Issues/**#15**

Our goal is to generate a C code that is as close as possible to the three-address code.

Excursion - 3AC code (copied from course notes)
- at most 3 addresses at any time
- labelled as goto-target
- The following operations are allowed:

Assign

- x = y op z
- x= op z

Copy

- x=y
- x=y[i]
- x[i]=y

Address/Ptr

- x=&y
- x=*y
- *x=y

jumps

- go to L
- - if x go to L
- - ifFalse x go to L
- - if x relop y go to L

Procedure

- param
  X1x_1X1
- ...
- param
  XNx_nXN
- Call
  P1Np_1nP1N

As of 3AC, we do not use procedures (functions) and addresses/ pointers as well as instructions for copying arrays. To be able to use address/pointer and array copy statements, we would need to extend our language specification.

For functions, we simply use regular C functions.

Since we don't use array syntax, we also need to handle strings separately. I would suggest including the C library <string.h> and using strcpy() and strcat() to perform string assignments and additions.

Examples

Examplestrings

ezC	3ac-C
string x = „Hallo“;	char[] x = „Hallo“;
string y = "Welt!";	char[] y = "Welt!";
print(x+y);	char[] z = strcpy(x);
	strcat(z, y);
	printf("%s", z);

Wie man dorthin kommt

Not only do we need to generate code, but we also need to perform type checking and area management. We can also opt for continuous evaluation at this point.

The current concept is to have a set of functions (one for each production) that we run at pre-order (parent node before child nodes), and then another set of productions that we run after ordering.

Given the tree data type from our code, we can create the following function to traverse:

```
class Node():
    ...
    def run(self):
        self.preorder_hook()
        for child in self.children:
            child.run()
        self.postorder_hook()
```

In the preorder_hook we should process all inherited attributes and in the postorder_hook all synthesised attributes. I think this means that we use syntax-driven definitions, not translations, even though they are technically equivalent.

Type testing

The type test is carried out in the following non-terminals:

- ABTRETUNG
- EXPRESSION and all descendants of EXPRESSION

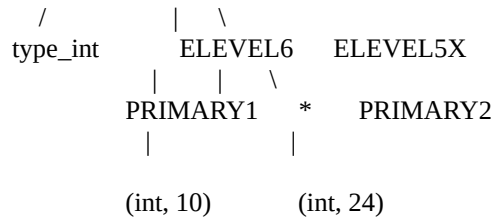
This means that if we have an operation with two operands, we must check whether both have the same type.

Example calculation

We have the following statement: int x = 10 * 24;

Simplified analysis tree:

```
Programm
          |
       ASSIGNMENT
     /  |    |  \
  TYPE  (ID,x)  =   EXPRESSION
```



The following things must be done for type checking: When descending the parse tree (aka inherited attributes, aka preorder hook):

- ASSIGNMENT can't actually do anything (regarding type checking), we have to "wait" until the types of TYPE and EXPRESSION are evaluated -> we can only handle that in the postorder_hook

- TYPE can read the type from the type specifier token, but could also do this while TYPE.t = "int" ascends.

- EXPRESSION can also do nothing

In the ascent we do:

- PRIMARY1.type= "int"
- ELEVEL6.type= PRIMARY.type="int"
- PRIMARY2.type= "int"
- ELEVEL5X.type= PRIMARY.type="int"
- EXPRESSION.value= if (calculable()) : 10* 24
- EXPRESSION.type= if (ELEVEL6.type == ELEVEL5X.type) then ASSIGNMENT.type= ELEVEL6.type sonst löst Exception() from
- ASSIGMENT.type = if (TYPE.type == ASSIGNMENT.type) then ASSIGNMENT.type= TYPE.type sonst löst Exception() from
- SymbolTable.write(ID=x, Typ= ASSIGNMENT.type, Wert= EXPRESSION.value)

Code generation

Code generation means that for each node we define the .code attribute as a string of C statements representing the subtree of which the node is the root.

Example 1

Same parse tree as above:Programm | AUFGABE / | |
TYP (ID,x) = AUSDRUCK / |
type_int ELEVEL6 ELEVEL5X | |
PRIMÄR1 * PRIMÄR2 | |

(int, 10) (int, 24)

When descending, we usually need to create all the necessary labels and set the .next attribute of each node and the labels attribute for each node. However, these things are only required in control flow statements, i.e. IFSTMT, WHILSTMT, RETURNSTMT, BLOCK.

In this example, the difficulty is to generate the code for the EXPRESSION subtree.

See image

auswerten, aus dem Buch Hinweis: Anstelle des .addr-Attributs verwende ich das .res-Attribut

Descending:

- EXPRESSION.res = new Temp()
- ELEVEL6.res= new Temp()
- ELEVEL5X.res= new Temp()

Ascending:

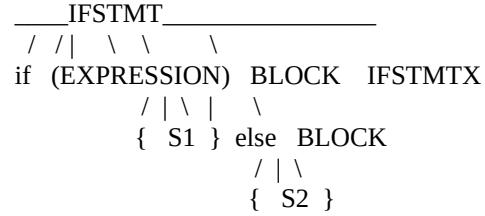
- ELEVEL6.code= f"ELEVEL6.type {ELEVEL6.res} = {PRIMARY1.value};"
- ELEVEL5X.code= f"ELEVEL5X.type {ELEVEL5X.res} = {PRIMARY2.value}; "
- EXPRESSION.code= f"{ELEVEL1.code} {ELEVEL2.code} {EXPRESSION.type} {EXPRESSION.res} = {ELEVEL6.res} * {ELEVEL5X.res}; "
- ASSIGNMENT.code= f" {ASSIGNMENT.type} {ID.name} = {EXPRESSION.res}; "

The above could of course be optimised, but for our purposes this should be sufficient.

Example 2

So why do we need functions on descent?

Take the following example:



This is where the .next file (also known as the labels) becomes interesting. If we assign them when descending, we can assume that IFSTMT.next is defined by its parent node. The .next attribute of both BLOCKs is now IFSTMT.next in any case.

But for the EXPRESSION node, we need to generate two new labels. We generate them on descent and pass them on to the EXPRESSION subtree. In the meantime, we memorise them in the IFSTMT.created_labels attribute and give them keys such as E_TRUE and E_FALSE. We need the label names in the lower EXPRESSION subtree to set them to the corresponding goto statements.

For example, let EXPRESSION evaluate to x==1.

The EXPRESSION code generation would then be something like this:

```
if (x==1) {goto EXPRESSION.true_label}
else {goto EXPRESSION.false_label}
```

It is therefore easier if we give the labels the keys now, as we can immediately include them in the code generated by the EXPRESSION subtree. The complete code for this example would/should read:

```
if (x==1) {goto E_True1}
else {goto E_False1}
```

```
E_True1:
   /* code for S1*/
   goto S_Next1
E_False1:
   /*code for S2*/
S_Next1:
```