Formal Languages and Compilers

ACSE: Building compilers with Bison and Flex

ACSE is an educational compiler that takes fictional language source code and converts it in a RISC-V assembly code

Daniele Cattaneo

Warning!

Prerequisites for this topic:

- Knowledge of assembly language
- Knowledge of the RISC-V architecture

Refer to supplemental material on WeBeep to catch up if necessary

Contents

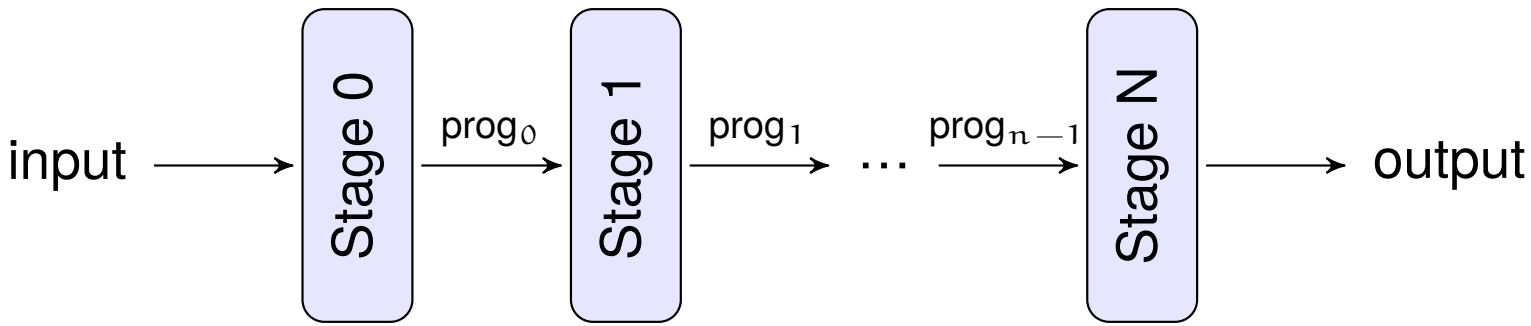
What does a compiler do?

The purpose of a compiler is:

- it translates a program written in a language L_0 into a **semantically equivalent** program expressed in language L_1.

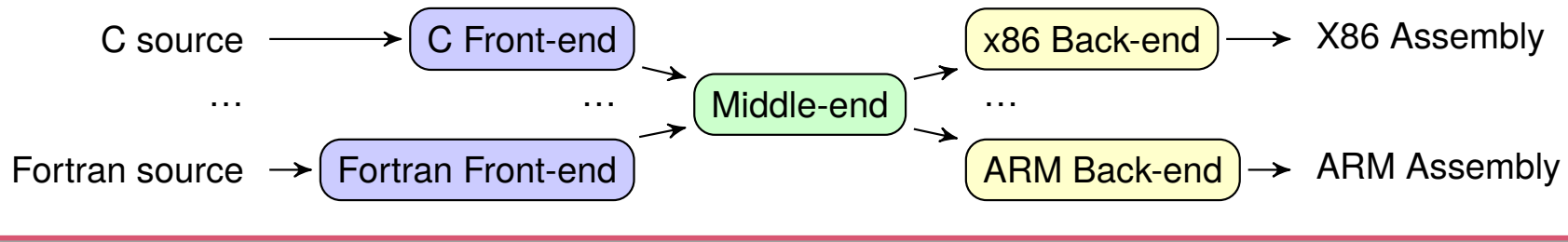
A compiler is organized as a pipeline:

- each stage applies a transformation to the input program producing an output program



Compiler pipeline

Generic compiler structure



Each stage has its own purpose:

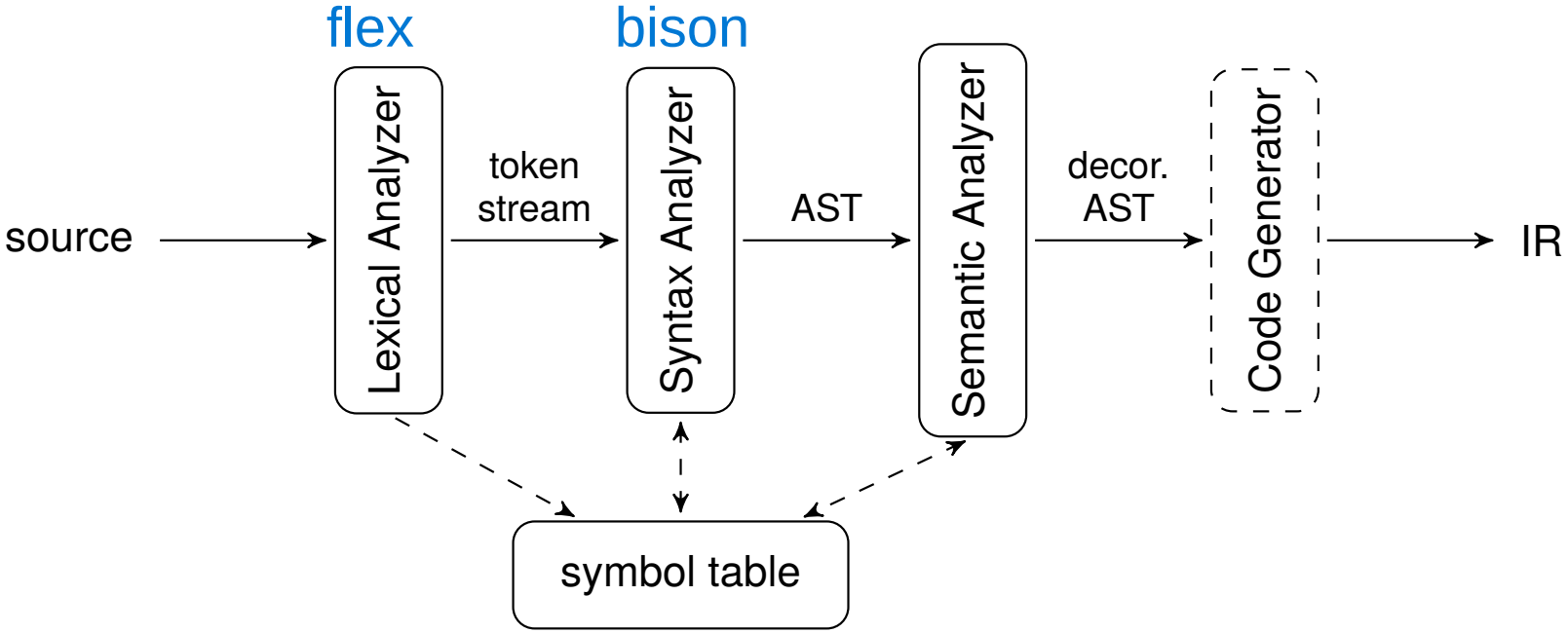
front-end converts from source language into an intermediate form (lexer and parser)

middle-end applies transformations and optimizations on the intermediate form

back-end convert from the intermediate form into target machine language

Front-end structure

The front-end translates a program from a source language to an intermediate form.



Main tasks:

- recognize language constructs
- verify syntactical/semantical correctness

A real world example: LLVM

one of the most common frameworks to build compilers

Many frontends:

- Various languages (C, C++, Objective C, Swift, Rust, Go)
- **clang** is the frontend for C-like languages
- Each frontend outputs a unified language called LLVM-IR

The intermediate representation is optimized in the **middle-end**

- Removes redundant or unused computations
- Rearranges loops to speed them up
- Tries to vectorize your code
- And more!

At last:

- Each intermediate instruction is mapped to real target CPU using a graph covering algorithm
- “Printers” emit assembly code or directly the binary

Contents

ACSE: Advanced Compiler System for Education

ACSE is a simple compiler:

- accepts a C-like source language (called LANCE)
- emits RISC-V (RV32IM) assembly language

It comes with two other helper tools forming an entire toolchain:

asrv32im Assembler (from assembly to machine code)

simrv32im RISC-V simulator

You can also use **RARS** instead of these two builtin tools

- RARS = Risc-v Assembler and Runtime Simulator
- Download from: `https://github.com/TheThirdOne/rars`

LANCE: LANguage for Compiler Education

LANCE is the source language recognized by ACSE:

- very small subset of C
- standard set of arithmetic/logic/comparison operators
- reduced set of control flow statements (`while`, `do-while`, `if`)
- only one scalar type (`int`)
- only one aggregate type (array of `int`s)
- no functions

Very limited support for I/O operations:

- **`read`**`(var)` reads an int from standard input and stores it into `var`
- **`write`**`(expr)` writes `expr` to standard output

A LANCE source file is composed by two sections:

- variable declarations
- program body as a list of statements

```
int x, y, z, i;
int arr[10];

read(x);
read(y);
z = 42;

i = 0;
while (i < 10) {
  arr[i] = (y - x) * z;
  i = i + 1;
}
z = arr[9];
write(z);
```

Compilation Process

How does ACSE compile a LANCE file to assembly?

Front-end:

1. The source code is **tokenized** by a **flex-generated scanner**
2. The stream of tokens is **parsed** by a **bison-generated parser**
3. The code is translated to a **temporary intermediate representation** by the **semantic actions in the parser**

Back-end:

4. The intermediate representation is **normalized** to **account for physical limitations of the RISC-V architecture**
5. Each instruction is **printed out** producing the **assembly file**

Same **overall structure** as **more complex** compilers, other details are simplified.

In fact in the intermediate representation it is assumed that machines have an infinite amount of registers and memory (no phisical limitation), this assumptions must be taken away when converting into machine code to apply to specific architectures

Contents

Root rules

A LANCE source file is split into two sections:

1. Variable declarations; root non-terminal: *var_declarations*
2. List of statements; root non-terminal: *statements*

The basic grammar rules (expressed in BNF):

$$
\begin{aligned}
\textbf{\textit{program}} &\rightarrow \textit{var\_declarations statements} \\
\textit{var\_declarations} &\rightarrow \textit{var\_declarations var\_declaration} \\
&\quad \mid \varepsilon \\
\textit{var\_declaration} &\rightarrow \ldots \\
\textit{statements} &\rightarrow \textit{statements statement} \\
&\quad \mid \textit{statement} \\
\textit{statement} &\rightarrow \ldots
\end{aligned}
$$

Grammar of declarations

Similar to C, but without inline initialization.

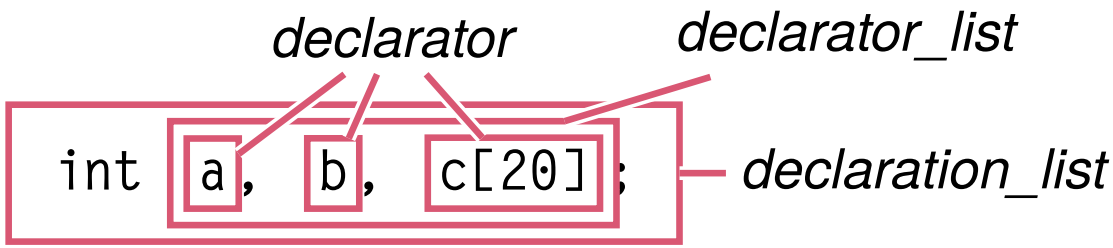
- IDENTIFIER is the non-terminal for a generic name (treated purely as a string)

is a TOKEN

TYPE is also a TOKEN
(only int type)

$$
\begin{aligned}
\textit{var\_declarations} &\rightarrow \textit{var\_declarations}\ \textit{var\_declaration} \\
&\mid \varepsilon \\
\textit{var\_declaration} &\rightarrow \text{TYPE}\ \textit{declarator\_list}\ \text{SEMI} \\
\textit{declarator\_list} &\rightarrow \textit{declarator\_list}\ \text{COMMA}\ \textit{declarator} \\
&\mid \textit{declarator} \\
\textit{declarator} &\rightarrow \text{IDENTIFIER} \\
&\mid \text{IDENTIFIER LSQUARE NUMBER RSQUARE}
\end{aligned}
$$

semicolumn



What is a statement?

> *A **statement** is a syntactic unit of an imperative programming language that expresses some action to be carried out.*
>
> — *Wikipedia*

Statements can be classified as:

Simple Indivisible element of computation

- Assignments, *read*, *write*, …

Compound Statements which contain multiple simple statements

- if, while, do-while

Simple statements

statement → *assign_statement* SEMI
| *if_statement*
| *while_statement*
| *do_while_statement* SEMI
| *return_statement* SEMI
| *read_statement* SEMI
| *write_statement* SEMI
| SEMI

A semicolon by itself is a valid statement!

assign_statement → *var_id* ASSIGN *exp*
| *var_id* LSQUARE *exp* RSQUARE ASSIGN *exp*

equal operator

return_statement → RETURN
read_statement → READ LPAR *var_id* RPAR
write_statement → WRITE LPAR *exp* RPAR
var_id → IDENTIFIER

Notice:

- ***exp*** is a **generic expression** (see next slide)
- ***var_id*** is a generic variable name (**symbol**)

Grammar of expressions

exp → NUMBER
| *var_id*
| *var_id* LSQUARE *exp* RSQUARE
| LPAR *exp* RPAR
| MINUS *exp*
| *exp* PLUS *exp*
| *exp* MINUS *exp*
| *exp* MUL_OP *exp*
| *exp* DIV_OP *exp*
| *exp* MOD_OP *exp*
| *exp* AND_OP *exp*
| *exp* XOR_OP *exp*
| *exp* OR_OP *exp*
| *exp* SHL_OP *exp*
| *exp* SHR_OP *exp*
| *exp* LT *exp*
| *exp* GT *exp*
| *exp* EQ *exp*
| *exp* NOTEQ *exp*
| *exp* LTEQ *exp*
| *exp* GTEQ *exp*
| NOT_OP *exp*
| *exp* ANDAND *exp*
| *exp* ORORR *exp*

PLUS	+
MINUS	-
MUL_OP	*
DIV_OP	/
MOD_OP	%
AND_OP	&
XOR_OP	^
OR_OP	|
SHL_OP	<<
SHR_OP	>>
LT	<
GT	>
EQ	==
NOTEQ	!=
LTEQ	<=
GTEQ	>=
NOT_OP	!
ANDAND	&&
ORORR	||
LPAR	(
RPAR	)
LSQUARE	[
RSQUARE	]

Operator precedences

Operator precedence and associativity:

```
%left OROR
%left ANDAND
%left OR_OP
%left XOR_OP
%left AND_OP
%left EQ NOTEQ
%left LT GT LTEQ GTEQ
%left SHL_OP SHR_OP
%left PLUS MINUS
%left MUL_OP DIV_OP MOD_OP
%right NOT_OP
```

precedence order

Same as C, **weirdnesses** included

- Operators & and | have **LOWER** priority than comparisons!

es in: 1 or b > 2

Bug in the expression grammar

The LANCE grammar supports **unary minus syntax** for negation:

$$\begin{aligned} \textit{exp} \rightarrow\ & \ldots \\ & \mid \text{MINUS } \textit{exp} \end{aligned}$$

But there's a problem:

- MINUS is **left associative** and has the **same priority** as PLUS
- This is correct for the normal **subtraction** operator
- But it is not correct for **negation**

Expression	Normal interpretation	LANCE interpretation
`- 1 * 2 - 3`	`((- 1) * 2) - 3`	`(- (1 * 2)) - 3`

At the exam, don't fall into the trap of forgetting how LANCE mis-interprets unary minus!

This bug can actually be fixed, look at the *bison* bonus slides to learn how.

Grammar of compound statements

code_block → LBRACE *statements* RBRACE

see simple statements slide

statement → *assign_statement* SEMI
| *if_statement*
| *while_statement*
| *do_while_statement* SEMI
| *return_statement* SEMI
| *read_statement* SEMI
| *write_statement* SEMI
| SEMI

if_statement → IF LPAR *exp* RPAR *code_block* *else_part*

else_part → ELSE *code_block*
| ε

while_statement → WHILE LPAR *exp* RPAR *code_block*

do_while_statement → DO *code_block* WHILE LPAR *exp* RPAR

- The ***code_block*** non-terminal is used in all situations where we can have **a list of statements enclosed by braces**.

Contents

Inside ACSE

The core elements of the ACSE compiler are:

scanner flex source in `scanner.l`

parser bison source in `parser.y`

codegen instruction generation functions: `codegen.h`

ACSE use bison to parse. The code (semantic action) is executed only at reduction phase, parsing left to right

ACSE is a **syntax directed translator**:

- Produces the compiled output directly **while parsing**
- The order of the compiled instructions depends on the syntax!

`parser.y` is the most important file in ACSE:

- Contains the (Bison-syntax) grammar of LANCE (what to be recognized)
- The semantic actions are responsible for the actual translation from LANCE to assembly (semantic actions contains the "translation" to assembly)

You can think of what happens **during compilation** as:

- Each node in the syntax tree is associated to some **C code**
- The tree gets visited bottom-up, left-to-right, and the code is executed
- When the C code is executed it replaces its node with:
  - Some **RISC-V assembly code** in a structured form
  - Optionally, a **semantic value** accessible to its parent
- The assembly code gets written in textual form to the compilation output file

The assembly code does not get executed at **compile time**!

- The C code only decides which instructions go where, it does not perform their effects yet
- The assembly runs only when the compiled program is launched (**runtime**)

ACSE Intermediate Representation

(intermediate representation)

The IR is the **data structure** used in a compiler to represent the program.

- This is where the **RISC-V assembly code is stored**

In ACSE it is composed of two main parts:

- The **instruction list**
- The **symbol table**

The **semantic actions** modify these two data structures to build the compiled program.

--> as some LANCE instructions are recognized and the relative semantic actions are executed, the instruction list and the symbol table are modified to store RISC-V assembly code (instructions and symbols)

The program instance

During parsing, the IR is stored in a **global structure** called **`program`**

- Declaration in `parser.y`, at the very top
- It also contains other contextual information

--> the global structure program, which contains the IR during parsing is declared at the top of the parser code

```
typedef struct {
  t_listNode *labels;
  t_listNode *instructions;  --> instruction list
  t_listNode *symbols;  --> symble table
  t_regID firstUnusedReg;
  unsigned int firstUnusedLblID;
  t_label *pendingLabel;
} t_program;

t_program program;
```

Almost every function in ACSE takes `program` as an argument.

ACSE IR instructions

Possible instructions that can be use?

The allowed instructions are a **superset** of RISC-V assembly

(which can be "stored" in the IR)

- Real IRs may be completely different than the target's assembly

Instructions regarding integers

ADD	SUB	MUL	DIV	REM		ADDI	SUBI	MULI	DIVI	REMI	
AND	OR	XOR	SLL	SRL	SRA	ANDI	ORI	XORI	SLLI	SRLI	SRAI
SEQ	SNE	SLT	SGE	SGT	SLE	SEQI	SNEI	SLTI	SGEI	SGTI	SLEI
		SLTU	SGEU	SGTU	SLEU			SLTIU	SGEIU	SGTIU	SLEIU
BEQ	BNE	BLT	BGE	BGT	BLE	LI	LW	Exit0		ReadInt	
J		BLTU	BGEU	BGTU	BLEU	LA	SW	PrintInt		PrintChar	

load immediate

load address

comparison istructions

Other additions:

- **infinite registers** (we are in middle end)
- no restrictions on immediates
- "syscall" instructions : Operations for I/O or system-level actions like Exit0, PrintInt, ReadInt, etc.

Register identifiers

A **register identifier** (`t_regID`) represents **a given register in the bank of infinite registers**

- Infinite registers are a concept common to almost every compiler
- Decouples ABI details from program semantics
- Compiler jargon: **temporaries** or **virtual registers**

ACSE denotes temporaries as **`temp`**$\langle n \rangle$ to distinguish them from **target machine registers** (`x0`, `x1`, `...` or `t0`, `ra`, `...`)

The value of the register identifier is the **number of the register**

- Register `temp10` has the register identifier `10`
- Register `temp`$\langle n \rangle$ has the register identifier n

You get a new temporary register by calling:

```
t_regID getNewRegister(t_program *program);
```

Machine register **`zero`** is **always** available trough the constant **`REG_0`**.

- This register is **hardwired** to always contain a zero
- Useful in various situations

Remember register at index 0 always contains 0

Adding instructions into a program

ACSE provides a set of **functions** that **add** one instruction **to the end** of the current program (they generate an assembly instruction and append it to the program data structure)

- Compiler jargon: **code generation** functions (hence the *gen* prefix)
- **Even real compilers work like this!** Especially JIT compilers
  - Real-world example: `https://asmjit.com`

```
// XXX = ADD SUB MUL DIV...
t_instruction *genXXX(t_program *program,
                      t_regID rd, t_regID rs1, t_regID rs2);
                        destination      source1        source2
// XXX = ADDI SUBI MULI DIVI...
t_instruction *genXXX(t_program *program,
                      t_regID rd, t_regID rs1, int immediate);

// Bcc label
t_instruction *genBcc(t_program *program,
                      t_regID rs1, t_regID rs2, t_label *label);
```

Register ids are integers, so they are passed by value.

Those functions will generate the assembly code which will use the registers ids; again those are only integers.

Accessing variables and arrays requires generating standard instruction sequences

- There are helper functions for that!

typically used to load from the memory

load variable from memory

```
t_regID genLoadVariable(t_program *program,
                        t_symbol *var);
```

all information of variable (contained in t_symbol structure): array or not, identifier, label (location in memory)

store register to a variable

```
void genStoreRegisterToVariable(t_program *program,
                                t_symbol *var, t_regID reg);
```

store constant to a variable

```
void genStoreConstantToVariable(t_program *program,
                                t_symbol *var, int val);

t_regID genLoadArrayElement(t_program *program,
                            t_symbol *array, t_regID rIdx);

void genStoreRegisterToArrayElement(t_program *program,
                t_symbol *array, t_regID rIdx, t_regID rVal);
void genStoreConstantToArrayElement(t_program *program,
                    t_symbol *array, t_regID rIdx, int val);
```

depends on the content that at runtime is inside that register

Semantic values

ACSE uses these semantic values to keep track of information across semantic actions:

Symbol	Type	Purpose
`NUMBER`	`int`	Numeric value of the token
`IDENTIFIER`	`char*`	String value of the token
`var_id`	`t_symbol*`	Object referring to the variable
`exp`	`t_regID`	Register containing the expression value

Contents

Variable declaration

```
declarator --> $$ (value of the left hand side of the rule)
  : IDENTIFIER --> $1  (it's the first value of the right hand side of the rule)
  {
    createSymbol(program, $1, TYPE_INT, 0);
  }
  | IDENTIFIER LSQUARE NUMBER RSQUARE
  {
    createSymbol(program, $1, TYPE_INT_ARRAY, $3);
  }
;
```

if 0: scalar

cannot call createSymbol with TYPE_INT but with a size !=0

$1

$3

enumerator

if not zero: size of the array

All the work is done in the *createSymbol()* function

- If the symbol already exists, the compiler exits (from within createSymbol())
- Otherwise, the symbol is added to the table
- *createSymbol()* takes ownership of the string

createSymbol will create a t_symbol* object and populate a structure that contains all the t_symbol*

contains: name of variabe, type, boolean (array or not), size

x = y + 3, parsed like:
exp PLUS exp exp
var_id num (contains 3)
now we want to parse var id

(see slide 18)

Symbol lookup

```
var_id
  : IDENTIFIER
  {
    t_symbol *var = getSymbol(program, $1);
    if (var == NULL) {
      yyerror("variable not declared");
      YYERROR;
    }
    $$ = var;
    free($1);
  }
;
```

--> abort the compilation

--> assigning var_id to the var we read

--> because IDENTIFIER is generated by flex and the rule that generate the string uses strdup(), which duplicate a string (allocating space for in another memory address with a malloc), so it needs to be freed.

The semantic value of *var_id* is set to the correct symbol table item

- If the symbol does not exist, compilation stops
- The string allocated in the lexer is freed in the semantic action

Contents

Semantic actions of assignments

```
assign_statement
  : var_id ASSIGN exp
  {
    genStoreRegisterToVariable(program, $1, $3);
  }
  | var_id LSQUARE exp RSQUARE ASSIGN exp
  {
    genStoreRegisterToArrayElement(program, $1, $3, $6);
  }
;
```

store the register inside the variable (help function seen before)

a[3+y] =4+x is possible since we have exp inside the brackets

Again all the work is done in the auxiliary functions

- exp generates the code for computing the expressions
- The actions here just generate the assignment itself (LA+SW)

Semantic actions of operators

```
exp
  : NUMBER
  {
    $$ = getNewRegister(program);
    genLI(program, $$, $1);
  }
  | var_id
  {
    $$ = genLoadVariable(program, $1);
  }
  | exp PLUS exp
  {
    $$ = getNewRegister(program);
    genADD(program, $$, $1, $3);
  }
  |
  // ...
;
```

--> gen Load immediate. takes a destination register and an immediate (integer). So it populate the runtime value of that register with the integer.

$1 is of type be t_symbol

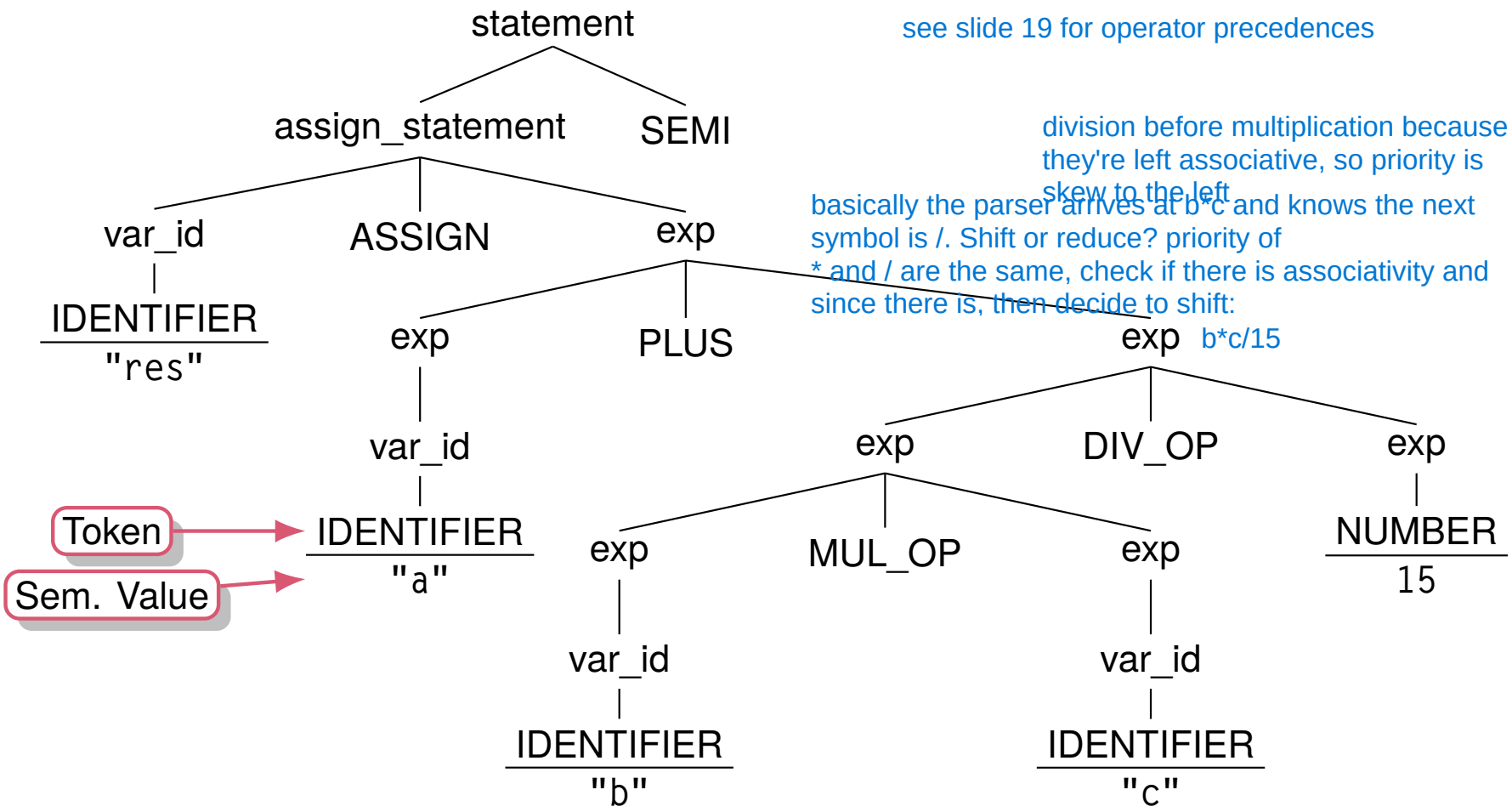
- `exp: NUMBER` generates a load of a constant in a new register
- `exp: var_id` generates a load of the variable's value in a new register
- The addition action just generates the addition between registers
- Other operators are similar

Remember: code executed when a rule is completely parsed

Example of how it all works

```
res = a + b * c / 15;
```

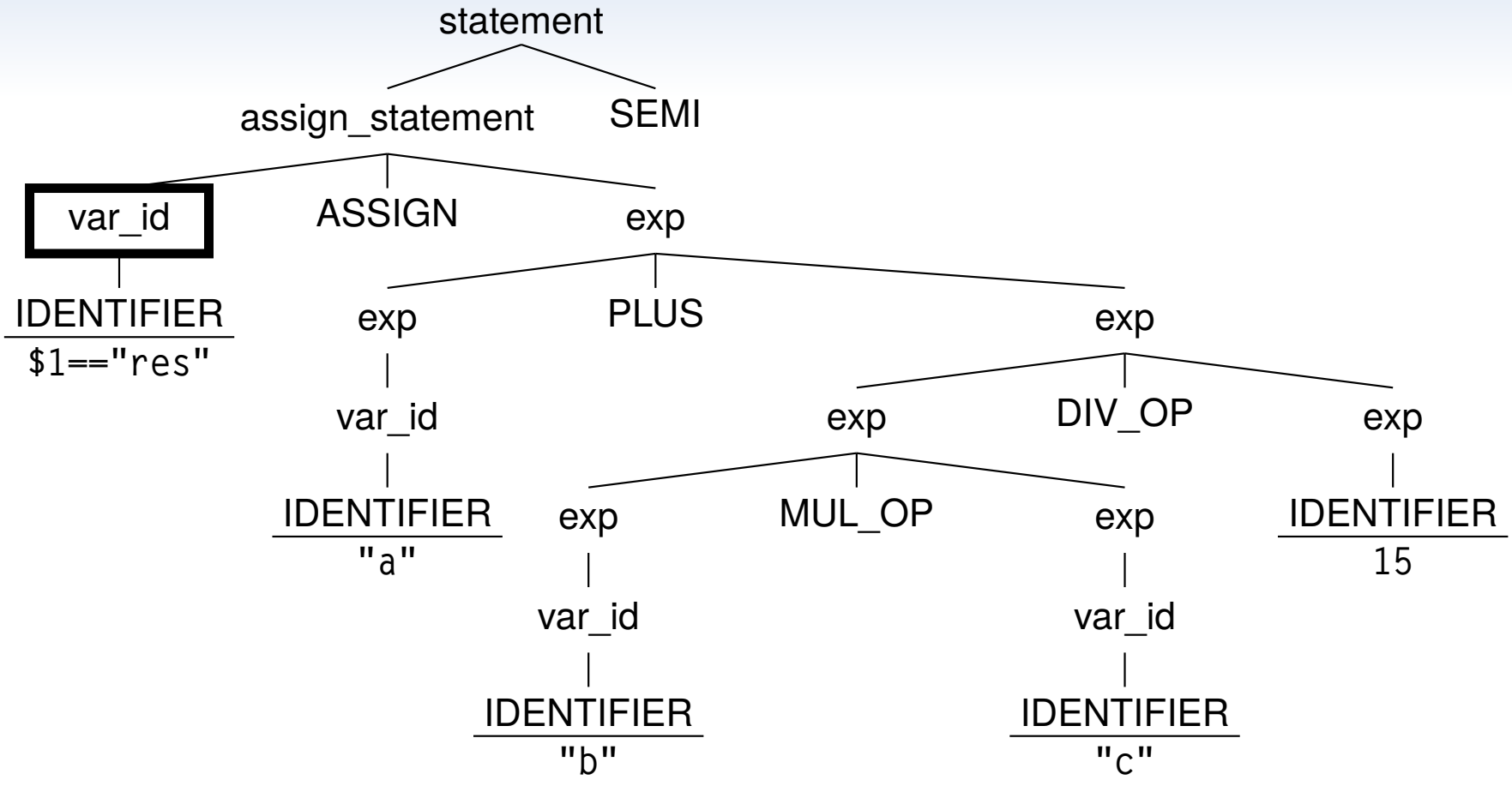
The parser builds its syntax tree; sem. values come from the lexer:



see slide 19 for operator precedences

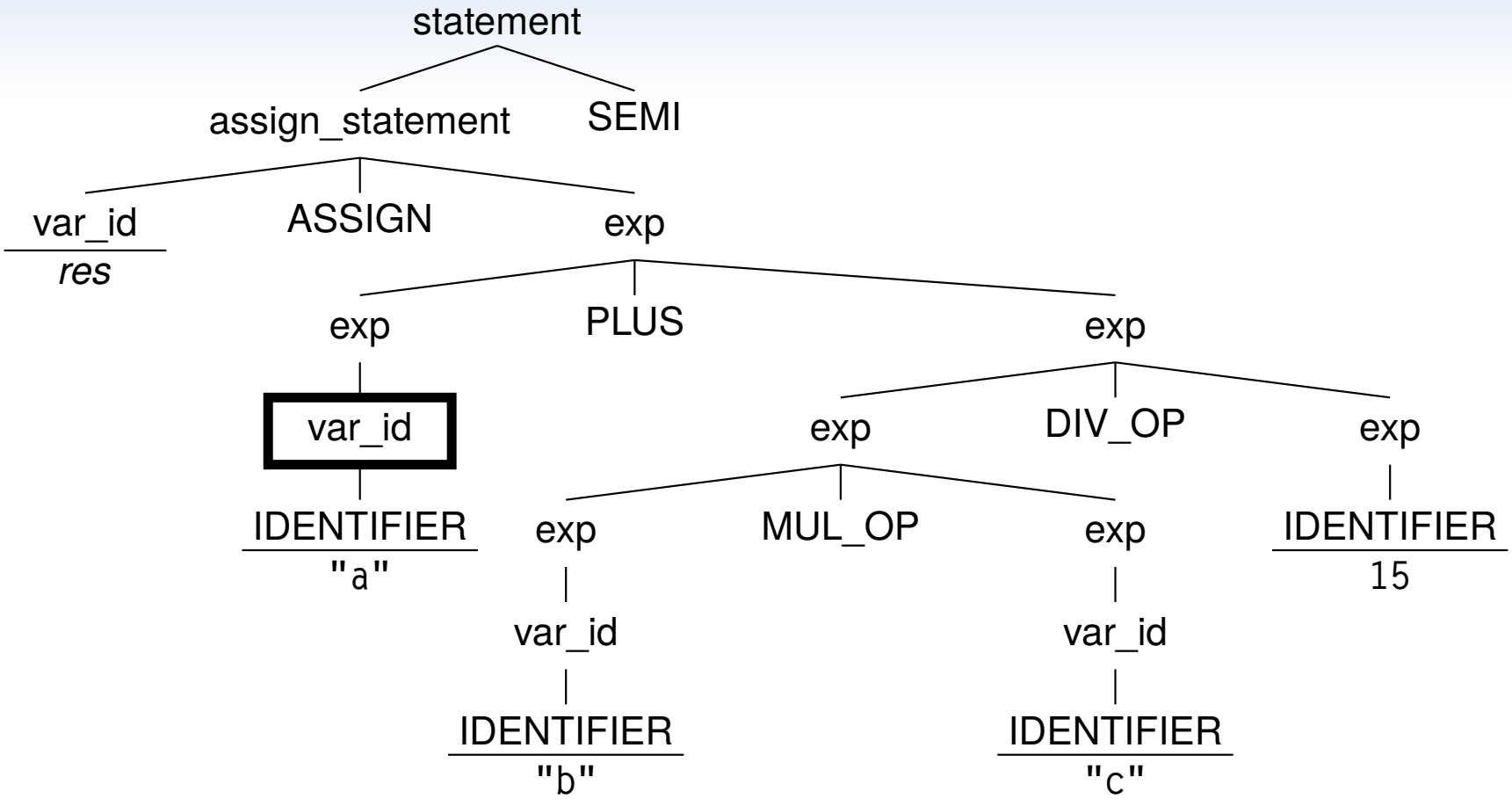
division before multiplication because they're left associative, so priority is skew to the left

basically the parser arrives at b*c and knows the next symbol is /. Shift or reduce? priority of * and / are the same, check if there is associativity and since there is, then decide to shift: b*c/15



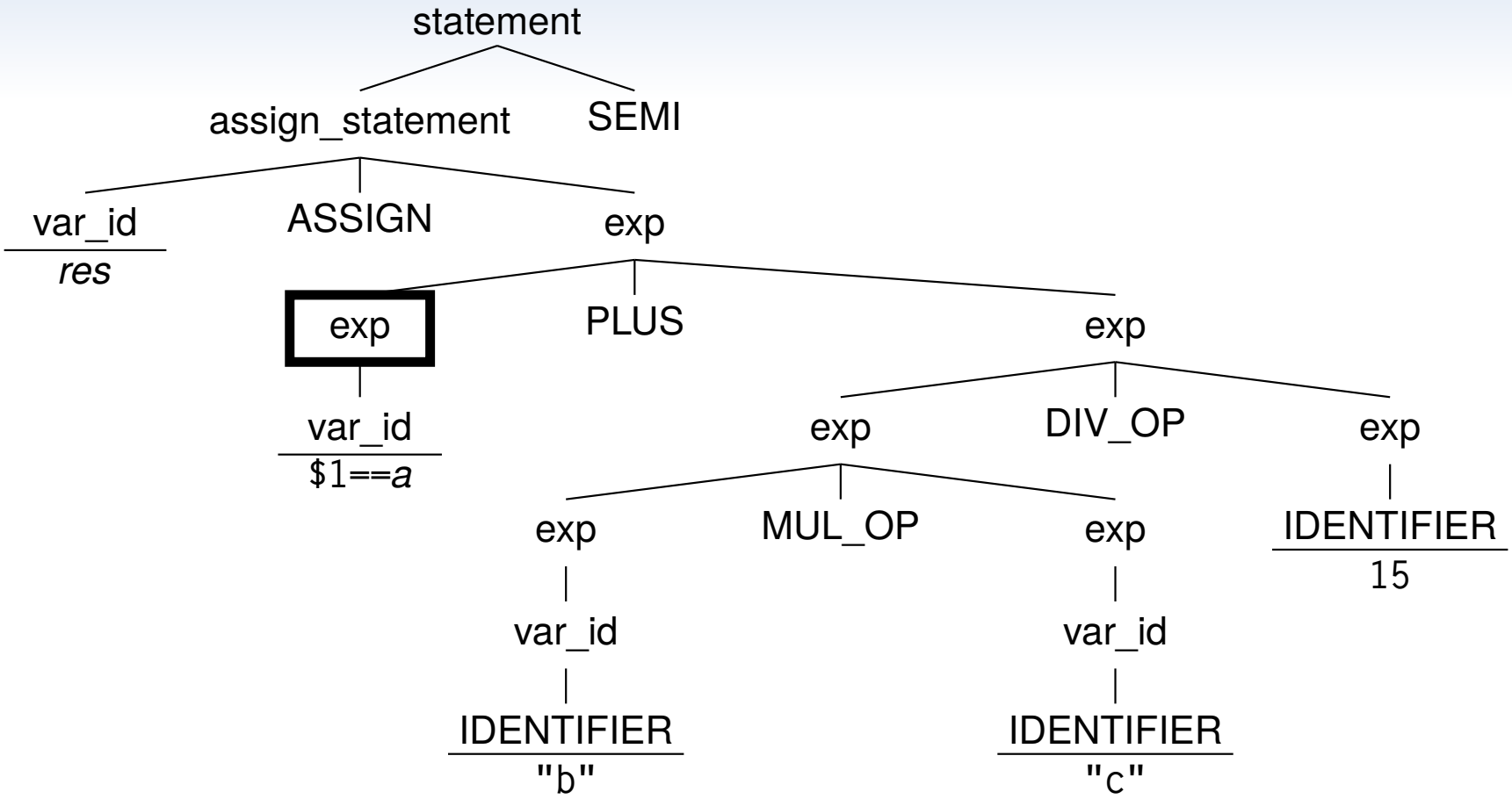
Code executed in the compiler

Lookup of symbol “res”



Code executed in the compiler

Lookup of symbol “a”

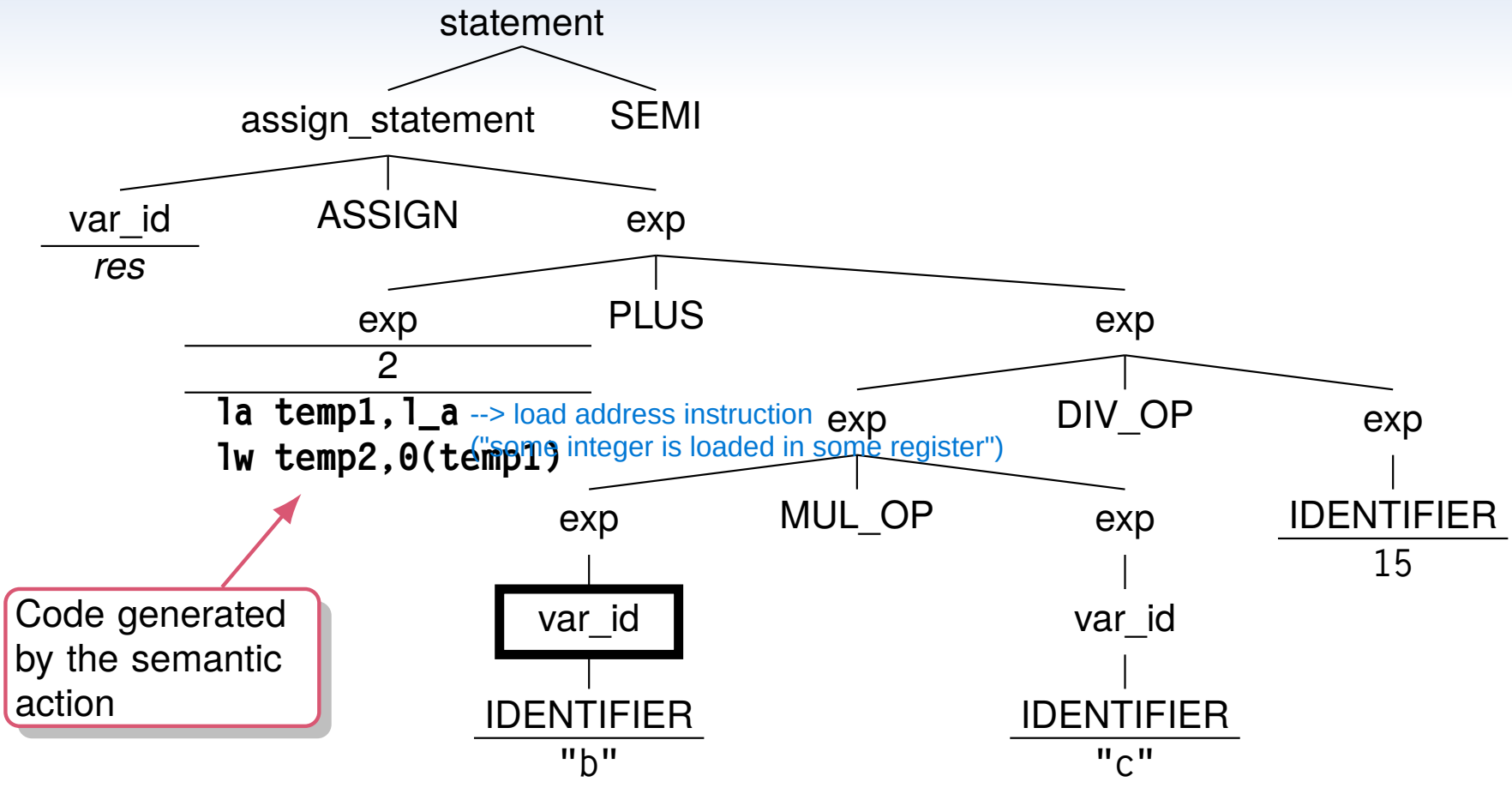


Code executed in the compiler

```
$$ = genLoadVariable(program, $1);
// At this point, $$ == 2
```

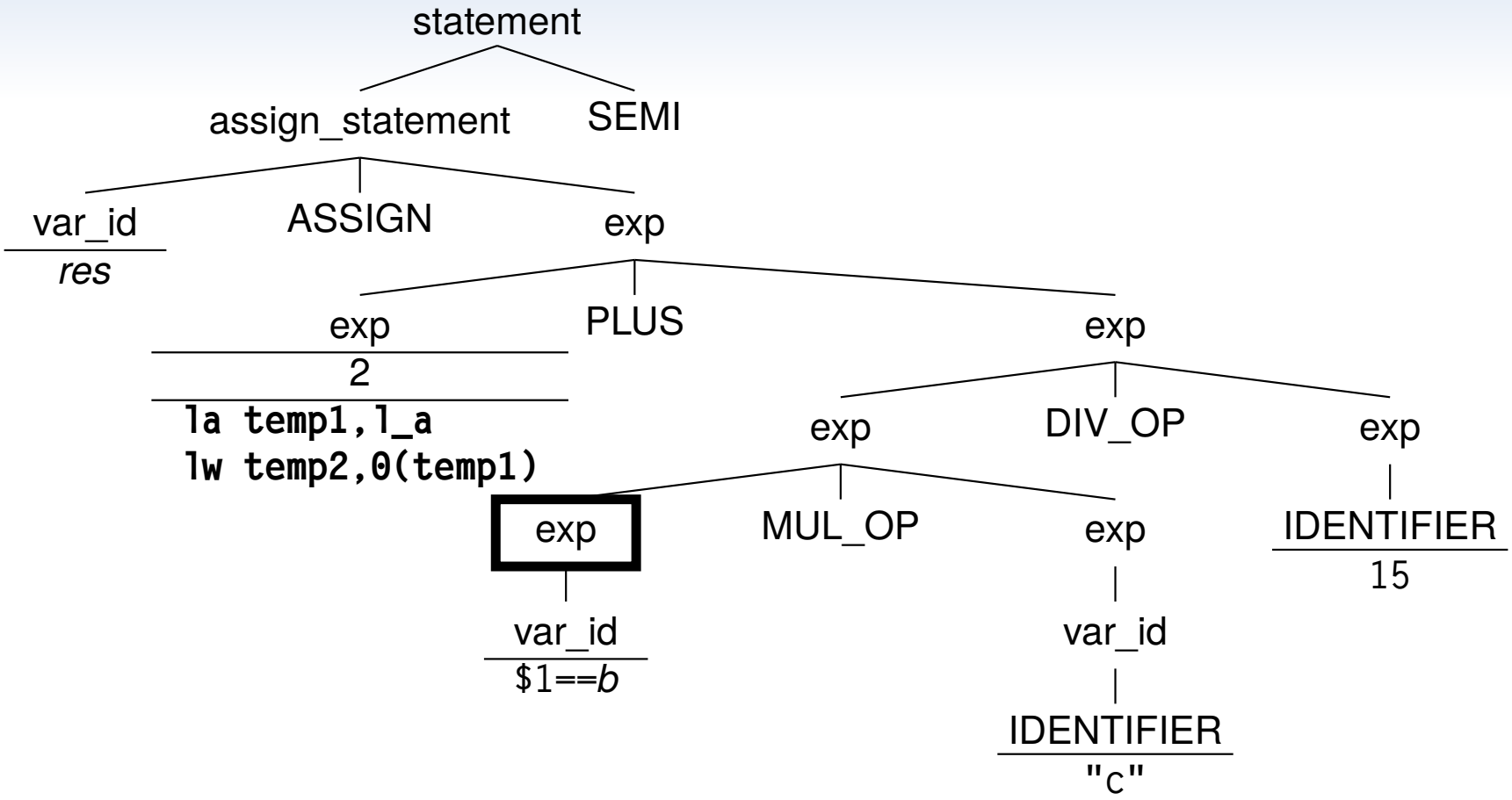
--> translated in assembly as: la instruction and lw instruction (see next slide). However for now we can see those functinos as black box

--> second time generating a register. This means that with genLoadVariable we put the read value ("a") into a new register, which is t2 since is the second register we generated till now. Now we assign to $$ the register t2. In this terms we can state that $$=2, meaning "in



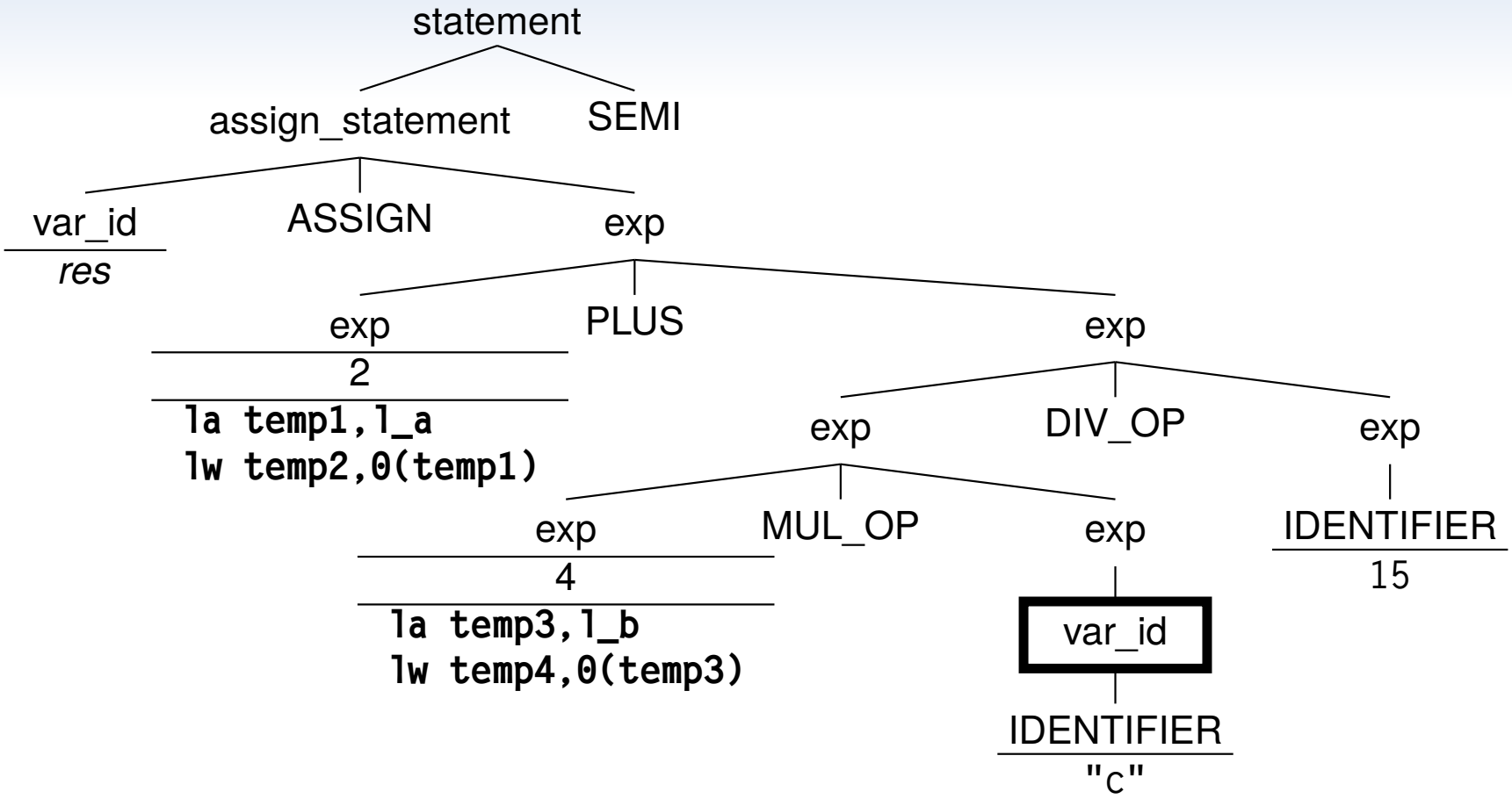
Code executed in the compiler

Lookup of symbol “b”



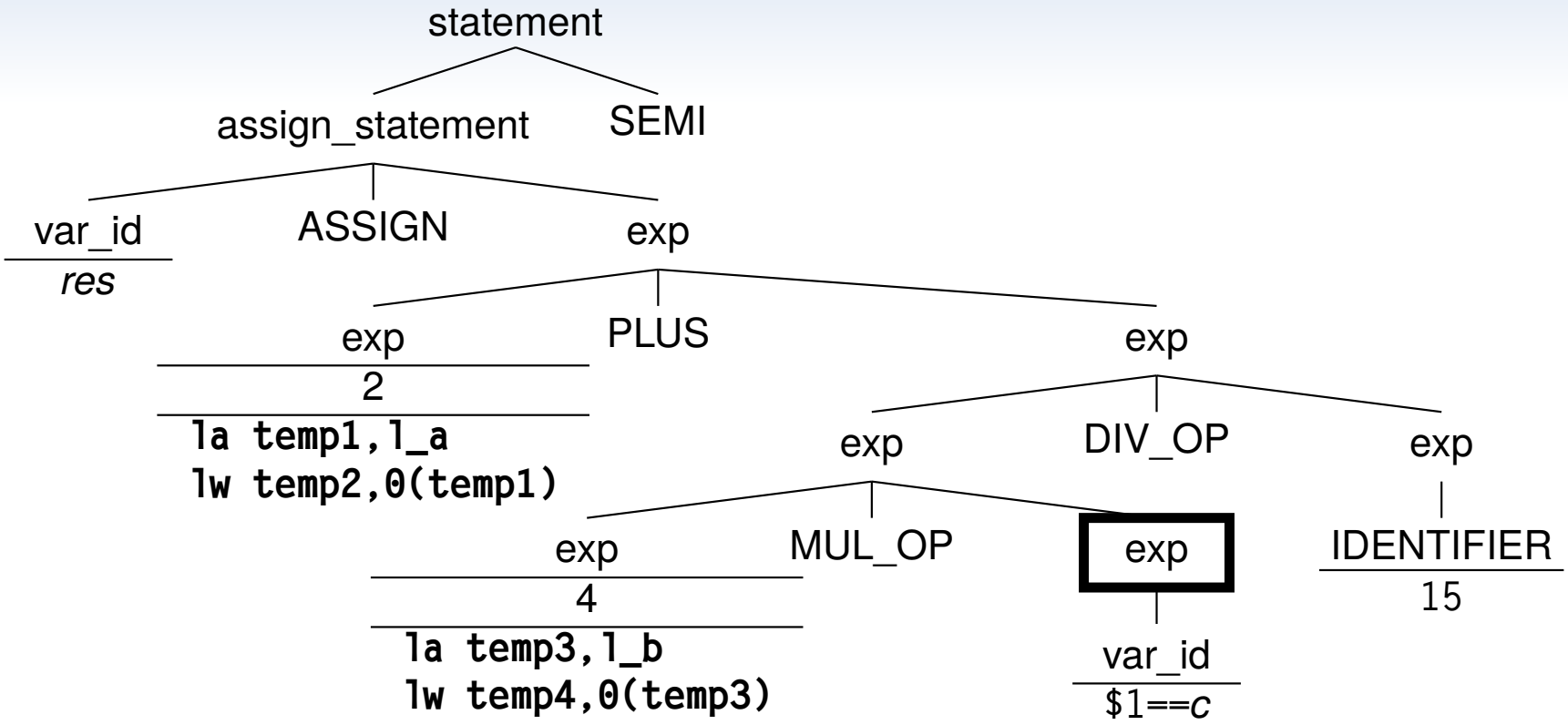
Code executed in the compiler

```
$$ = genLoadVariable(program, $1);
// At this point, $$ == 4
```



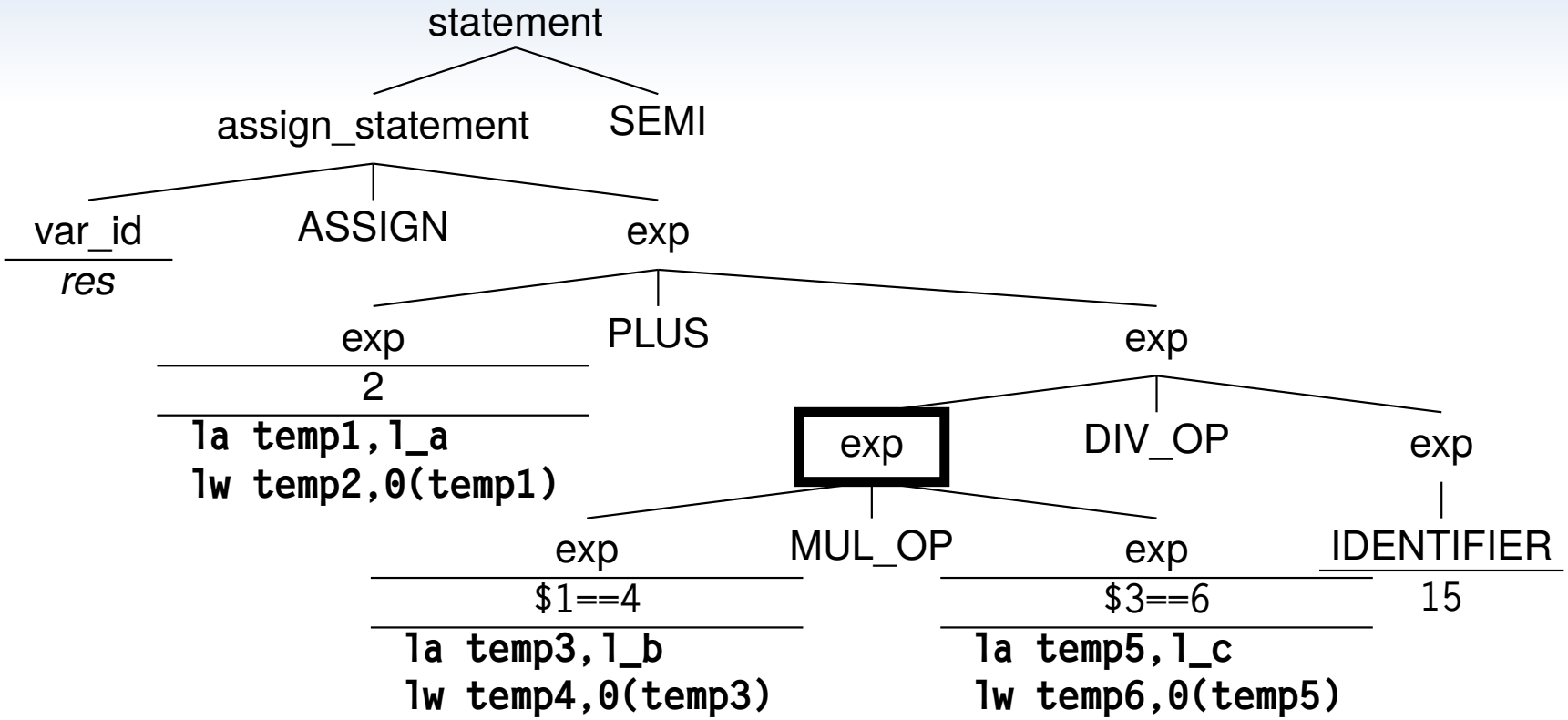
Code executed in the compiler

Lookup of symbol “c”



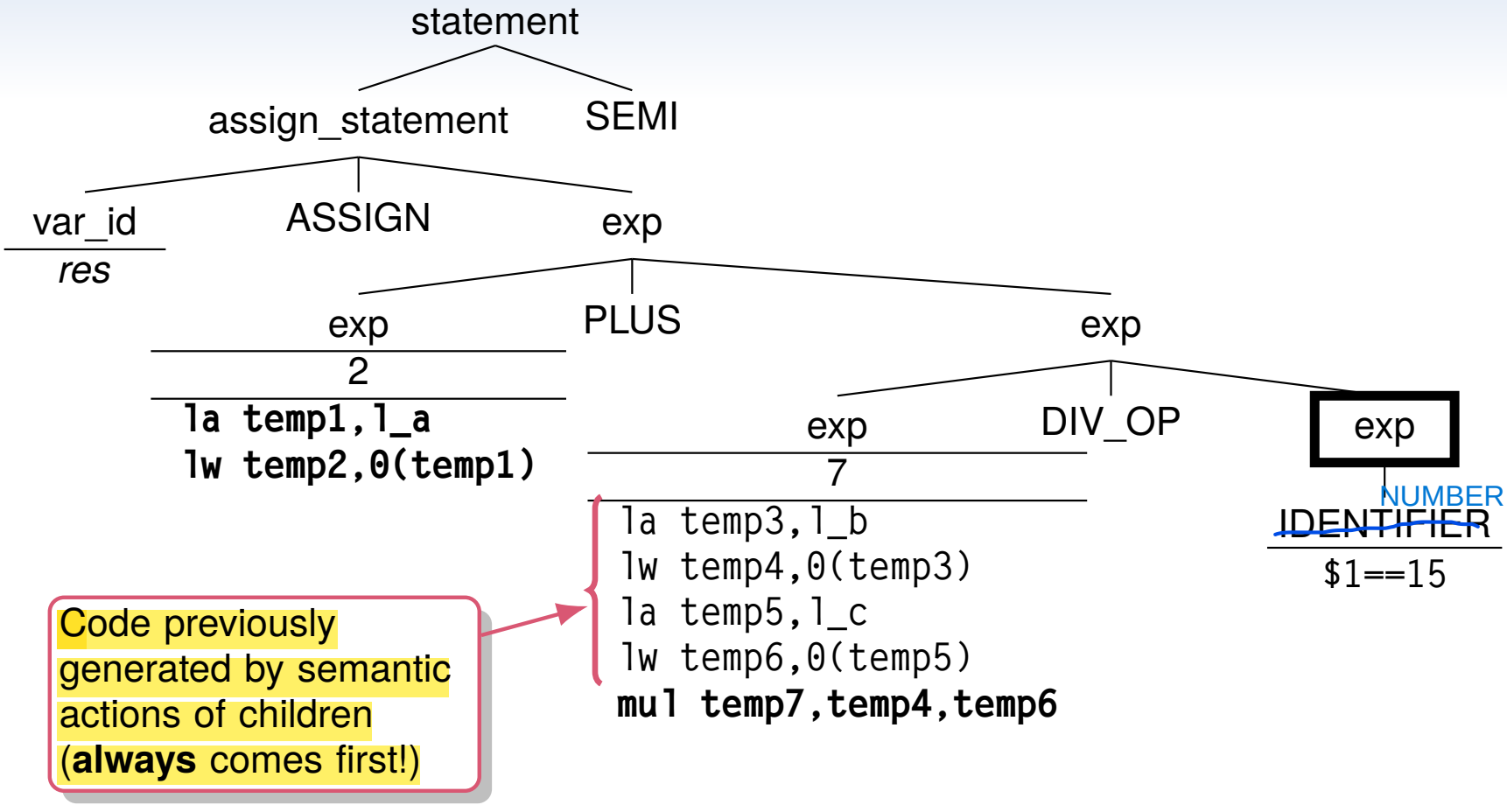
Code executed in the compiler

```
$$ = genLoadVariable(program, $1);
// At this point, $$ == 6
```



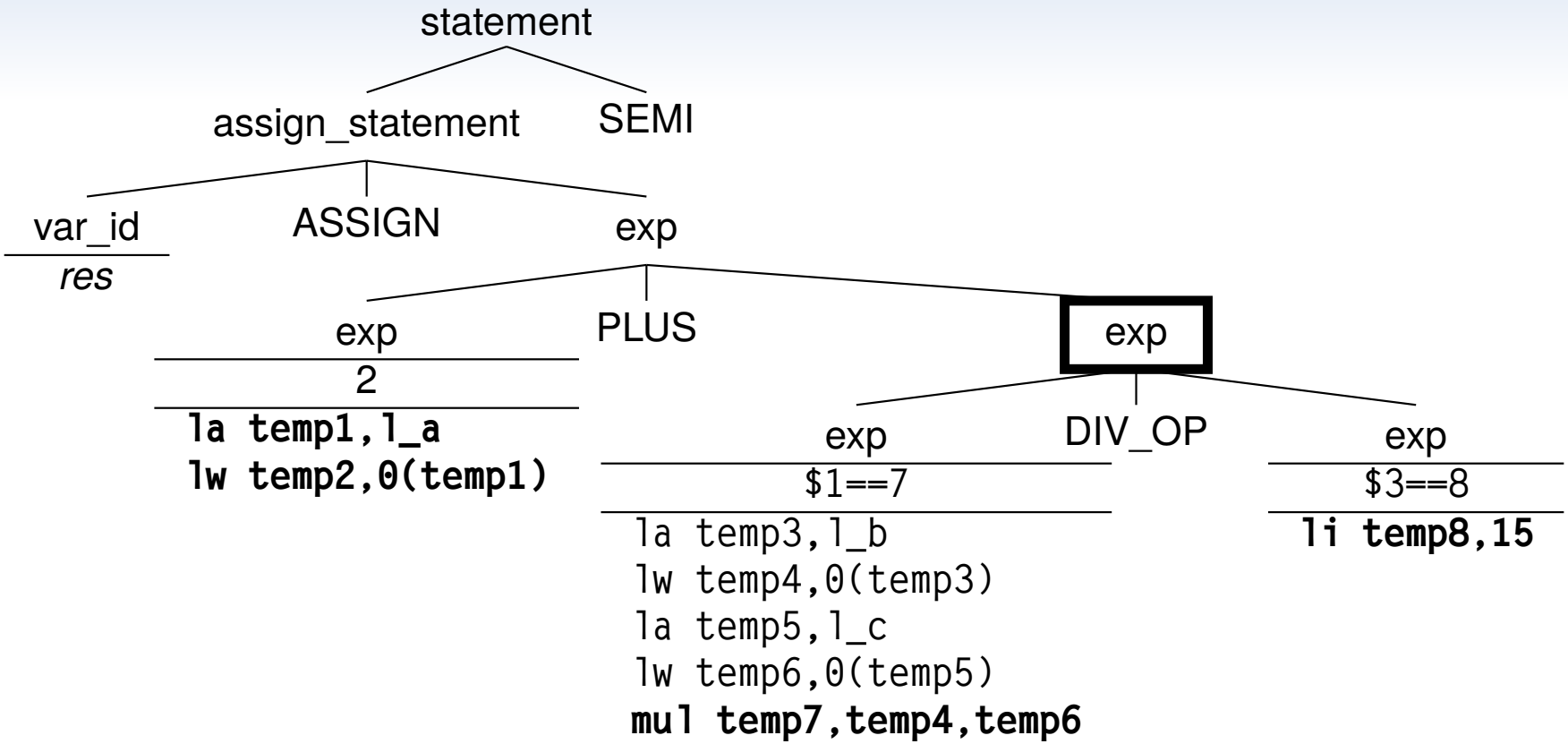
Code executed in the compiler

```
$$ = getNewRegister(program); // 7
genMUL(program, $$, $1, $3);
```



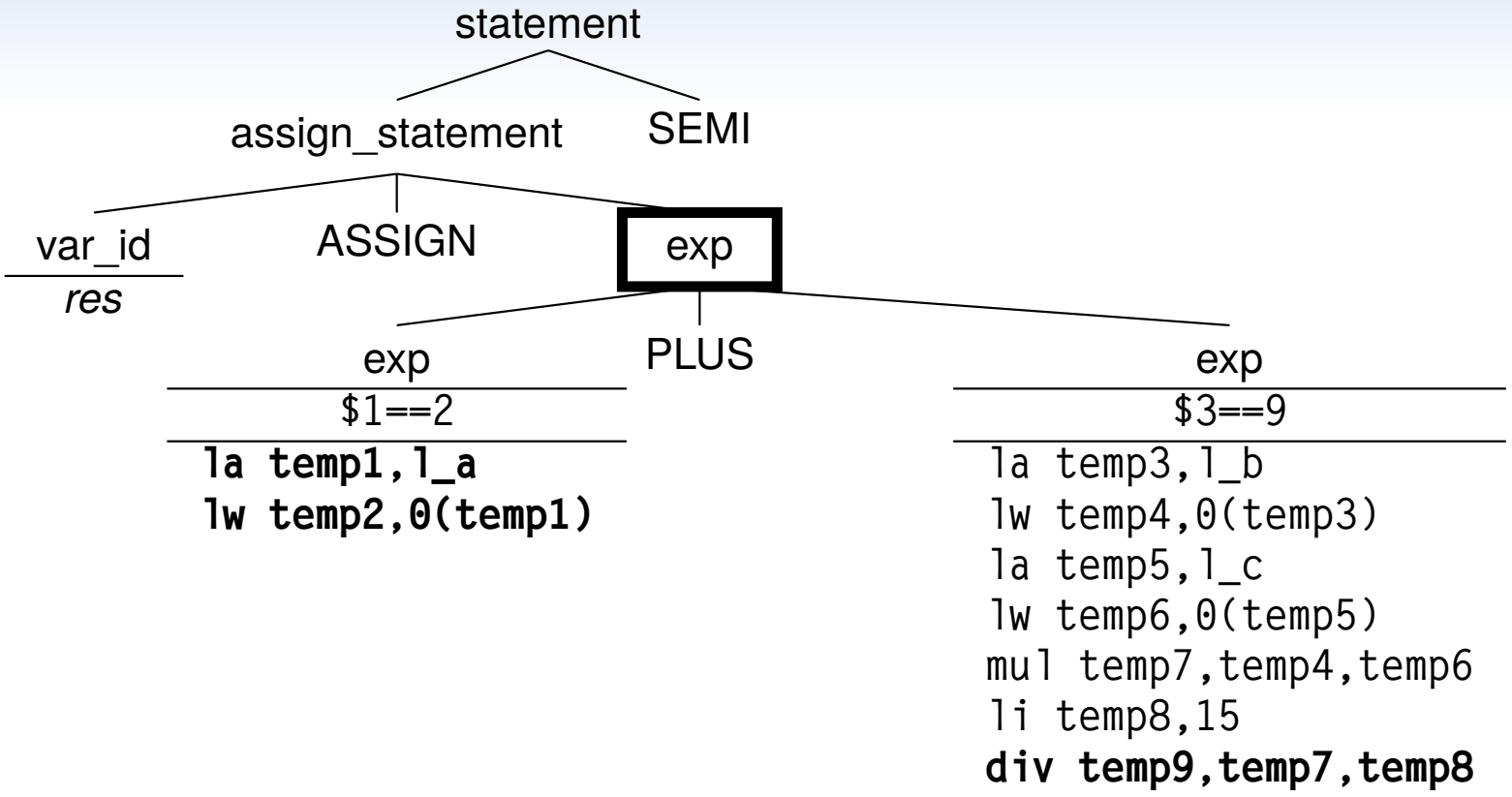
Code executed in the compiler

```
$$ = getNewRegister(program); // 8
genLI(program, $$, $1);
```



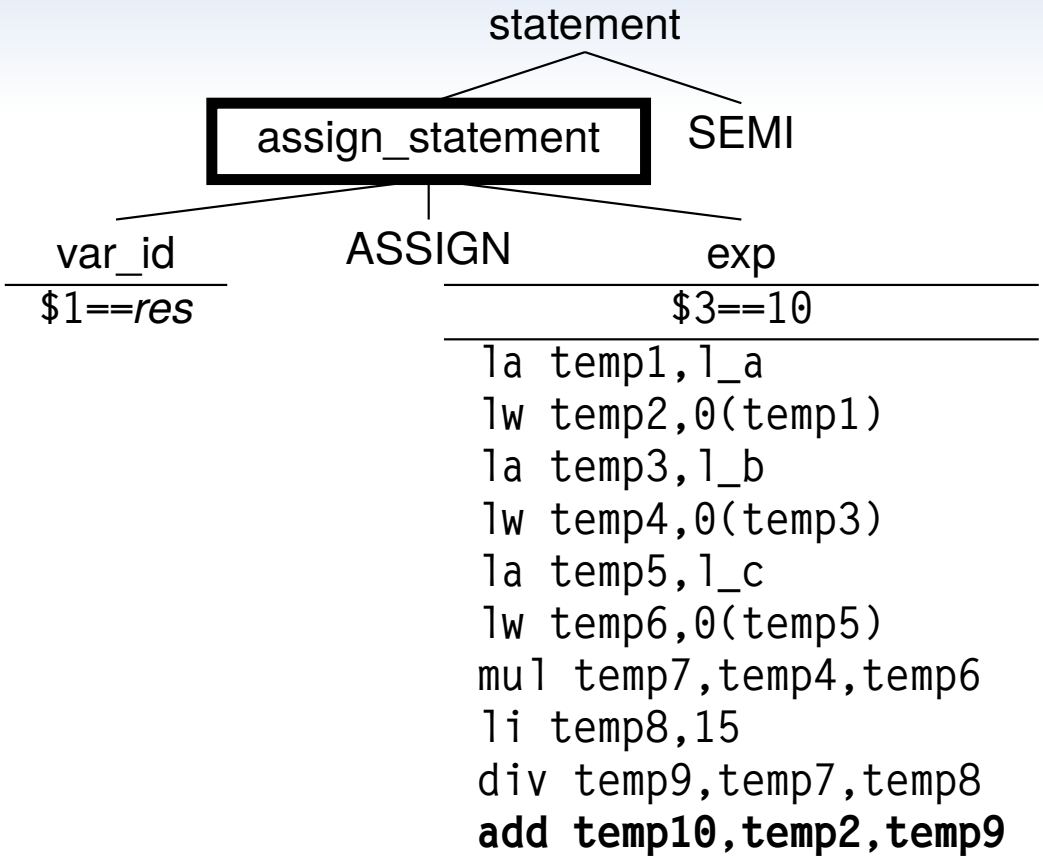
Code executed in the compiler

```
$$ = getNewRegister(program); // 9
genDIV(program, $$, $1, $3);
```



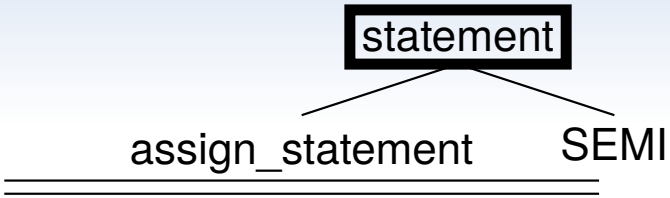
Code executed in the compiler

```
$$ = getNewRegister(program); // 10
genADD(program, $$, $1, $3);
```



Code executed in the compiler

```
genStoreRegisterToVariable(program, $1, $3);
```




```
la temp1,l_a
lw temp2,0(temp1)
la temp3,l_b
lw temp4,0(temp3)
la temp5,l_c
lw temp6,0(temp5)
mul temp7,temp4,temp6
li temp8,15
div temp9,temp7,temp8
add temp10,temp2,temp9
la temp11,l_res
sw temp12,0(temp11)
```

Just an offset

Code executed in the compiler

(...semantic actions of the parent of assign_statement...)

Logical operators

normalization

```
exp
  // ...
  | exp ANDAND exp
  {
    t_regID rNormOp1 = getNewRegister(program);
    genSNE(program, rNormOp1, $1, REG_0);
    t_regID rNormOp2 = getNewRegister(program);
    genSNE(program, rNormOp2, $3, REG_0);
    $$ = getNewRegister(program);
    genAND(program, $$, rNormOp1, rNormOp2);
  }
  | exp OROR exp
  {
    t_regID rNormOp1 = getNewRegister(program);
    genSNE(program, rNormOp1, $1, REG_0);
    t_regID rNormOp2 = getNewRegister(program);
    genSNE(program, rNormOp2, $3, REG_0);
    $$ = getNewRegister(program);
    genOR(program, $$, rNormOp1, rNormOp2);
  }
;
```

SNE->

SNE->

bitwise --> operation

For logical operators the operands must be **normalized**

- Operand $= 0$ $\rightarrow$ remains 0
- Operand $\neq 0$ $\rightarrow$ becomes 1

This is done by generating one **SNE** (set not equal) instruction per operand

Then we can generate the same instruction as bitwise operators

Contents

Branches

In general there are two kinds of branches:

Forward The label is **after** the branch

Backward The label is **before** the branch

Forward branch

```
        J label
        // ...
label:  // ...
        // ...
```

Backward branch

```
        // ...
label:  // ...
        // ...
        J label
```

Creating labels

Problem: ACSE is a **syntactic directed translator**

- Normally, labels that appear after a branch must be also **generated** after the branch
- We need a way to **allocate** a label without **generating** it – in other words, without **inserting it** into the instruction list

This is why ACSE provides **2** primary functions for creating labels:

- *createLabel()*
- *assignLabel()*

createLabel() creates a label structure **without inserting it into the instruction list**.

```
t_label *createLabel(t_program *program);
```

assignLabel() inserts the label in the instruction list.

```
void assignLabel(t_program *program, t_label *label);
```

Think of this function as if it **prints the label to the output**
(like *genXXX()* functions print *instructions* to the output)

While statements

Grammar:

```
while_statement
  : WHILE LPAR exp RPAR code_block
;
```

- The expression inside the parenthesis is called the **loop condition**
- First, the condition is computed

→ If it is true, the **body of the statement** is executed and we go back to the condition

→ Otherwise we continue with the rest of the program

- We need three semantic actions
- Trick: the label objects are stored in the semantic value of WHILE

parser.h

```
typedef struct {
  t_label *lLoop;
  t_label *lExit;
} t_whileStmt;
```

parser.y

```
%union {
    ...
    t_whileStmt whileStmt;
    ...
}
%token <whileStmt> WHILE
```

parser.y

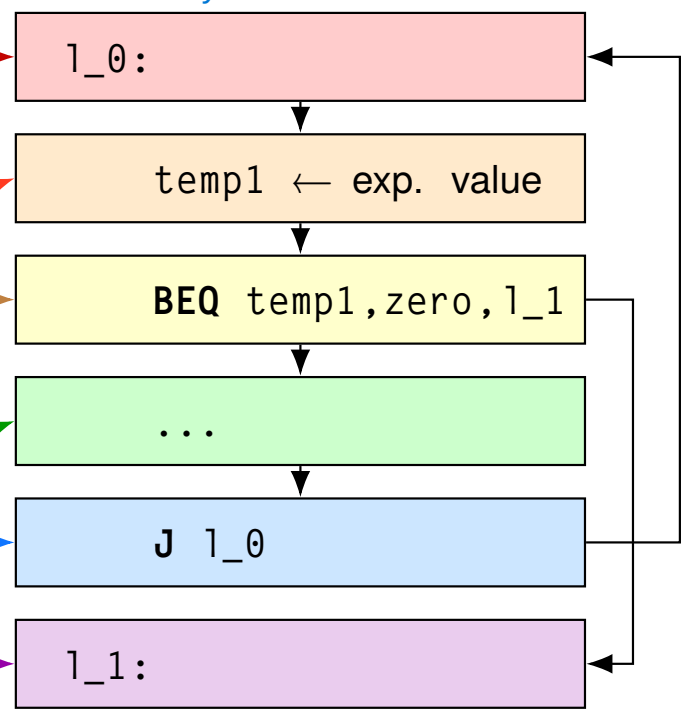
```
while_statement
   : WHILE $1
   {
$2   $1.lLoop = createLabel(program);
     assignLabel(program, $1.lLoop);
   } $3
   LPAR exp RPAR
   {
     $1.lExit = createLabel(program);not yet assigned
     genBEQ(program, $4, REG_0, $1.lExit);
   }
   code_block $6
   {
     genJ(program, $1.lLoop);
     assignLabel(program, $1.lExit);
   }
;
```

Semantic actions:

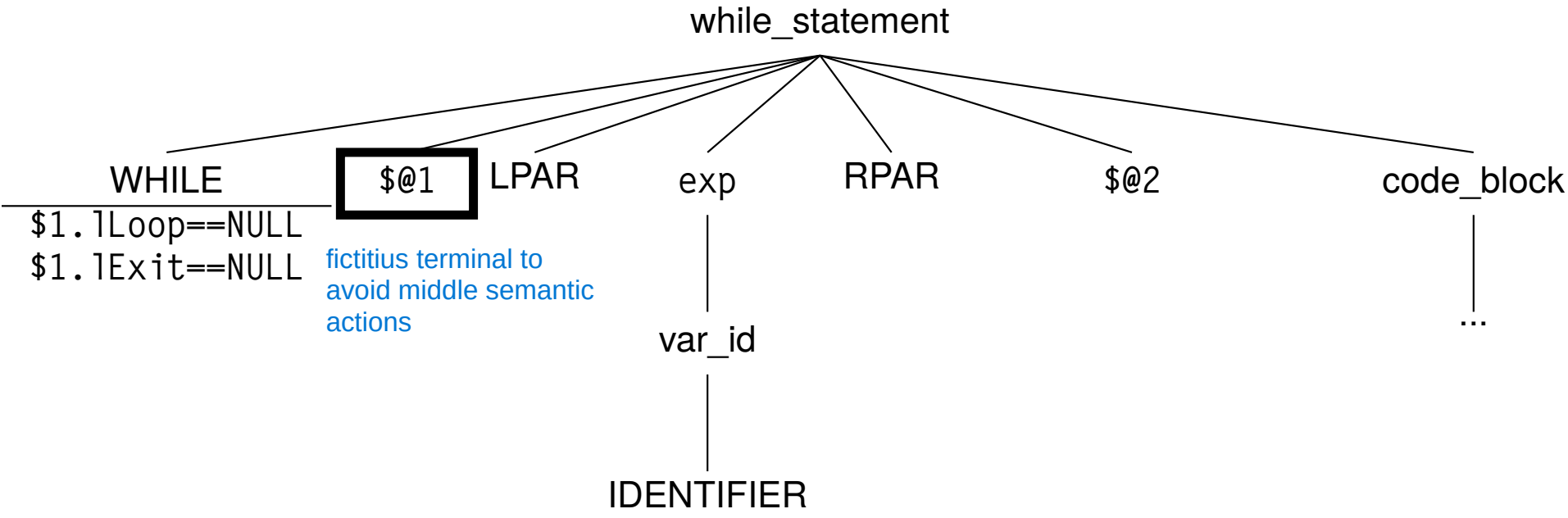
1. Adds a label before the expression computation code for later
2. Generates jump to after the statement if the condition is false
3. Jumps back to the condition code
   Adds the label to after the statement

Usual error is "putting" the red block into the yellow block

```
while_statement:
  WHILE
  {
    $1.lLoop = createLabel(program);
    assignLabel(program, $1.lLoop);
  }
  LPAR exp RPAR
  {
    $1.lExit = createLabel(program);
    genBEQ(program, $4, REG_0, $1.lExit);
  }
  code_block
  {
    genJ(program, $1.lLoop);
    assignLabel(program, $1.lExit);
  }
;
```



`while (v) { v=v-1; }`

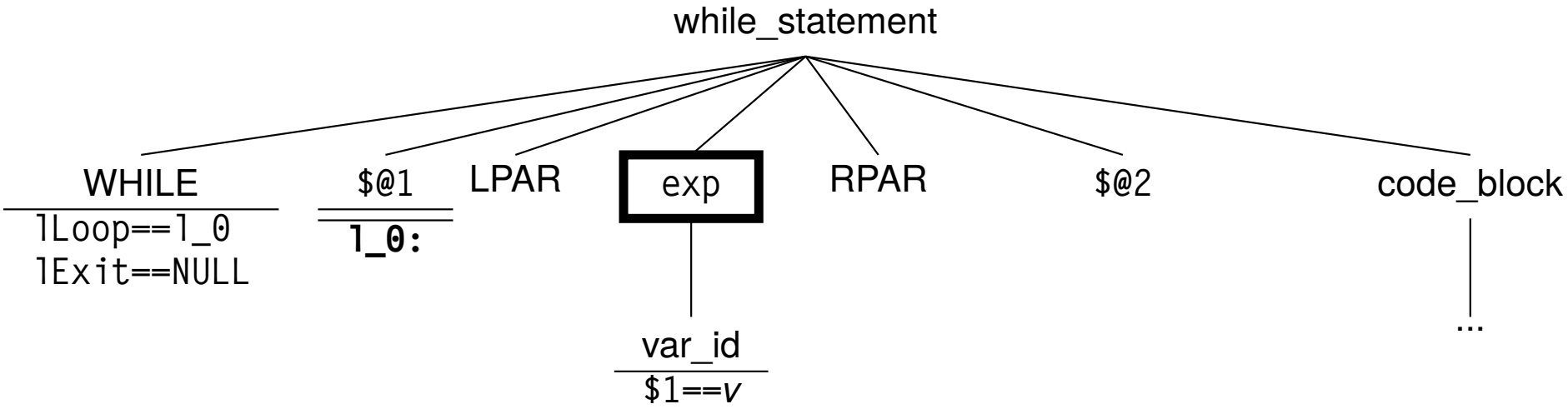


Code executed in the compiler

```
$1.lLoop = createLabel(program);
assignLabel(program, $1.lLoop);
```

We are spelling the register names as $\texttt{t}\langle n \rangle$ instead of $\texttt{temp}\langle n \rangle$ for brevity

```
while (v) { v=v-1; }
```

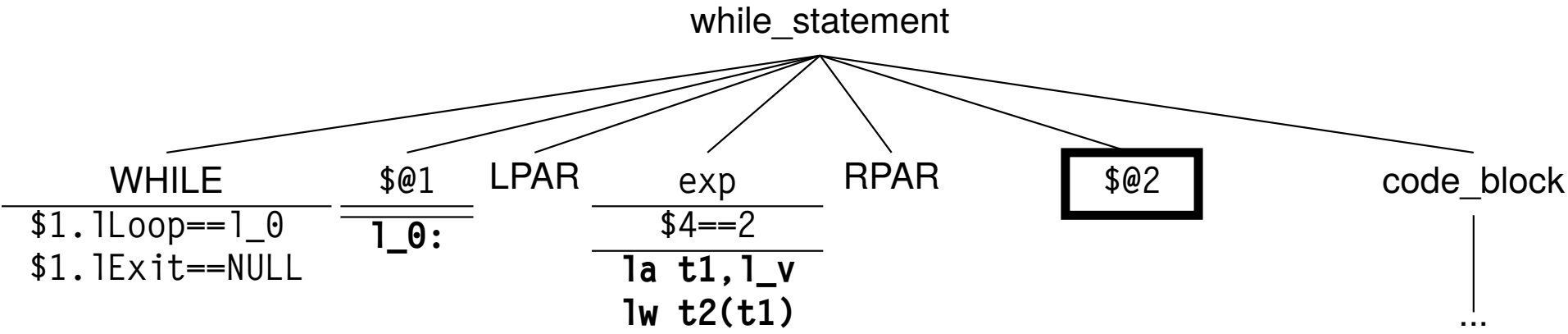


Code executed in the compiler

```
$$ = genLoadVariable(program, $1);
```

We are spelling the register names as t⟨*n*⟩ instead of temp⟨*n*⟩ for brevity

```
while (v) { v=v-1; }
```

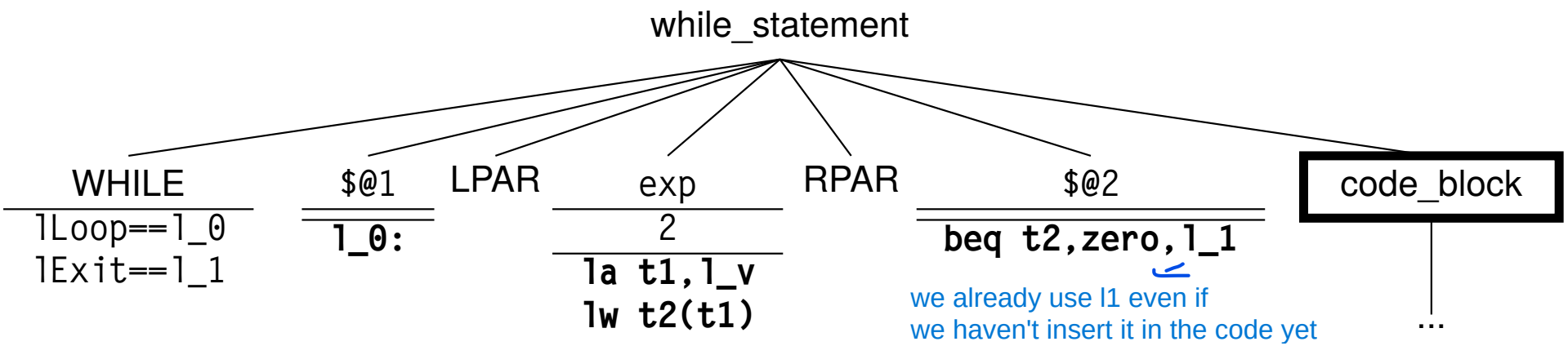


Code executed in the compiler

```
$1.lExit = createLabel(program);
genBEQ(program, $4, REG_0, $1.lExit);
```

We are spelling the register names as t⟨*n*⟩ instead of temp⟨*n*⟩ for brevity

```
while (v) { v=v-1; }
```

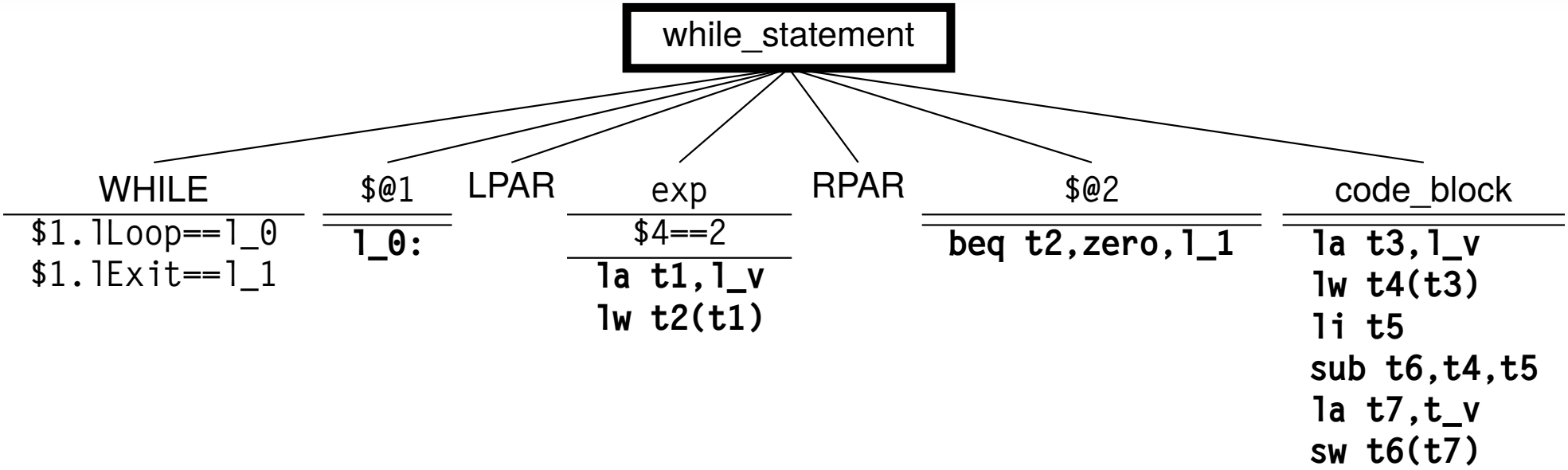


Code executed in the compiler

Multiple rules related to the assignment `v=v-1;`

We are spelling the register names as $\texttt{t}\langle n \rangle$ instead of $\texttt{temp}\langle n \rangle$ for brevity

```
while (v) { v=v-1; }
```



Code executed in the compiler

```
genJ(program, $1.lLoop);
assignLabel(program, $1.lExit);
```

We are spelling the register names as t⟨*n*⟩ instead of temp⟨*n*⟩ for brevity

```
while (v) { v=v-1; }
```

while_statement

```
l_0:
 la t1,l_v
 lw t2(t1)
 beq t2,zero,l_1
 la t3,l_v
 lw t4(t3)
 li t5
 sub t6,t4,t5
 la t7,t_v
 sw t6(t7)
 j l_0
l_1:
```

Code executed in the compiler

(...semantic actions of the parent of while_statement...)

We are spelling the register names as t⟨*n*⟩ instead of temp⟨*n*⟩ for brevity

Do-While statements

```
do_while_statement
  : DO
  {
    $1 = createLabel(program);
    assignLabel(program, $1);
  }
  code_block WHILE LPAR exp RPAR
  {
    genBNE(program, $6, REG_0, $1);
  }
;
```

The expression is computed after the block

- Much simpler block structure and semantic actions
- The semantic value of DO is a label object pointer

If statements

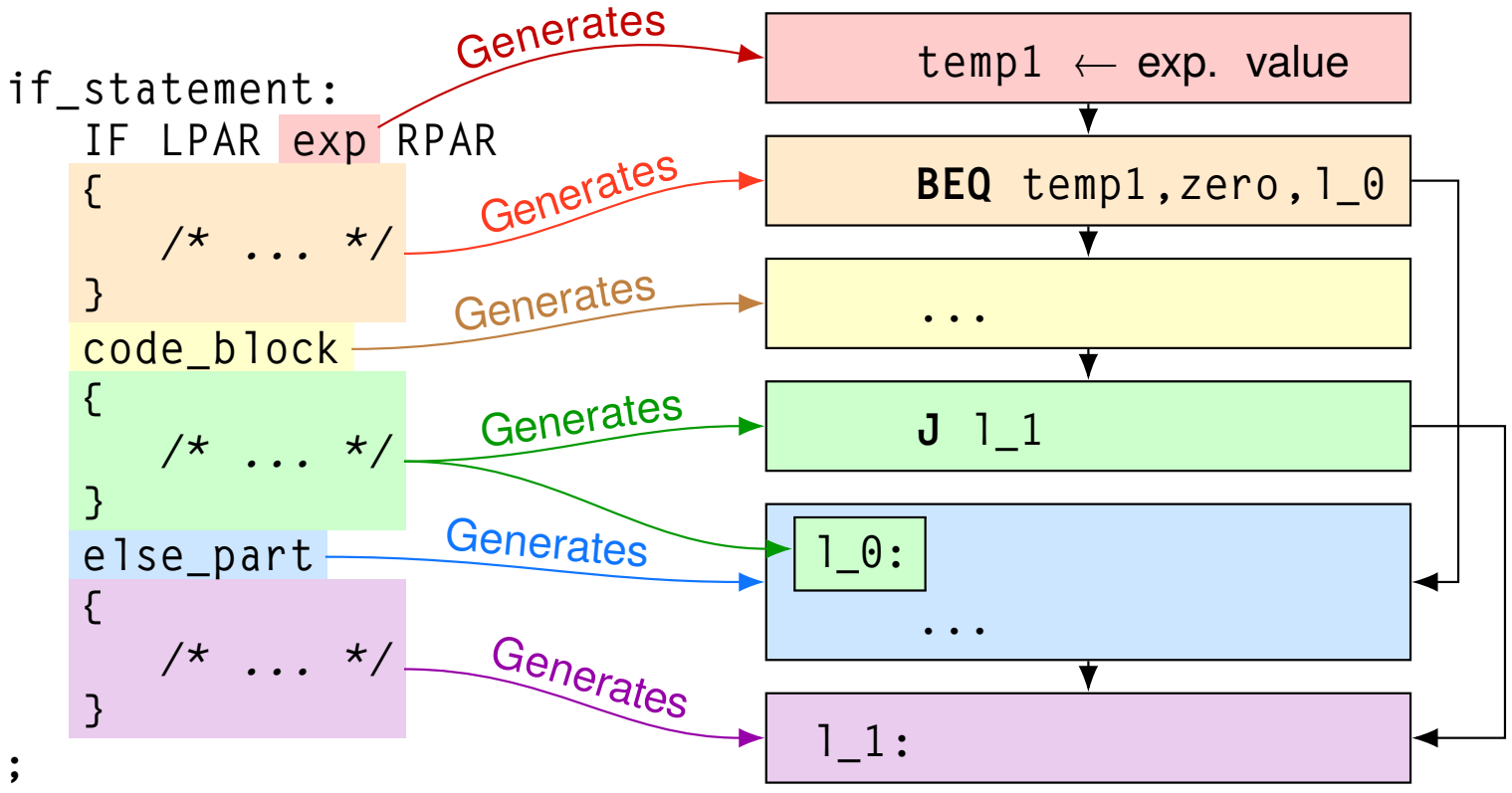
Grammar:

```
if_statement
  : IF LPAR exp RPAR code_block else_part
;
else_part
  : ELSE code_block
  |
;
```

- The expression inside the parenthesis is called the **condition**
- The first code block (*then* part) is executed only if the **condition** is **not equal to zero**
- `else_part` is effectively an optional code block that gets executed in alternative to the first

Semantic actions:

1. Generates jump to the *else* part if the condition is false
2. Generates jump over the *else* part if the condition is true
3. Assigns the label after the statement



- The label must be shared between the first and second action
- *else_part* doesn't need any semantic actions

parser.h

```
typedef struct {
  t_label *lElse;
  t_label *lExit;
} t_ifStmt;
```

parser.y

```
%union {
    ...
    t_ifStmt ifStmt;
    ...
}
%token <ifStmt> IF
```

parser.y

```
if_statement
  : IF LPAR exp RPAR
  {
    $1.lElse = createLabel(program);
    genBEQ(program, $3, REG_0, $1.lElse);
  }
  code_block
  {
    $1.lExit = createLabel(program);
    genJ(program, $1.lExit);
    assignLabel(program, $1.lElse);
  }
  else_part
  {
    assignLabel(program, $1.lExit);
  }
;
```

Contents

Return

```
return_statement
  : RETURN
  {
    genExit0Syscall(program);
  }
;
```

A return statement simply exits from the program, and hence translates to a call to the `Exit0` syscall.

Read

```
read_statement
  : READ LPAR var_id RPAR
  {
    t_regID rTmp = getNewRegister(program);
    genReadIntSyscall(program, rTmp);
    genStoreRegisterToVariable(program, $3, rTmp);
  }
;
```

A read statement translates to a `ReadInt` syscall. The value it returns is then stored in the appropriate variable.

Write

```
write_statement
  : WRITE LPAR exp RPAR
  {
    genPrintIntSyscall(program, $3);
    t_regID rTmp = getNewRegister(program);
    genLI(program, rTmp, '\n');
    genPrintCharSyscall(program, rTmp);
  }
;
```

A write statement translates to a `PrintInt` syscall.

- In the RARS syscalls, `PrintInt` does not add a final newline

→ We need to also generate a call to `PrintChar` do make it happen