Constraints liberate liberties constrain

Miguel Lemos

February 14, 2018

Abstract

As programmers, we tend to think of expressive power of a language or library as an unmitigated good.By making a prototype C that is functional I want to show the contrary; that restraint and precision are usually better than power and flexibility.

1 Formal Grammar

In formal language theory, a grammar (when the context is not given, often called a formal grammar for clarity) is a set of production rules for strings in a formal language. The rules describe how to form strings from the language's alphabet that are valid according to the language's syntax. A grammar does not describe the meaning of the strings or what can be done with them in whatever context, only their form. [1]

2 Lexicon

As in usual in formal grammars, we will suppose that we have some finite BaseExp of basic expressions. For our purposes, BaseExp can be taken to be a finite subset of ~~English words~~ C valid expressions. A *lexicon* is then simply taken to be a relation between basic expressions and pairs consisting of syntactic categories and semantic objects of the appropiate types. More formally, a *lexicon* is a relation:

(1) lex $\subseteq$ BasExp (Cat(BasCat)$\times$ D) subject to the semantic well-typing constraint that:

(2) if (e, ...α, f) $\in$ lex is a lexical entry then f is an element of the domain $D_{\tau(\alpha)}$. [2]

With *lexicon* defined, a *lexeme* is defined in computer science differently than *lexeme* in linguistics. A *lexeme* in computer science roughly corresponds to what might be termed a word in linguistics (the term word in computer science has a different meaning than word in linguistics), although in some cases it may be more similar to a morpheme.

A *lexeme* is a string of characters which forms a syntactic unit.[3]

Some authors term this a *token* , using 'token' interchangeably to represent (a) the string being tokenized, and (b) the token data structure resulting from putting this string through the tokenization process.[4][5]

A *token* or *lexical token* is a structure representing a lexeme that explicitly indicates its categorization for the purpose of parsing.[6] A category of tokens is what might be termed a part-of-speech in linguistics. Examples of token categories may include identifier and integer literal, although the set of token categories differ in different programming languages. The process of forming tokens from an input stream of characters is called tokenization. Consider this expression in the programming language C:

sum = 3 + 2;

Tokenized and represented by the following table:

Lexeme	Token Category
sum	Identifier
=	Assignment operator
3	Integer literal
+	Addition operator
2	Integer literal
;	End of statement

3 Using ANTLR for Lexical Analisis

Let lexical analysis be the extraction of individual words or lexemes from an input stream of symbols and passing corresponding tokens back to the parser: the use of ANTLR provided the tools to process code and develop a lexical grammar, for it to be used in lexical analysis.

Here are some of the regular expressions used, in the correct ANTLR sintax:

(3) LOGIC_OP : '&&'|'||'|'!';

(4) DATA_TYPE : 'int'|'float'|'char';

(5) IDENTIFICATOR: (LETTER|DIGIT)+;

Where some patterns can be appreciated: The Lexers must be written in all-caps, and can be anidated, like is the case with (5), which uses two lexers of lower precedence. On the topic of precendence: Precedence is used to avoid ambiguity:

(6) INT : (0..9)+;

(7) FLOAT : INT '.' INT';

(8) DIGIT : (INT|FLOAT)';

ANTLR would have to choose, when being faced with an integer, between INT and DIGIT. And it would do so by the order they are written. where the higher they are, the higher the precedence over the others. This way (6) would always win over (6). Which is the opposite of what would happen if they were written like this:

(7) FLOAT : INT '.' INT';

(8) DIGIT : (INT|FLOAT)';

(6) INT : (0..9)+;

With this in mind, (5) would have precedence over LETTER and DIGIT tokens. Each time a LETTER or a DIGIT was found, it would be treated as an IDENTIFICATOR. To avoid this, and preserve the precedence of the used tokens, a syntax rule should be used, not a lexer one. This was done in the case of the DIGIT rule.

(9) digit : INT|FLOAT;

where:

(10) for_loop : 'for' '(' ID (EQUAL INT)? ';' comparation ';' assignation ')' (code_block|stat);

would guarantee that no type other than (6) was used inside a for loop as index.:

In my compiler I supposed no floating point digits would be used inside of the for loop. I have been informed since that C allows for floats to be used inside for loops conditions, but the case remains: If it was the case that you wanted to specify a token of low precedence to be used, if you wanted not to let the index of a for loop to be DIGIT but only for it to be INT, then the figures (9) and (10) would provide the desired functionality.

With that said, it seems like the following figure would actually be more representative of a valid C syntax.

(11) for_loop : 'for' '(' ID (EQUAL digit)? ';' comparation ';' assignation ')' (code_block|stat);

The opposite situation of this is when a token has a null precedence, by design. And ANTLR observes that possibility, having the *fragment* keyword to use when a token is never going to be used other than as a part of composed ones.

Finally it must be observed that no lexer rule is recursive, because that it is not a property supported by regular expressions. This feature is left for the syntax rules, as follows.

4 Recursive syntax

The composition of lexer rules into high order functions allows for recursion, and with recursion is gained the necessary expressivity needed to represent all of the possible combinations in the language, be they computer or natural. A CFG is created, which is higher in the Chomsky hierarchy for relative expressiveness than regular expressions, nondeterministic finite automatons and regular grammars.[6]

5 Abstract Sintax Tree

An Abstract Sintax Tree can be implemented as a stack of *scopes*, starting from the global one. As an example, each start of a function is the start of a scope. But a local variable can also be created inside a conditioned block of code, where it cannot be accesed outside of it. At the end a generalization can be made where it is the braces that define the starts and the ends of scopes. Now what is a *scope*? A Scope is a dictionary of identificators, used to identify the variables. And what happens if a variable belongs to a higher scope, and is used on a lower scope, where it is not registered?

To satisfy this problem, scopes must have a jerarchical structure, where they store a reference to the scope directly higher to them, until the scope is global and has null for a reference. A *tree* is then formed, because many lower scopes can share a parent scope.

6 Running the AST

From running the implementation of the AST a problem was found: The variables declarated inside of the function parameters were not being taken into account during the tree walk, where a visitor object was supposed to push the scopes and their identificators. And the reason for it was discovered to be that in the created grammar, function parameters were not supposed to contain variable_declarations inside of them. Where as seen in the figure 12, the only difference among the two sintax rules are that parametersList allows for type only declarations, while a variable must have an identificator.

```
(12) parametersList : VALID_C_TYPES
                    | VALID_C_TYPES ID
                    | VALID_C_TYPES ID EQUAL to_value
                    | parametersList ',' parametersList
                    | ;

variable_declaration :
VALID_C_TYPES ( EQUAL to_value)? (','
VALID_C_TYPES? ID (EQUAL to_value)?)*);
```

With that being said, a fix came from merging the rules so that when a variable is identified inside of a function parameter list, it is parsed as a declaration of a variable.

```
(13) parametersList : VALID_C_TYPES
                    | variable_declaration
                    | ;
```

Now in order to support multiple comma separated VALID_C_TYPES the following rule could be implemented:

```
(14) parametersList :
VALID_C_TYPES (',' VALID_C_TYPES )*
                    | variable_declaration
                    | ;
```

But we still have the problem of the combination of the two series, the one of VALID_C_TYPES and the one for identifiable variables. To achieve that recursion must be implemented:

```
(15) parametersList :
VALID_C_TYPES (',' VALID_C_TYPES )*
                    | variable_declaration
                    | parametersList ',' parametersList
                    | ;
```

Now there is a problem, a parameterList could be a series of empty terminals, and because we are using recursion we are technically supporting

an infinite ammount of comma separated empty terminals as a function parameter list. To fix this, and all similar cases, we must avoid supporting recursivity on rules that support empty terminals.

A final version of the parameterList sintax rule would be: (16)

```
VALID_C_TYPES (',' VALID_C_TYPES )*
        | variable_declaration
        | parametersList ',' parametersList;
```

Where a function declaration, and prototype, can have either a parameterList, or an empty terminal, as valid parameters:

(17)

```
f_d:VALID_C_TYPES ID '(' parametersList? ')' code_block ;

f_p:VALID_C_TYPES ID '(' parametersList? ')' ;
```

7 Traversing the AST

As said, the AST can be thought of as a stack of scopes, where each contains the symbols that belong to them. In a hashset, preferably, for heuristic purposes. Now, a stack, by definition, contains a pop() function, that on call remove the last pushed element of the stack. This means that once the code is fully traversed, nothing of the AST data structure will remain stored. In order to preserve the information after compilation, a *symbol table* must be created, to be looked up to. But more of that later. What use can be made of the traversing of the AST itself?

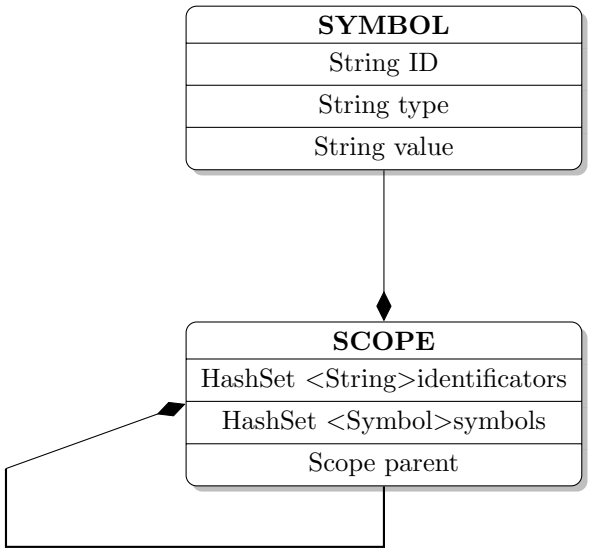
CErrorException: Undeclared variable x at line 8

By traversing the AST, higher scopes can be accesed from the lower, more recently added ones, so that variables that dont belong to the lower scopes and may belong to the higher scopes can be successfully identified as so, else a custom exception be handed to the user, stating the line and variable identificator.

8 Creating a Symbols Table

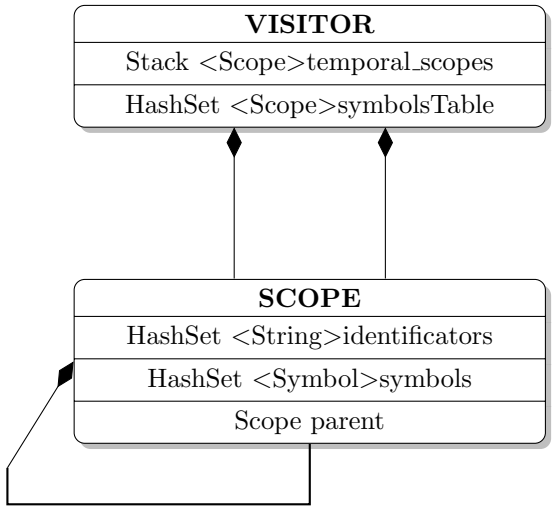
Having a stack of scopes at hand, and knowing its existence is short-lived, the creation of a persistent storage becomes important. The current implementation of the scopes stack only stored the variables identificators. Now it should store the type and the value. It could store them all together; it would be suboptimal in the case of identificator-only queries. And so, a hashset that stores Symbols instead of only Strings, is needed, parallel to the old one. This hashset will store all of the symbols inside the Scope.

(18)



Given that it is the Visitor object the one visiting the nodes and adquiring the information, it sounds reasonable for it to be owner of the two data structures composed of Scopes: *temporal_scopes*, the Stack, and *symbolsTable*, the detailed and persistent HashSet, as seen in the figure (19).

(19)



This, while heavier on memory usage, allows for query-specific optimization, by giving ID queries priority over complete symbol ones.

9 Valid symbol initialization

A first semantic analysis check could be as simple as checking that identifiers are used appropiately within the program: C is capable of handling character literal on integer intialization, among other weird possible conversions, but a warning would not be out of place if a user would initialize integers using chars.

Warning : Converting from int to char at line 1

Now there is a problem: ANTLR creates rules contexts like StringContext or any_sintax_ruleContext, for each of the sintax rules created in the grammar. It does not do so, however, for the lexer rules. And so the above warning is completely feasible using the current set of sintax rules, be them:

string : '"'(ID|digit)+ '"' | '\'' (ID| DIGIT)+ '\'';
digit: INT | FLOAT;

Because it can recognize that the type of a variable, in this case *int*, should not be initialized by a StringContext, which translated by a dictionary evaluates to *char*.

(20) { "StringContext" : "char" }

The problem begins when trying to check for floating point truncation. Then the dictionary has no use against the ambiguity of the digit sintax rule, suffering from key collision because the digit syntax rule can be either a float or an int.

(21) { "StringContext" : "char",
"DigitContext" : "float",
"DigitContext" : "int"
}

Transforming the lexers INT and FLOAT into the sintax rules int and float made (22) possible.

(22) { "StringContext" : "char",
"FloatContext" : "float",
"IntContext" : "int",
}

(23) var_declaration :
VALID_C_TYPES ID (EQUAL to_value)?
(' ,' VALID_C_TYPES? ID
(EQUAL to_value)?)*;
(24) to_value : ID|digit|string|f_c|math_operator;
(25) digit : int | float;

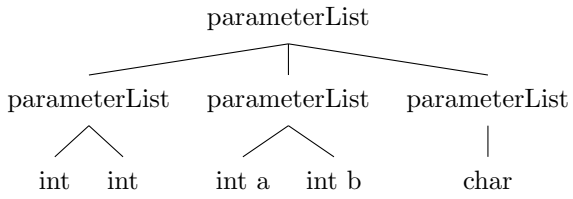
So that when asking variable_declaration(23) for the type VALID_C_TYPES evaluated to, this string, which can be 'int', 'float' or 'char' to name a few, can be successfully compared against the dictionary entry for the ruleContext to_value was evaluated to. Say it was the float syntax rule, then the ruleContext would be named FloatContext, and the entry for FloatContext would resolve into "float", which compared against "int" returns false, and a warning is released.

Warning : Converting from float to int at line 1

10 Flattening trees

Let f_p be a function prototype: (26)

f_p(int, int, int a, int b, char);



The problem starts when trying to get the list of parameters from the parameterList rule, where the data structure is not a list but a tree.

11 The imperative way

Thinking imperatively is a regular problem for developers when facing problems that can be solved with imperative solutions. Applying old algorithms to new problems seems like a reasonable solution. Here is how I adapted the BFS algorithm to my problem.

```
List<String> Parameters = new ArrayList();
Queue<CParser.ParametersListContext> parametersContexts = new ArrayDeque<>();
parametersContexts.add(ctx.parametersList());

while (!parametersContexts.isEmpty()){
  //parameters can contain parameters, so its important to get them all
  CParser.ParametersListContext currentParametersListContext = parametersContexts.poll();
  ParseTree parameterFirstChild = currentParametersListContext.children.get(0);
  try{
    Parameters.add(
    ((CParser.Variable_declarationContext)parameterFirstChild).VALID_C_TYPES().stream()
    .map( n -> n.getText())
    .collect( Collectors.joining( "," ))
    );
  }
  catch(java.lang.ClassCastException e){}

  try{
    Parameters.add(((org.antlr.v4.runtime.tree.TerminalNodeImpl)parameterFirstChild)
    .getParent().getText());
  }
  catch(java.lang.ClassCastException e){}

  currentParametersListContext.children.forEach(child -> {
    try{
      CParser.ParametersListContext nextParametersListContext =
        ((CParser.ParametersListContext) child);
      parametersContexts.add(nextParametersListContext);
    }
    catch(java.lang.ClassCastException e){}
  });
}

List<String> flattenedParametersTypes = Arrays.asList(Parameters.toString().split(","));
```

12 The functional way

Now, if there is a correct way of flattening a tree-based data structure, it is probably functional.

```
List<String> flattenedParametersTypes = ctx.parametersList().children.stream()

.flatMap(enterRules::flattenParameters)
.filter(string -> !string.isEmpty())
.collect(Collectors.toList());

flattenedParametersTypes =
 Arrays.asList(
   flattenedParametersTypes.toString().split(",")
 );
```

Now if those were all the lines needed to perform the task, the functional way would have proven itself to be incredibly expressive. And it is, its just that the total ammount of lines is bigger than that because of the auxiliar function used. Which goes as follows:

```
public static Stream<String> flattenParameters(ParseTree parameterListParseTree) {
  Stream<String> stringStream = null;
  try{
    //this filters children from ruleF_pContext like the parentheses
    CParser.ParametersListContext parameterList =
       ((CParser.ParametersListContext)parameterListParseTree);


    stringStream = Stream.concat(

    Stream.of(
      //At this point we can safely assume we are visiting every parameterList

      //but because ParameterLists contain ParameterList
      //we only want to work with the terminals, the ones
      //that don't contain ParameterLists

      ((!parameterList.parametersList().isEmpty())? "":
        //and now we safely cast between the two remaining options:

        //is it just a list of TOKENS?
        parameterList.children.get(0) instanceof
         org.antlr.v4.runtime.tree.TerminalNodeImpl ?
            parameterList.getText()
            :
        //or is it a variable declaration rule?
        parameterList.children.get(0) instanceof
         CParser.Variable_declarationContext ?
            ((CParser.Variable_declarationContext)parameterList.children.get(0))
            .VALID_C_TYPES().stream()
               .map( n -> n.getText()
               )
               .collect( Collectors.joining( "," ))
            : ""
        )

    ),
    parameterList.children.stream().flatMap((parameterListParameterList) ->
     enterRules.flattenParameters(parameterListParameterList))); // recursion here

  }catch(java.lang.ClassCastException e){}

  return stringStream;
}
```

The final result of both implementations is an array that contains the types of the parameters in the way they are meant to be visualized, as a list: [int, int, int, char, char]. (*BFS returns a reversed list).

13 Intermediate code gen

Again, if there is a correct way of translating an idiom to another, it is probably functional.
For mathematical expressions the following functions were initially proposed:

parseMath →parseSum

parseSum →parseProduct

parseProduct →parseFactor

parseFactor→parseMath

where the arrows means that a function on the left may call the one on the right for help.

14 The four cases problem

Given three monoids, be them SUM, PRODUCT, and FACTOR, all of them will operate with two operands on each side, and an operation at the center. These operands are mathematical operators.
Here is how I described them in my grammar:

(27)
math_operation : sum ;
sum : product unapplied_low_op * ;
product : factor unapplied_medium_op * ;
factor : math_operand unapplied_high_op *;
math_operand : digit | ID | f_c | grouped;

unapplied_low_op :
(MATH_OP_LOW_PRIORITY product);
unapplied_medium_op :
(MATH_OP_MEDIUM_PRIORITY factor);
unapplied_high_op :
(MATH_OP_HIGH_PRIORITY math_operand);

MATH_OP_LOW_PRIORITY : '+' | '-';
MATH_OP_MEDIUM_PRIORITY : '*' | '/';
MATH_OP_HIGH_PRIORITY : '^';

The only math operator that contains itself is the one called "grouped".

This means that if we take into account the cases where a math operator is recursive or not, for each operand, we are left with four cases:

CASE	LEFT-OPERAND	RIGHT-OPERAND
CASE 1:	NON-RECURSIVE	NON-RECURSIVE
CASE 2:	RECURSIVE	NON-RECURSIVE
CASE 3:	NON-RECURSIVE	RECURSIVE
CASE 4:	RECURSIVE	RECURSIVE

The non-recursive math operators can be translated with no effort. A transparent translation. For a monoid case 1, the translation would go as follows:

given y = 1 + 2

then

t0 := 1 + 2

where 0 is the index of the intermediate code line, the index of a TAD (Three Address Code), thus t0.

The recursive math operators can be translated with extra calculations. For a monoid case 2, the translation would go as follows:

given y = (1 + 3) + 2

then

t0 := 1 + 3

t1 := t0 + 2

Now because this problem will repeat itself for SUM, PRODUCT, and FACTOR, it makes little to no sense to write the solution for this problem on parseMath, parseSum, and parseProduct.
A single function is needed then, and a generic one.

15 A generic solution

This function would need to have as parameter the functions to be used for parsing.
When parseSum would call this function, it would pass as parameter parseProduct.
When parseProduct would call this function, it would pass as parameter parseFactor.

16 The carried ID problem

Say to have the following operation: 1 + 2
This would be translated into: t0 = 1 + 2

But this: (1 + 2) + (3+4)
Would get translated into this:
t0 := 3 + 4
t1 := 1 + 2
t2 := t0 + t1

The first case is a CASE 1, the second one is a CASE 4.

Up until then there are no problems with ID carrying. Now let's see what happens with the following operation:
(1 + 2 + 3) + (4 + 5 + 6)

t0 := 1 + 2
t1 := t0 + 3
t2 := 4 + 5
t3 := t2 + 6
t4 := t1 + t3

Here t4 uses not t2 in the first operand, but t1. And depending on the lenght of the right side calculation, the distance of the first operand can get further away.

We can infer this problem, the carried ID problem, is a problem we are always going to have with the left operand. That we can achieve by always calculating the left operand first, and then the right one. The relevant pseudo-code for the CASE 4 would be then:

leftOperand ←parsedOperand
leftOperandID ←leftOperand.getID
rightOperand ←parsedOperand
rightOperandID ←rightOperand.getID
TAD ←"leftOperandID op rightOperandID "
where TAD comes from THREE ADDRESS CODE. and op could be a plus sign.

Now, because we are dealing with THREE ADDRESS CODE translation, we can assume that at any point the worst case scenario would be to have one carried ID. If we were to translate to FOUR ADDRESS CODE we would have two carried IDs.

Which generalizes to the following formula:
N CARRIED ID for N+2 address code
where the 2 comes from the two adresses occupied by the operation and the right operand.

17 The function call problem

In some TAD implementations function calls have their own special operand, were aside from the GOTO usage other operands are used, such as PUSHPARAMETER and POPPARAMETER, where the next interpreter would read those operands and implement them.

The problem is that in order for a TAD implementation to make sense to the reader without the help of a higher interpreter, it needs to lack extra interpretable operands.

And so, I went for an added language restriction instead, that would negate make the need for added intepretation.

By using the substitution principle, functions calls can be replaced by their content. And so functions are actually values.

We can continue adding restrictions while we are at, and state that global state should be avoided at all cost, as functions would operate with the parameters they are given, and nothing else. No secundary effects. Only input and then, deterministic output.

This makes the language purely functional.

18 Function parameters

In order to pass the values to the function, each time a function is called the parameters need to be explicitly instanciated with the values of the passed parameters.
int sumPlus10(int x, int y){

  int z = 10; return x + y;
}
int a = 1, b = 2;
a = sumPlus10(a,b)

becomes:
a := 1
b:= 2
z:= 10
a := a + b + z

Where the body of the function is pasted above the assignation, and the return is pasted as the assignment.

19 Multiple returns

Given:

int min(int x, int y){

  if (x <y) return x; else return y;
}
int a = 1, b = 2;
a = min(a,b)

becomes:
a := 1
b:= 2
conditional0 := a <b
ifNot conditional goto L1
t0 := a
L1:
t0 := b
a := t0

20 Free Scala-like sintax

Now because of the usage of the substitution model used in languages like Scala, Lisp, and Haskell, the following three sintaxes are allowed:

```
int main(int z){
  double insideMain = 1;
  insideMain = aRandomMathOperation(z)
  return insideMain
}
```

```
int main(int z){
  double insideMain = 1;
  insideMain = aRandomMathOperation(z)
}
```

```
int main(int z){
  double insideMain = 1;
   aRandomMathOperation(z)
}
```

which given the following function call
z = main(z)

ends up, respectively, as
t0 := z + 1
insideMain := t0
z : t0

or
t0 := z + 1
insideMain := t0
z := t0

or
t0 := z + 1
z := t0

This is because functions and values become one and the same, where the compiler during a function call does a copy-paste of the function, replacing the names of the function variables with the ones given by the parameters in the function call. A function becomes a macro ready to be placed where it is called, and because no secundary effects are allowed, no extra considerations are needed. Nesting functions becomes easy, and by closure variables from a higher context are available to the functions below.

21 Nested calls

```
int main(int z){
  double insideMain = 1;
   addOne( addOne( addOne(z)));
}
```

results in
z := 0
insideMain := 1
t0 := z + 1
t1 := t0 + 1
t2 := t1 + 1
z := t6

and

```
int main(int z){
  double insideMain = 1;
   addThree( addTwo( addOne(z)));
}
```

in
z := 0
insideMain := 1
t0 := z + 1
t1 := t0 + 2
t2 := t1 + 3
z := t6

22 A note on the erasure of loops

For-loops and while loops should have been erased from the sintax in favor of approaches that favor recursion.

23 Constant propagation

Given a constant asignation, say x = 1, then when another assignation is performed it is faster to repeat the constant asignment, instead of asigning the value of the reference.

Going back to the three adress representation of the min function, it becomes apparent that this change would improve a lot the resulting code:

```
int min(int x, int y){

if (x < y) return x;
else return y;
}
int a = 1, b = 2;
a = min(a,b)
```

which became:
a := 1
b:= 2
conditional0 := a <b
ifNot conditional goto L1
t0 := a
L1:
t0 := b
a := t0

now becomes:
a := 1
b:= 2
conditional0 := a <b
ifNot conditional goto L1
t0 := 1
L1:
t0 := 2
a := t0

on the first pass.

On the second pass it now targets a := t0 as a viable candidate for optimization.

However, this should not be: Given freedom, the optimizer would overwrite the reference to t0 two times. One for the value 1, and then again for the value 2, leaving a:= 2 as a result.

This is wrong, as a should reference the result of the function.

The solution for this was the use of annotations. By leaving a note on top of a:=t0 that explains it should be leaved unoptimized, it is then ignored, and it becomes dependant of the function return.

References

[1] https://en.wikipedia.org/wiki/Formal_grammar

[2] page 173, "Formal Grammar: Theory and Implementation", by Robert Levine

[3] page 111, "Compilers Principles, Techniques, & Tools, 2nd Ed." (WorldCat) by Aho, Lam, Sethi and Ullman, as quoted in https://stackoverflow.com/questions/14954721/what-is-the-difference-between-token-and-lexeme

[4] Perl 5 Porters. "perlinterp: Perl 5 version 24.0 documentation". perldoc.perl.org - Official documentation for the Perl programming language. perldoc.perl.org. Retrieved 26 January 2017.

[5] Guy Coder (19 February 2013). "What is the difference between token and lexeme?". Stack Overflow. Stack Exchange Inc. Retrieved 26 January 2017.

[6] https://en.wikipedia.org/wiki/Lexical_analysis

[7] https://en.wikipedia.org/wiki/Expressive_power_(computer_science)