXPR**)t{**ALGO**}**ELSE node,
in which the ELSE case is ***non-empty*** (i.e., ELSE ::= **e{**ALGO**}**),
we stipulate:
if the *sub-AST under* **t{**ALGO**}** "contains" the SPL halt command (**h**),
and if also the *sub-AST under* **e{**ALGO**}** "contains" the SPL halt command (**h**),
then the *sub-AST* of **INSTR** is **dead code** *// **Comment:** similar to Rule 24 above.*

Turn the page →

TO DO

For ***INPUT*** you will need an AST (such as in Practical 2),
as well as also an available Symbol Table (such as the one produced by Practical 3).

Your new AST Crawler Algorithm (of this Practical 4) augments the Information in the Symbol Table with “Has-value” or “Has-no-value” information for the variables *without deleting* from the Symbol Table the Procedure Scope Information produced in Practical 3. (In fact, you need this information to be able to “jump” from the call of a procedure to its *corresponding* body-code as mentioned above).

For ***OUTPUT*** you display the newly augmented Symbol Table, as well as ERROR messages in such cases in which the given Input was semantically invalid.

And now **HAPPY CODING** :)