Code Generation: An Example

A Program

- Consider the following program written according to the grammar given in the laboratory assignment-5. Its semantics is as usual.
- We shall generate intermediate 3-address code and GNU x86-64 assembly language target code for this program.

A Program

```
global
    def
        n, i, sum : int
    end
    print "Enter a positive integer: " ;
    read %d n;
    sum := 0;
    i := 0;
    while i <= n:
```

```
        sum := sum + i;
        i := i + 1
    end;
    print %d sum
end
```

Initialization of Data Structures

- The first construct that will be reduced is the declList of the program.

  prog → GLOBAL declList stmtListO END

- But it is necessary to perform actions like initialization of symbol table etc. before that.

Initialization of Data Structures

- We may put a new non-terminal between GLOBAL and declList.
- The grammar looks like

$$\text{prog} \rightarrow \text{GLOBAL m1 declList stmtListO END}$$

$$\text{m1} \rightarrow \varepsilon$$

- Actions for initialization are associated with the rule m1 $\rightarrow \varepsilon$.

In Bison

- Bison compiler allows mid-rule action. As an example between GLOBAL and declList in the previous case.
- The compiler introduces a new non-terminal like m1 producing ε.
- But there is a danger of transforming the grammar to non-LALR.

Variable Declaration

The right-most derivation of the variable declaration is as follows:

$$
\begin{aligned}
& \text{declList} \\
\Rightarrow_{rm} \quad & \text{decl declList} \\
\Rightarrow_{rm} \quad & \text{decl} \\
\Rightarrow_{rm} \quad & \text{DEF typeList END} \\
\Rightarrow_{rm} \quad & \text{DEF varList COLON type END} \\
\Rightarrow_{rm} \quad & \text{DEF varList COLON INT END}
\end{aligned}
$$

Variable Declaration

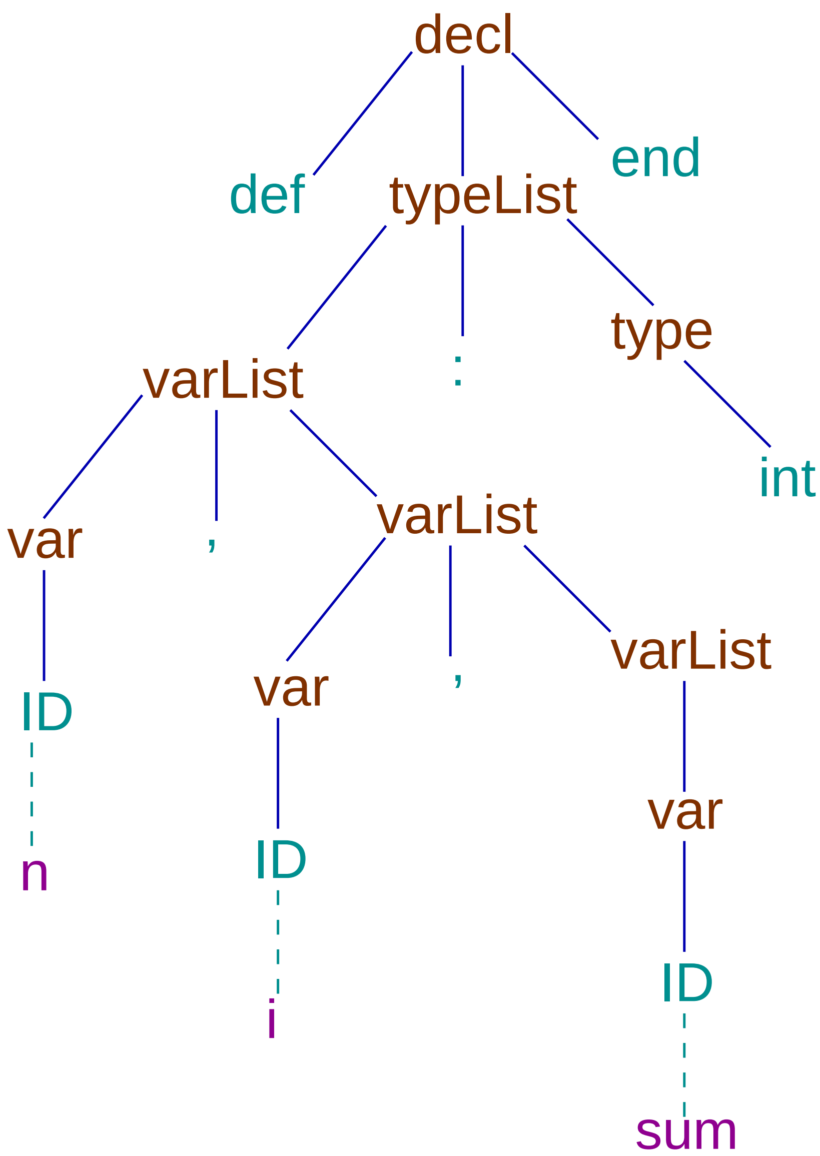
$\Rightarrow_{rm}$ DEF var COMMA varList COLON INT END

$\Rightarrow_{rm}$ DEF var COMMA var COMMA varList
COLON INT END

$\Rightarrow^{*}_{rm}$ DEF var COMMA var COMMA var
COLON INT END

$\Rightarrow^{*}_{rm}$ DEF ID COMMA ID COMMA ID
COLON INT END

Variable Declaration: Parse Tree


decl
def
typeList
end
varList
:
type
var
,
varList
int
ID
var
,
varList
n
ID
var
i
ID
sum

Attributes and Semantic Actions

What are the attributes of different non-terminals and what are the semantic actions during reduction?

Variable Declaration: Note

- Every time an ID is reduced to var, the corresponding lexme is inserted in the current symbol-table, and the symbol-table index is stored as an attribute of var[a].
- The non-terminal varList has a list of symbol-table indices corresponding to the vars underlying it.

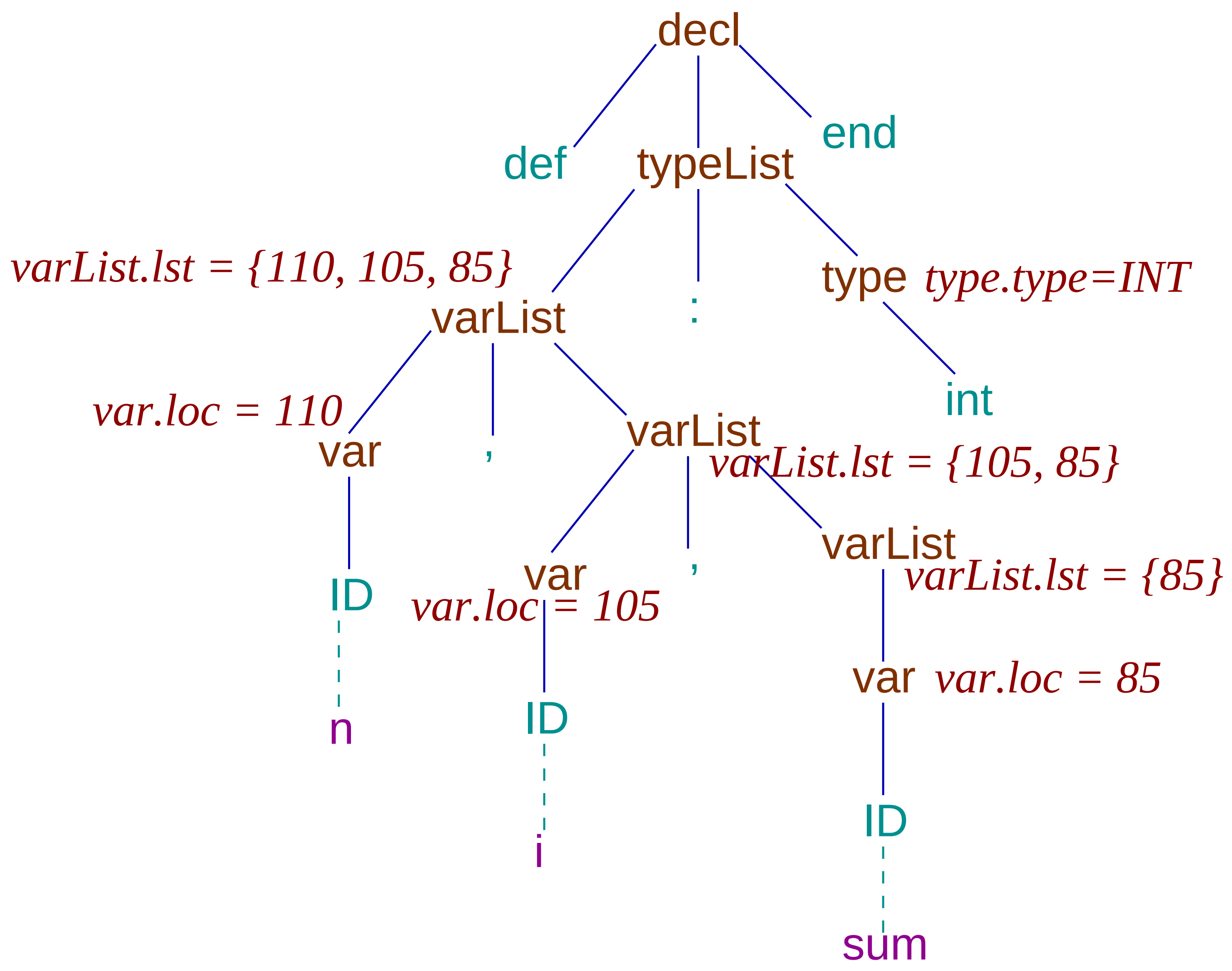
[a] There may be other attributes of var.

Variable Declaration: Note

- A reduction to typeList updates the symbol table entries with type and other information.
- The symbol-table looks like as follows:

index	lexme	type	offset
	...	...	...
85	`sum`	INT	-12
	...	...	...
105	`i`	INT	-8
	...	...	...
110	`n`	INT	-4
	...	...	...

Decorated Parse Tree


decl
def
typeList
end
varList.lst = {110, 105, 85}
varList
:
type
type.type=INT
var.loc = 110
var
,
varList
int
varList.lst = {105, 85}
var
,
varList
varList.lst = {85}
ID
var.loc = 105
var
var.loc = 85
n
ID
i
ID
sum

Rightmost Derivation: Statements

$$
\begin{aligned}
& \text{stmtListO} \\
\Rightarrow_{rm} \; & \text{stmtList} \\
\Rightarrow_{rm} \; & \text{stmtList SEMICOLON stmt} \\
\Rightarrow_{rm} \; & \text{stmtList SEMICOLON printStmt} \\
\Rightarrow_{rm} \; & \text{stmtList SEMICOLON PRINT FORMAT exp} \\
\Rightarrow_{rm} \; & \text{stmtList SEMICOLON PRINT FORMAT ID}
\end{aligned}
$$

Rightmost Derivation: Statements

$\Rightarrow^*_{rm}$ stmt SEMICOLON $\cdots$ PRINT FORMAT ID

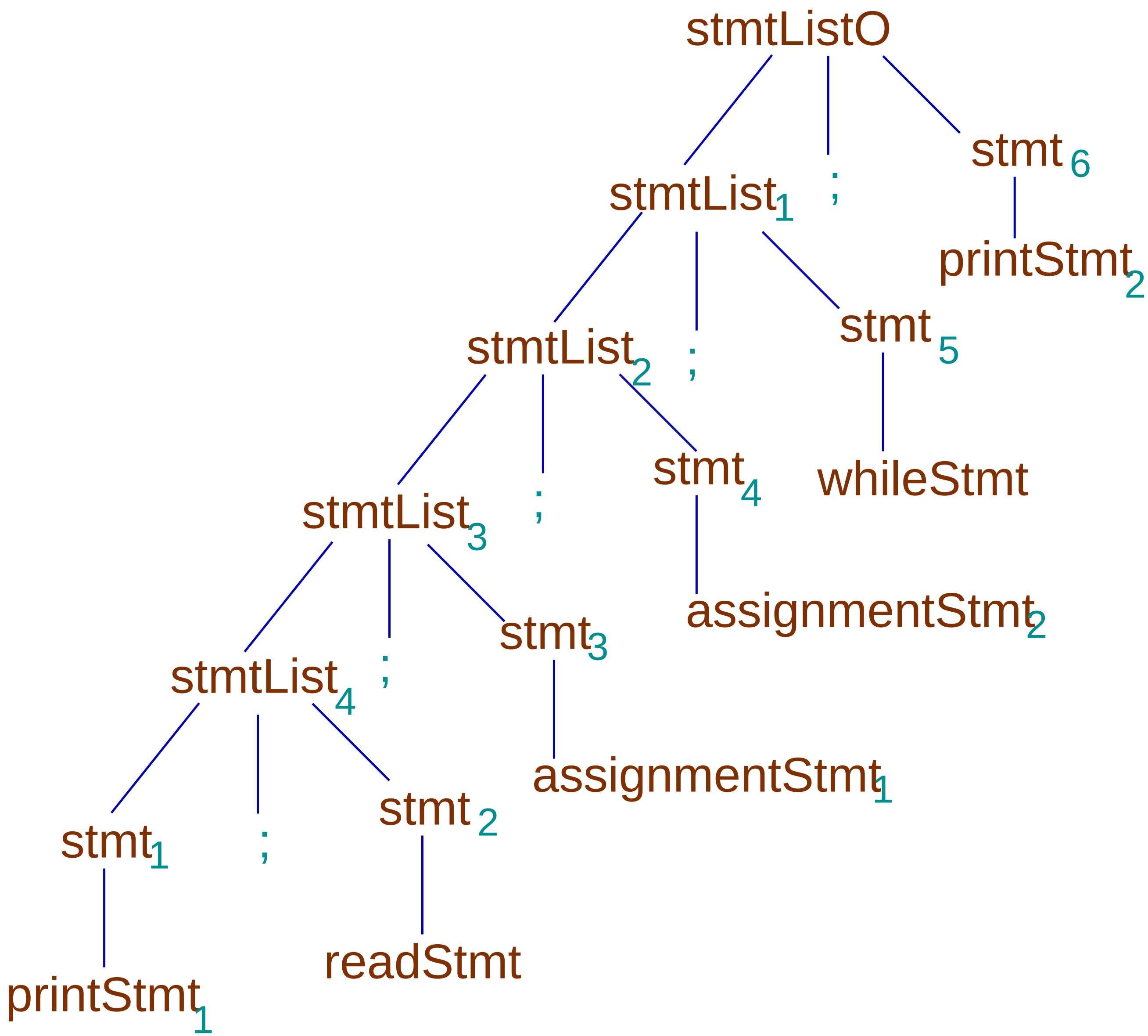
$\Rightarrow_{rm}$ printStmt SEMICOLON $\cdots$

PRINT FORMAT ID

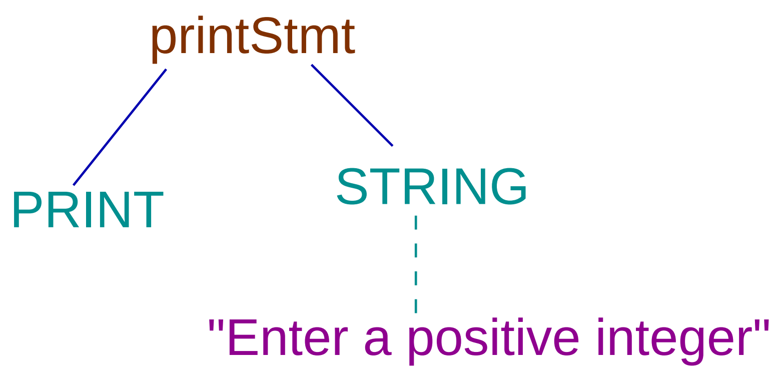
$\Rightarrow_{rm}$ PRINT STRING SEMICOLON $\cdots$

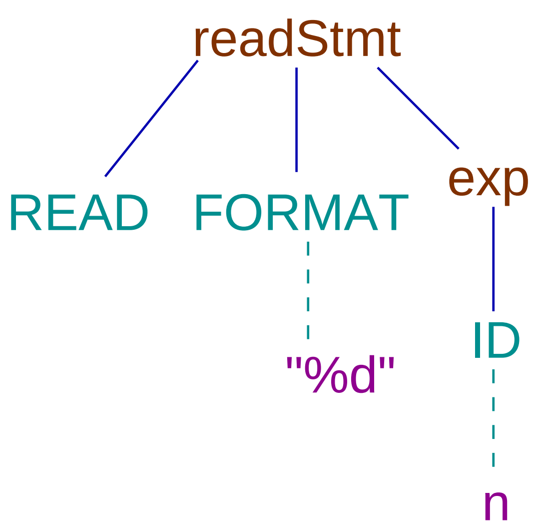
PRINT FORMAT ID

Statement List: Parse Tree


stmtListO
stmtList1
;
stmt 6
printStmt2
stmtList2
;
stmt 5
stmtList3
;
stmt4
whileStmt
stmtList4
;
stmt3
assignmentStmt2
stmt1
;
stmt 2
assignmentStmt1
printStmt1
readStmt

Print Statement_1 and Read Statement: Parse Trees





Note

Both printStmt$_1$ and readStmt has read-only data. We may store them either in the symbol-table or in a separate global data structure. We choose the second option.

Global Data

	Label	RO/RW	Type	Size	Data
0	`.LR00`	RO	STRING	27	"Enter a positive integer: "
1	`.LR01`	RO	STRING	3	"%d"
2	`.LR02`	RO	STRING	3	"%d"

3-address Code

- We have talked about 3-address codes.
- We assume that the sequence of 3-address codes are stored in a global array of structures.

Print Statement_1: 3-address code

- IO in most programming languages is done by library function call but we have hard-coded, it in our language.
- We use special 3-address codes for IO instructions. That will be finally translated to our library function calls (assignment 2) or C library function calls.

Print Statement$_1$: 3-address code

Command	Index of Global Data Table
printStr	0

Read Statement: 3-address code

An integer data is read in an integer variable.

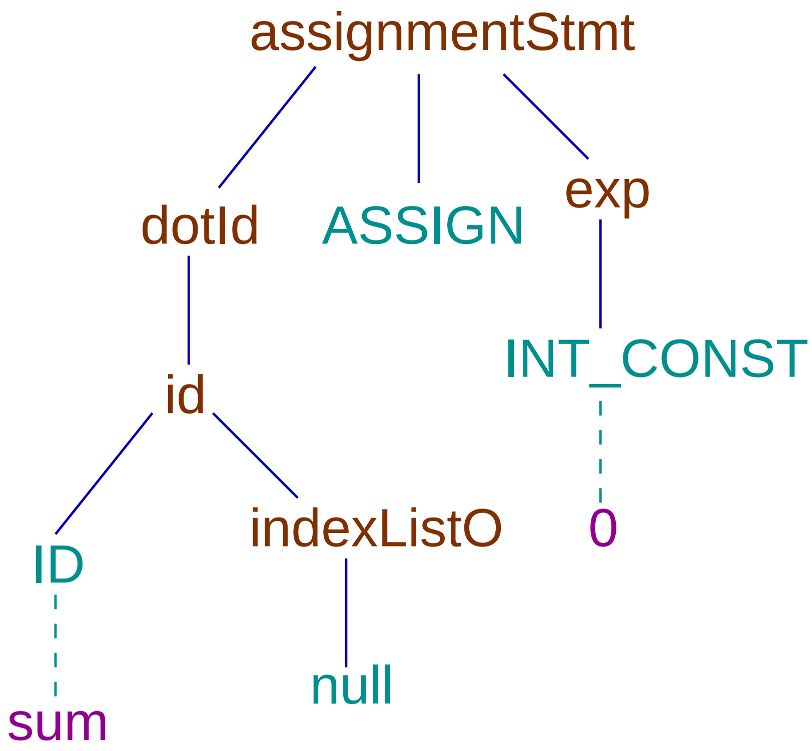
Command	Index of Symbol Table
readInt	110

Sequence of 3-address Codes

Index	Command	Other Fields
i	printStr	0
$i+1$	readInt	110

The index starts with i as there may be some preamble code before this.

Assignment Statement$_1$: Parse Tree



Assignment Statement$_1$: 3-address code

We consider simplest situation.

- The current symbol-table is searched with the lexme ”sum” of ID. If it is not found, it will be inserted in the symbol-table, but its type will be NOT_DECL etc[a]. as it is an error.

[a]We shall not talk about error-recovery etc. at this point.

- If it is found in the symbol-table, the index is preserved as a synthesized attribute of id.loc and also in dotId.loc.
- The situation will be more involved if id corresponds to an array element or a field of a structure.

Assignment Statement$_1$: 3-address code

- For the non-terminal exp, an internal variable name is created and entered in the symbol-table with appropriate type, displacement etc. Corresponding index is preserved as the synthesized attribute exp.loc.
- A 3-address code assigns the integer-constant to the new internal variable.

Assignment Statement$_1$: 3-address code

- Finally during the reduction to assignmentStmt, the internal variable is assigned to the program variable.
- But this is certainly not a good code and it is not difficult to remove the internal variable and assign the constant directly to the program variable.

Assignment Statement$_1$: 3-address code

Command	IntConst	Dst Index
assignIntConst	0	84 ($0)
assignVar	84 ($0)	85 (sum)

Modified to

Command	IntConst	Dst Index
assignIntConst	0	85 (sum)

Sequence of 3-address Codes

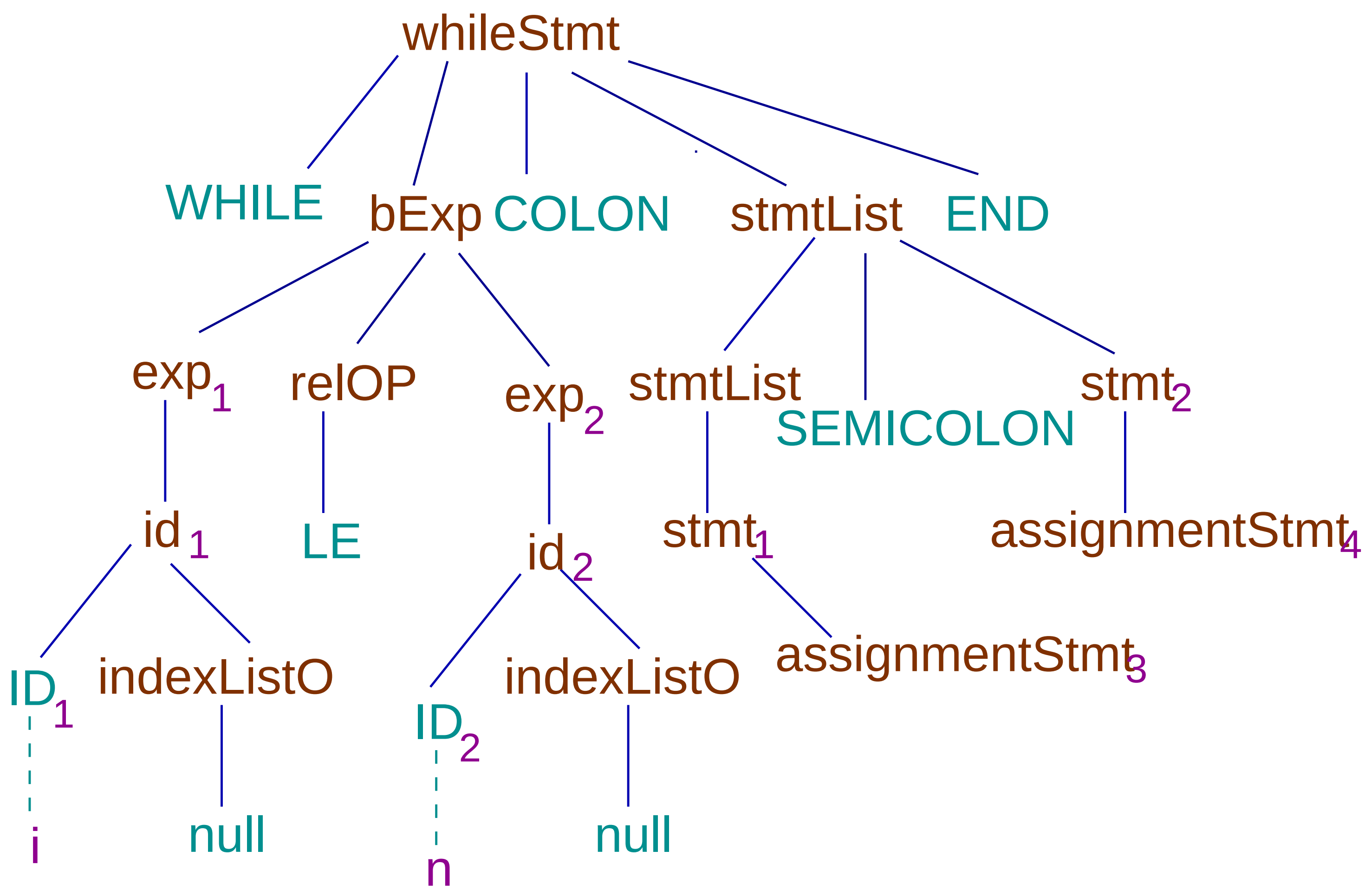
Index	Command	Other Fields	
i	printStr	0	
$i+1$	printInt	110	
$i+2$	assignIntConst	0	85 (sum)

Assignment Statement$_2$: 3-address code

The code of the second assignment statement is similar. The 3-address code sequence after the first four statements is,

Index	Command	Other Fields	
i	printStr	0	
$i+1$	printInt	110	
$i+2$	assignIntConst	0	85
$i+3$	assignIntConst	0	105

while-Statement: Parse Tree



Boolean Expression(bExp): 3-address code

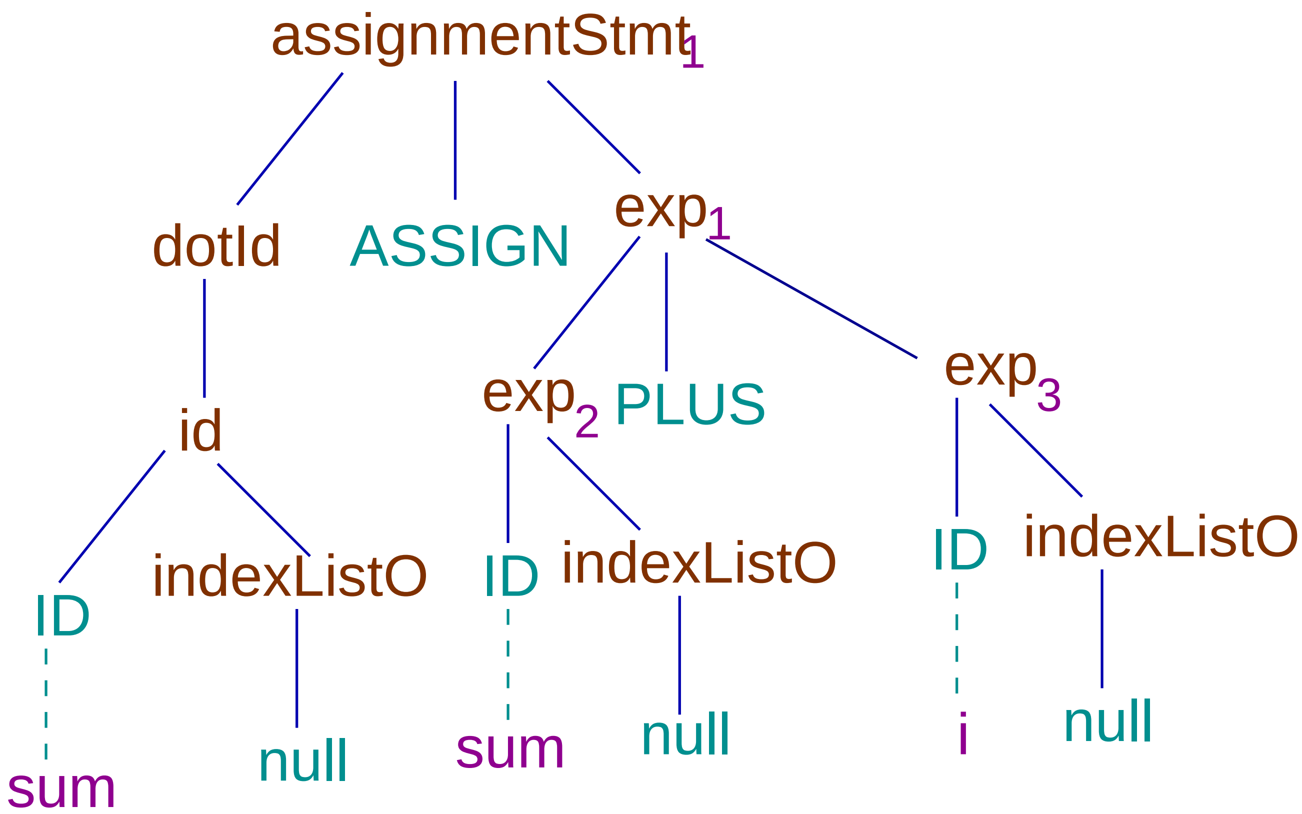
- $\texttt{exp}_1\texttt{.loc}$ has the index of i in the symbol-table. Similarly $\texttt{exp}_2\texttt{.loc}$ has the index of n in the symbol table.
- The 3-address code of the bExp is

Command	Src_1 Indx	Src_2 Indx	Jmp Loc.
ifLE	105 (i)	110 (n)	??
goto	??		

Note

- Two jump addresses in the 3-address codes of bExp are unknown at this point.
- We remember indices of these two 3-address instructions as synthetic attributes of bExp - bExp.trueList and bExp.falseList.
- "Address holes" in these 3-address instructions will subsequently be filled up.

Assignment Statement$_3$: Parse Tree



Assignment Statement$_3$: 3-address code

- The synthesized attributes of dotId.loc, exp_2.loc and exp_3.loc store the symbol-table indices corresponding to the program variables sum, sum and i respectively.
- The reduction of $\text{exp}_2 + \text{exp}_3$ to exp_1 creates an internal variable \$0, inserts it in the symbol table (index $(36 + 48) \bmod 128 = 84$) with appropriate type[a].

[a] Expressions may be of different types.

Assignment Statement_3: 3-address code

- The reduction produces the following 3-address code.

Command	Src_1 Indx	Src_2 Indx	Dst Indx
assignIntPlus	85 (sum)	105 (i)	84 ($0)

- More code may be needed if sum and i are of different types.

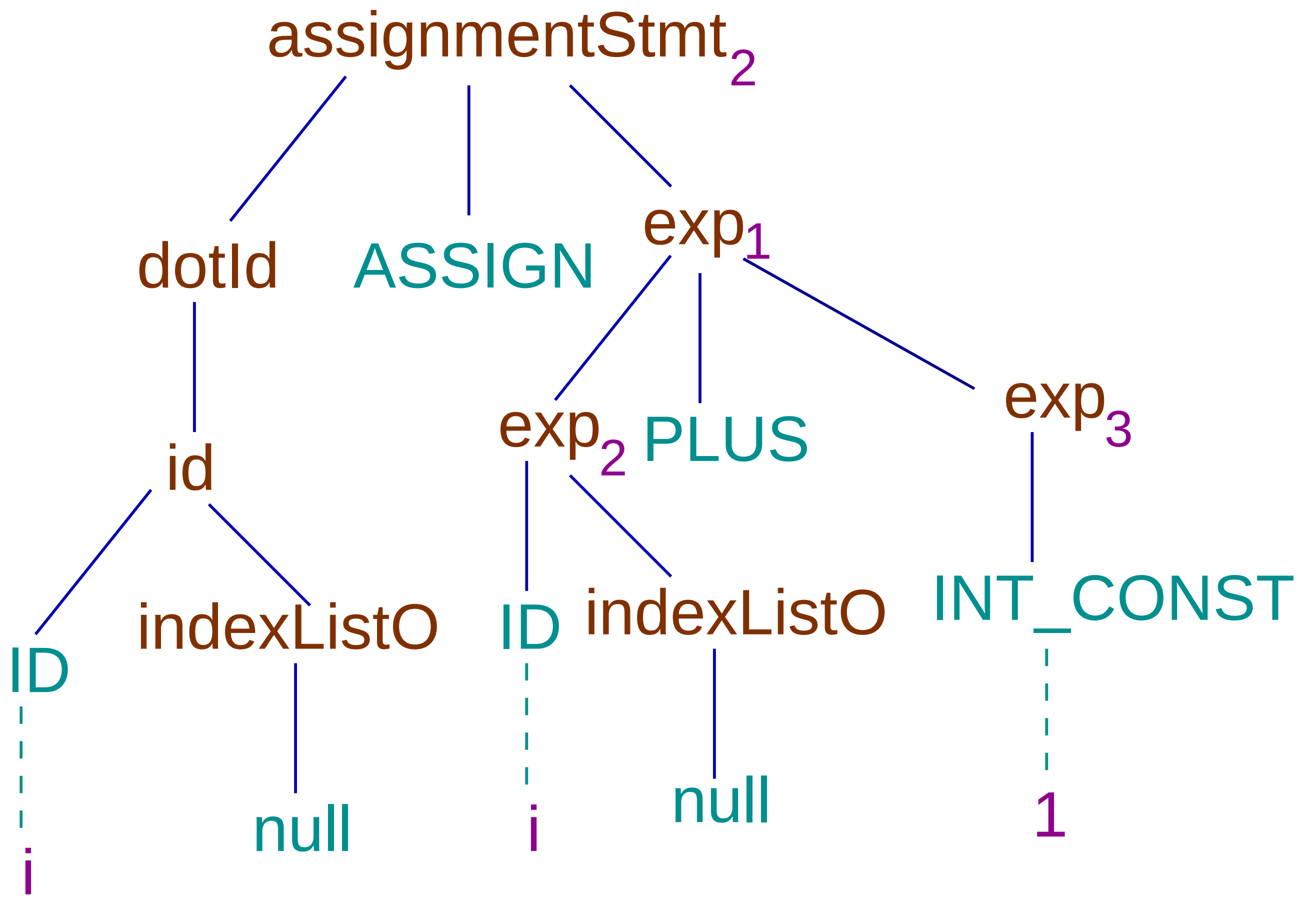
Assignment Statement$_3$: 3-address code

- Finally the reduction of doId = exp_1 to assignmentStmt produces the following three address code.

Command	Src Indx	Dst Indx
assign	84 ($0)	85 (sum)

- Again more code is needed if types of sum and $0 are different.

Assignment Statement$_4$: Parse Tree



Assignment Statement$_4$: 3-address code

- The 3-address code corresponding to `i = i + 1` is almost similar to `sum = sum + i`.
- The constant 1 may be stored in an internal variable.

Command	IntConst	Dst Indx
assignIntConst	1	85 ($1)

Assignment Statement_4: 3-address code

- Then i will be added to $1 and the value will be stored in another internal variable $2. But we may avoid the extra variable $1

Command	Src_1 Indx	IntConst	Dst Indx
assignIPC	105 (i)	1	86 ($1)

Assignment Statement$_4$: 3-address code

- Finally \$1 is assigned to i.

Command	Src Indx	Dst Indx
assign	86 (\$1)	105 (i)

Assignment Statement$_4$: 3-address code

Final code looks as follows:

Command	Other fields		
assignIntPlusConst	105 (i)	1	85 ($1)
assign	86 ($1)	105 (i)	

Note

- In the hash function computing symbol table index both sum and \$1 have same values - $(115 + 117 + 109) \bmod 128 = 85$ and $(36 + 49) \bmod 128 = 85$.
- So there is a collision in the symbol-table and that is to be properly handled. It is not enough to store 85 in the 3-address code. There will be no way to identify the actual name.

While Statement: 3-address code

- There are two blocks of 3-address codes corresponding to the while-statement. The question is how to stitch them.
- One important point to remember is that branch statement causes inefficiency in execution.

While Statement: 3-address code blocks

Boolean Expression			
Command	Field_1	Field_2	Field_3
ifLE	105 (i)	110 (n)	??
goto	??		

While Statement: 3-address code blocks

While Body			
Command	Field_1	Field_2	Field_3
assignIntPlus	85 (sum)	105 (i)	84 ($0)
assign	84 ($0)	85 (sum)	
assignIntPlusConst	105 (i)	1	85 ($1)
assign	86 ($1)	105 (i)	

Stitching - I

- In the program the code for boolean expression comes before the code of while-body. If we maintain this order, we need the following.
- A label at the beginning of the code for boolean expression - we call it $L0.
- A label at the beginning of the while-body - we call it $L1.

Stitching - I

The question is how to create these labels and fill the holes in the bExp code.

Stitching - I

- We modify the production rule of whileStmt as follows:

original: whileStmt $\rightarrow$ WHILE bExp COLON stmtList END

modified: whileStmt $\rightarrow$ WHILE m_1 bExp COLON m_2 stmtList END

$m \rightarrow \varepsilon$

Stitching - I

- Labels are generated during reduction of m_1 ($L1) and m_2 ($L2). They are stored as synthesized attributes of marker non-terminal m1.lbl and m2.lbl.
- Jump addresses of the 3-address codes corresponding to bExp.trueList is updated by m2.lbl.

Stitching - I

- A 3-address code ‘goto m1.lbl is generated at the end of the while-body.
- Jump addresses of the 3-address codes corresponding to bExp.falseList are to be updated by whileStmt.next.
- The code looks like -

While Statement: 3-address code blocks

Command	$Field_1$	$Field_2$	$Field_3$
Label	$L1		
ifLE	105 (i)	110 (n)	$L2
goto	??		
Label	$L2		
assignIntPlus	85 (sum)	105 (i)	84 ($0)
assign	84 ($0)	85 (sum)	

Command	Field_1	Field_2	Field_3
assignIntPlusConst	105 (i)	1	85 ($1)
assign	85 ($1)	105 (i)	
goto	$L1		

Stitching - I

- Two easy modifications can be made. Following code can be modified -

Command	Field_1	Field_2	Field_3
ifLE	105 (i)	110 (n)	$L2
goto	??		
Label	$L2		

Stitching - I

Command	Field_1	Field_2	Field_3
ifGT	105 (i)	110 (n)	??

- This makes the label $L2 redundant.
- We may introduce a label at the end and fill ?? with that.
- The new code sequence is -

While Statement: 3-address code

Command	Field_1	Field_2	Field_3
Label	$L1		
ifGT	105 (i)	110 (n)	$L2
assignIntPlus	85 (sum)	105 (i)	84 ($0)
assign	84 ($0)	85 (sum)	

Command	Field_1	Field_2	Field_3
assignIntPlusConst	105 (i)	1	85 ($1)
assign	85 ($1)	105 (i)	
goto	$L1		
Label	$L2		

Final Print Statement: 3-address code

Command	Field_1
printInt	85 (sum)

Program: 3-address code

Command	$Field_1$	$Field_2$	$Field_3$
printStr	0 (.LRO0)		
readInt	110 (n)		
assignIntConst	0	85 (sum)	
assignIntConst	0	105 (i)	
Label	$L1		
ifGT	105 (i)	110 (n)	$L2

assignIntPlus	85 (sum)	105 (i)	84 ($0)
assign	84 ($0)	85 (sum)	
assignIntPlusConst	105 (i)	1	85 ($1)
assign	85 ($1)	105 (i)	
Goto	$L1		
Label	$L2		
printInt	85 (sum)		

Symbol Table

index	lexme	type	offset
	. . .	. . .	. . .
84	$0	INT_T	−16
85	sum	INT	−12
85	$1	INT_T	−20
	. . .	. . .	. . .

index	lexme	type	offset
105	`i`	INT	-8
	...	...	...
110	`n`	INT	-4
	...	...	...

This gives us the size of memory space (may be on stack) required by the variables.

Stitching - II

- We may keep the boolean expression code below the code of the while-body. Boolean expression will start with a label $L2 (say).
- A synthesized attribute bExp.code may be used to preserve the boolean expression code.

Stitching - II

- The code corresponding to while-body starts with a label $L1.
- The execution of the loop starts with a jump to $L2, to test the boolean condition.
- Jump addresses of the 3-address codes corresponding to bExp.trueList are updated with $L1.

Program: 3-address code

Seq. No.	Command	Field_1	Field_2	Field_3
1	printStr	0 (.LRO0)		
2	readInt	110 (n)		
3	assignIntConst	0	85 (sum)	
4	assignIntConst	0	105 (i)	
5	goto	$L2		
6	Label	$L1		

7	assignIntPlus	85 (sum)	105 (i)	84 ($0)
8	assign	84 ($0)	85 (sum)	
9	assignIntPlusConst	105 (i)	1	85 ($1)
10	assign	85 ($1)	105 (i)	
11	Label	$L2		
12	ifLE	105 (`i`)	110 (`n`)	$L1
13	printInt	85 (sum)		

Generating Target Code

- Once the symbol-table, global data-table and sequence of 3-address codes are available, we are ready to generate target code.
- We generate equivalent assembly language code of x86-64 for the GNU assembler gas.
- For IO we may use standard C library or our own library (assignment 2).

Generating Target Code

- We need to allocate space (bind) for program variables and compiler generated variables.
- One simple solution is to keep all variables in the memory. But two important features prohibit that.

Generating Target Code

1. A memory access is much slower compared to CPU operations. So keeping operands in the memory will slow-down the process.
2. Many CPU operations require operands to be in the registers.

Register Allocation

- In any modern CPU, the number of general purpose registers may vary from a few to more than hundred.
- But the total number of variables in a 3-address code stream may be much larger.
- So it is necessary to decide which variables will stay in registers and for how long.

Register Allocation

- If it is necessary to bring some data from the memory to a CPU register, and no register is free, the content of some register is written back (spilling) to memory to make it available.
- So it is essential to keep track of the current binding of different variables and availability of registers.

Register Allocation

- Life span of a data, its assignment to a variable (definition), up to its last usage is an important information.
- But the computation of that requires more sophisticated analysis of the intermediate representation.

Register Allocation

We shall use the following ad hoc scheme.

- In the symbol-table we already have an offset field specifying the memory offset of a variable from the base of the activation record.
- We introduce one more field - reg. This field shows whether the most recent value of the variable is in memory or in a register. It also stores the name of the assigned register.

Register Allocation

- There is an accepted application binary interface (ABI) for the usage of registers.
- We shall use the following GCC convention for x86-64 architecture.

Register Usage Convention

GPR(64)	Usage Convention
rax	return value from a function
rbx	callee saved
rcx	4th argument to a function
rdx	3rd argument to a function
	return value from a function
rsi	2nd argument to a function
rdi	1st argument to a function
rbp	callee saved

64-bit GPR	Usage Convention
rsp	hardware stack pointer
r8	5th argument to a function
r9	6th argument to a function
r10	callee saved
r11	reserved for linker
r12	reserved for C
r13	callee saved
r14	callee saved
r15	callee saved

Function return address is at the top of the stack.

Modified Symbol Table

index	lexme	type	offset	reg/mem
	. . .	. . .	. . .	. . .
84	$0	INT_T	−16	eax
85	sum	INT	−12	
85	$1	INT_T	−20	
	. . .	. . .	. . .	. . .

index	lexme	type	offset	reg/mem
105	`i`	INT	-8	
	...	...	...	...
110	`n`	INT	-4	
	...	...	...	...

The requirement of stack space is 32B (multiple of 16B).

Global Data Table

	Label	RO/RW	Type	Size	Data
0	`.LR00`	RO	STRING	27	"Enter a positive integer: "
1	`.LR01`	RO	STRING	3	"%d"
2	`.LR02`	RO	STRING	3	"%d"

x86-64 Assembly Language Code Generation

We use information from global data table to generate the following code:

```
        .section .rodata
.LRO0:
        .string "Enter a positive integer: "
.LRO1:
        .string "%d"
.LRO2:
        .string "%d "
```

x86-64 Assembly Language Code Generation

Next part of the code is almost constant.

```
        .text
.globl  main
        .type   main, @function
main:
        pushq %rbp
        movq  %rsp, %rbp
```

x86-64 Assembly Language Code Generation

The total memory space requirement for all the variables (program defined and compiler generated) is available from the symbol table. We allocate this space in the stack frame. We could have done this in the common data area as well.

```
subq  $32, %rsp
```

Program: 3-address code

Seq. No.	Command	$Field_1$	$Field_2$	$Field_3$
1	printStr	0 (.LRO0)		
2	readInt	110 (n)		
3	assignIntConst	0	85 (sum)	
4	assignIntConst	0	105 (i)	
5	goto	$L2		
6	Label	$L1		

7	assignIntPlus	85 (sum)	105 (i)	84 ($0)
8	assign	84 ($0)	85 (sum)	
9	assignIntPlusConst	105 (i)	1	85 ($1)
10	assign	85 ($1)	105 (i)	
11	Label	$L2		
12	ifLE	105 (`i`)	110 (`n`)	$L1
13	printInt	85 (sum)		

x86-64 Assembly Language Code Generation

Code for

```
print "Enter a positive integer: " ;
printStr .LR00
```

is as follows:

```
movl  $.LR00, %eax
movq  %rax, %rdi
call  printf
```

It is like a parameterised template where starting address of the string is the parameter.

x86-64 Assembly Language Code Generation

Code for `read %d n;`

`readInt 110(n)`

is as follows:

```
movl $.LR01, %eax
movq %rax, %rdi
leaq -4(%rbp), %rsi
call  __isoc99_scanf
```

For a simple variable n (0ffset −4) the code

`leaq -4(%rbp), %rsi`

is also a template.

x86-64 Assembly Language Code Generation

Code for `sum := 0;`

`assignIntConst 0 85(sum)`

is as follows:

```
movl $0, -12(%rbp)
```

Here the constant and the offset of the variable are parameters.

x86-64 Assembly Language Code Generation

Code for `sum := sum + i;`

```
assignIntPlus 85(sum) 105(i) 84($0)
assign 84($0) 85(sum)
```

is as follows:

```
movl -8(%rbp), %eax
addl -12(%rbp), %eax
movl %eax, -16(%rbp)  # $0 <-- sum + i
movl %eax, -12(%rbp) # eax has the current
                        # value of $0
```

x86-64 Assembly Language Code Generation

Code for `i := i + 1;`

```
assignIntPlusConst 105(i) 1 85($1)
assign 85($1) 105(i)
```

is as follows:

```
movl -8(%rbp), %ecx
addl $1, %ecx  # $1 is available in ecx
movl %ecx, -8(%rbp)
```

Not only the offset of the variable, but also the available register is also a parameter.
This code can be improved as
`addl $1, -8(%rbp)`.
It is important as the instruction is within a

loop.

x86-64 Assembly Language Code Generation

Code for `if i <= n goto L1;`

`ifLE 105(i) 110(n) .L1`

is as follows:

```
movl -4(%rbp), %eax
cmpl %eax, -8(%rbp)
jle  .L1
```