Code Generation

The Stages of Compilation

- The compilation process can be broadly divded into four stages.
  - **Scanning:** Group the individual characters in the source into meaningful chunks called tokens, and detect errors related to syntax of tokens.
  - **Parsing:** Group the tokens into meaningful high-level structures like statements and expressions, and detect errors related to syntax of structures.
  - **Semantic Analysis:** Gather further information about the semantics (meaning) of the program, e.g. scope of identifiers and types of expressions, and detect errors related to semantics.
    - The program should be free of compile-time errors after this stage.
  - **Code Generation:** Translate each structural component of the program into the target language using the information obtained in the previous stages.

Code Generation

- The idea behind code generation is very similar to type checking.
- We traverse the tree, and depending on the rule at the current node, we output code implementing the functionality of the rule.
- Type checking has a "correct answer", but there are (infinitely) many possible correct code generation strategies.
- Modern compilers use very complex algorithms to produce code that is optimized for speed (i.e., the generated program should be fast).
- We will focus on ease of implementation at the expense of speed.
- First, let's look at some examples of generated code.

Code Generation: Basic Example

```
int wain(int a, int b) { return a; }
```

- WLP4 programs, when compiled to MIPS, take parameters in $1 and $2 (either via mips.twoints or mips.array) and return a value in $3.
- Here are two possible ways to generate code for this program.

```
add $3, $1, $0
jr $31
```

```
lis $4
.word 4
sw $1, -4($30)
sub $30, $30, $4
sw $2, -4($30)
sub $30, $30, $4
sub $29, $30, $4
lw $3, 8($29)
jr $31
```

- The second program is much more complicated, but is actually **easier to generate**.

Handling Variables

```
int wain(int a, int b) {
  int c = 240;
  int d = 241;
  int e = 242;
  return d;
}
```

- Let's say we try to compile a program with non-parameter local variables. The parameters start out in \$1 and \$2, but what about other variables?
- Do we put them in \$3, \$4 and \$5? What if we run out of registers?

Handling Variables

```
int wain(int a, int b) {
  int c = 240;
  int d = 241;
  int e = 242;
  return d;
}
```

- The reason the second program is easier to generate is because it stores all variables **on the stack**. We will follow this strategy.
- There could be more variables than registers, and allocating some to registers and some to the stack makes things more complicated.

The Frame Pointer

```
int wain(int a, int b) {
   int c = 240;
   int d = 241;
   int e = 242;
   return d;
}
```

- Suppose we store our variables on the stack as follows.

	Stack
$30	e = 242
$30+4	d = 241
$30+8	c = 240
$30+12	b = $2
$30+16	a = $1

- We can access them using stack pointer offsets, e.g., to load "d" into $3 we could use "lw $3, 4($30)".

The Frame Pointer

```
int wain(int a, int b) {
  int c = 240;
  int d = 241;
  int e = 242;
  return d;
}
```

- Suppose we store our variables on the stack as follows.

	Stack
$30	e = 242
$30+4	d = 241
$30+8	c = 240
$30+12	b = $2
$30+16	a = $1

- But if $30 changes for any reason, all our offsets change, which makes code generation for variable accesses harder.

The Frame Pointer

```
int wain(int a, int b) {
   int c = 240;
   int d = 241;
   int e = 242;
   return d;
}
```

		Stack
\$29-8	\$30	e = 242
\$29-4	\$30+4	d = 241
\$29	\$30+8	c = 240
\$29+4	\$30+12	b = \$2
\$29+8	\$30+16	a = \$1

- To solve this problem, we make a copy of \$30 in \$29 and do not change it after initialization.
- Offsets from \$29 stay constant throughout the program.

The Frame Pointer

```
int wain(int a, int b) {
  int c = 240;
  int d = 241;
  int e = 242;
  return d;
}
```

		Stack
$29-8	$30	e = 242
$29-4	$30+4	d = 241
$29	$30+8	c = 240
$29+4	$30+12	b = $2
$29+8	$30+16	a = $1

- $29 is called the **frame pointer** and is used to access variables in the current **stack frame**.
- Each procedure gets its own stack frame (more on this later).

Code Generation

```
int wain(int a, int b) { return a; }
```

- Now we can make sense of this generated code.

```
lis $4
.word 4          ; initialize constant $4 = 4
sw $1, -4($30)   ; push a = $1 (offset 8 from frame pointer)
sub $30, $30, $4 ; decrement stack pointer $30 by 4
sw $2, -4($30)   ; push b = $2 (offset 4 from frame pointer)
sub $30, $30, $4 ; decrement stack pointer $30 by 4
sub $29, $30, $4 ; set the frame pointer $29 to $30-4
lw $3, 8($29)    ; set $3 = a
jr $31           ; return
```

Notes about the Frame Pointer

- In the previous example, we set the frame pointer $29 after pushing the parameters of wain.
- The exact position of the frame pointer doesn't really matter as long as you are consistent about it and compute the offsets correctly.
- The scheme on the previous slide is the one used by the course notes.
- This scheme uses the frame pointer to separate parameters and non-parameter local variables.
  - Positive offsets for parameters, non-positive for non-parameters.

Code Generation: Expressions

```
int wain(int a, int b) {
  int c = 241;
  return a+b-c;
}
```

- An intuitive way to represent this as a MIPS program might be:

```
add $5, $1, $2
lis $3
.word 241
sub $3, $5, $3
jr $31
```

- But again, it's hard to write a code generator that works this way!

Code Generation: Expressions

```
int wain(int a, int b) {
  int c = 241;
  return a+b-c;
}
```

- We run into a similar problem as for handling variables.
- We need to store *temporary expression results* somewhere. For example, we put a+b in $5 so that we could use $3 to hold c.
- If we use registers, we will run out of registers when processing large expressions, and it's also tricky to decide which registers to use.

Code Generation: Expressions

```
int wain(int a, int b) {
  int c = 241;
  return a+b-c;
}
```

- Solution: Use the stack again!
- When we encounter a single term (variable or constant) we load it into $3 (like we would if we were returning that value).
- For a binary operation, we compute the left subexpression, push the result to the stack, compute the right subexpression, pop the left result, and use the two results to get the final value of the expression.

Code Generation: Expressions

```
int wain(int a, int b) {
  int c = 241;
  return a+b-c;
}
```

- Let's just consider the "a+b" part.

```
$3 = a            lw $3, a_offset($29) ; assuming frame pointer is set up
push $3 to stack  sw $3, -4($30)
                  sub $30, $30, $4     ; assuming $4 contains 4
$3 = b            lw $3, b_offset($29)
pop into $5       add $30, $30, $4
                  lw $5, -4($30)
$3 = a+b          add $3, $5, $3
```

Code Generation: Expressions

```
int wain(int a, int b) {
  int c = 241;
  return a+b-c;
}
```

- Let's just consider the "a+b" part.

```
$3 = a             lw $3, a_offset($29) ; assuming frame pointer is set up
push $3 to stack   sw $3, -4($30)       ; offsets from $30 change here!
                   sub $30, $30, $4     ; assuming $4 contains 4
$3 = b             lw $3, b_offset($29) ; but $29 stays constant!
pop into $5        add $30, $30, $4
                   lw $5, -4($30)
$3 = a+b           add $3, $5, $3
```

Code Generation: Expressions

```
int wain(int a, int b) {
  int c = 241;
  return a+b-c;
}
```

- The overall structure of "a+b-c" is the same:

```
$3 = a+b           ; ENTIRE BLOCK OF CODE FROM PREVIOUS SLIDE GOES HERE
push $3 to stack   sw $3, -4($30)
                   sub $30, $30, $4
$3 = c             lw $3, c_offset($29)
pop into $5        add $30, $30, $4
                   lw $5, -4($30)
$3 = (a+b)-c       sub $3, $5, $3
```

Code Generation: Expressions

```
int wain(int a, int b) {
  int c = 241;
  return a+b-c;
}
```

- How do we know that we need to do "(a+b)-c" and not "a+(b-c)"?
- The parse tree tells us!! Our grammar unambiguously forces the tree structure to account for order of operations correctly.
- If you just traverse the tree and use this push/pop strategy at each node, your code will evaluate expressions in the right order.

Notes about Expressions

- Because WLP4 programs return a value in \$3, it is convenient to follow the convention that *all expressions* generate code which puts their result in \$3.
- This means we do not need extra code to move an expression result to \$3 for return purposes.
- However, this also means that when we pop the "left subexpression result" from the stack, we can't put it in \$3 since \$3 will already be holding the "right subexpression result".
- We will use \$5 as a temporary register to pop values into, but this is an arbitrary choice that has no special meaning.

Implementing Code Generation

- For now, we will assume that wain is the only procedure.
- Later, we will discuss how to non-wain procedures.
- The general idea is to output code based on the rules in the tree.
- If wain is the only procedure, then the tree starts off like this:

start → BOF procedures EOF
procedures → main
main → INT WAIN LPAREN dcl COMMA dcl RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

Implementing Code Generation

- Because we are using a concrete syntax tree instead of an abstract syntax tree, a lot of tree nodes are superfluous.
- start → BOF procedures EOF
  - To generate code for "start", generate code for "procedures".
- procedures → main
  - To generate code for "procedures", generate code for "main".
- For simplicity, we will assume you have a single function called "code" that handles code generation for all rules.
  - It's arguably cleaner to split it into multiple smaller helper functions.
- For these rules, "code" can just do a recursive call on the child node.

Implementing Code Generation

main → INT WAIN LPAREN dcl COMMA dcl RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

- This rule defines the overall structure of the main function. You will need to look at the subparts and generate code for each.
- The first step is to set up the frame pointer and put the local variables on the stack.
- Before examining any subtrees, initialize constants, e.g., put 4 in $4.
- Another useful constant is 1, but $1 contains the first parameter of wain, so the course notes use $11 to store 1.

Implementing Code Generation

main → INT WAIN LPAREN dcl COMMA dcl RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

- The two "dcl" subtrees and the "dcls" subtree define the local variables of wain (parameters and non-parameters).
- Using the same techniques as in the semantic analysis phase, create a table that maps local variable names to *frame pointer offsets*.
  - You could also just directly extend your semantic analysis code instead of reusing the techniques.
  - You could add the offsets to your existing symbol table structure instead of creating a separate table.

Implementing Code Generation

main → INT WAIN LPAREN dcl COMMA dcl RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

- Once you have a table of frame pointer offsets, push the local variables to the stack at the appropriate locations.
- There are essentially two approaches to this. One is to ensure that you compute the offsets and push the variables in the same order.
    - Note that std::map and std::unordered_map in C++ store keys in an arbitrary order, so don't rely on iterating over the map to produce the correct order.
- You could also use the table to directly store variables at their offsets.
    - But make sure you update the stack pointer appropriately!

Implementing Code Generation

main → INT WAIN LPAREN dcl COMMA dcl RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

- Approach 1: Compute offsets and push variables in the same order.
  - Keep a counter of how many variables you have seen.
  - Look at the first dcl subtree. Store the pair (variable name, (counter – 2) * -4) in the offset table, increment the counter, and output code to push $1 to the stack.
    - It's (counter – 2) to account for the fact that there are 2 parameters of wain.
  - Look at the second dcl subtree. Repeat the previous step but with $2.
  - Now set the FP to divide parameters and non-parameters:
    `sub $29, $30, $4`
  - Traverse the dcls subtree and for each dcl you find, repeat the previous step but store the variable's initial value instead of $1 or $2.

Implementing Code Generation

main → INT WAIN LPAREN dcl COMMA dcl RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

- Approach 2: Use the table to store variables directly at their offsets.
  - This might be easier if you prefer to compute the offset table in a separate pass, instead of doing it at the same time as pushing the variables.
  - Set the frame pointer to $30 – 12 (so that it is one slot above the second parameter).
  - Loop over the offset table, and for each variable, output code that stores $1, $2, or the initial value at the correct offset from $29.
  - **Then, set $30 to the address of the highest-up thing you placed on the stack ($29 + most negative offset). This is extremely important.**
  - If you don't do this, then when you push things to the stack later, it will wipe out your local variables (since they will be above the stack pointer!!)

Implementing Code Generation

main → INT WAIN LPAREN dcl COMMA dcl RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

- The big picture:

1. Initialize constants (e.g. $4 = 4). [Done]
2. Set up the frame pointer and put variables on the stack. [Done]
3. Generate code for all statements in the "statements" subtree.
4. Generate code for the return "expr".
5. Generate code that cleans up the stack ("optional", but can be helpful for debugging) and returns (jr $31).

Implementing Code Generation

main $\rightarrow$ INT WAIN LPAREN dcl COMMA dcl RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

- Skipping ahead to step 5…

5. Generate code that cleans up the stack ("optional", but can be helpful for debugging) and returns (jr $31).

- It's "optional" in the sense that nothing bad will actually happen if you don't reset the stack (nor does Marmoset check for it).
  - It's sort of a "memory leak", but the OS can handle it after the program ends.
- But it can be useful as a "sanity check".

Implementing Code Generation

main → INT WAIN LPAREN dcl COMMA dcl RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

- Skipping ahead to step 5…

5. Generate code that cleans up the stack ("optional", but can be helpful for debugging) and returns (jr $31).

- At the end of the program, pop N times where N is the number of local variables you pushed onto the stack.
- If you used the stack correctly, the stack pointer $30 should be at its original address. If it's not, this might be a sign of a bug!

Implementing Code Generation

main → INT WAIN LPAREN dcl COMMA dcl RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

- How about step 4?

4. Generate code for the return "expr".

- Aside from rules related to pointers and procedures, we have mostly already covered this.
- For rules like this:

  expr → term   term → factor   factor → LPAREN expr RPAREN

- Just recurse on the relevant child node.

Implementing Code Generation

main → INT WAIN LPAREN dcl COMMA dcl RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

- How about step 4?

4. Generate code for the return "expr".

- Aside from rules related to pointers and procedures, we have mostly already covered this.
- For factor → ID, look up the ID in the frame pointer offset table, and generate code that loads from that offset into $3.
- For factor → NUM, put the numeric constant in $3 with lis.

Implementing Code Generation

main → INT WAIN LPAREN dcl COMMA dcl RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

- How about step 4?

4. Generate code for the return "expr".

- For binary operation rules, e.g. expr → expr PLUS term:
  - Recursively generate code that puts the left subexpression result in $3.
  - Generate code that pushes $3 to the stack.
  - Recursively generate code that puts the right subexpression result in $3.
  - Generate code that pops from the stack into $5.
  - Perform the operation using the values in $5 and $3, storing the result in $3.

Implementing Code Generation

main → INT WAIN LPAREN dcl COMMA dcl RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

- How about step 4?

4. Generate code for the return "expr".

- We still need to discuss expressions involving pointers (dereference, address-of, pointer arithmetic, memory allocation with new).
- Expressions can also include *procedure calls*. We need to discuss how non-wain procedures are implemented.

Implementing Code Generation

main → INT WAIN LPAREN dcl COMMA dcl RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

- We also still need to discuss step 3:

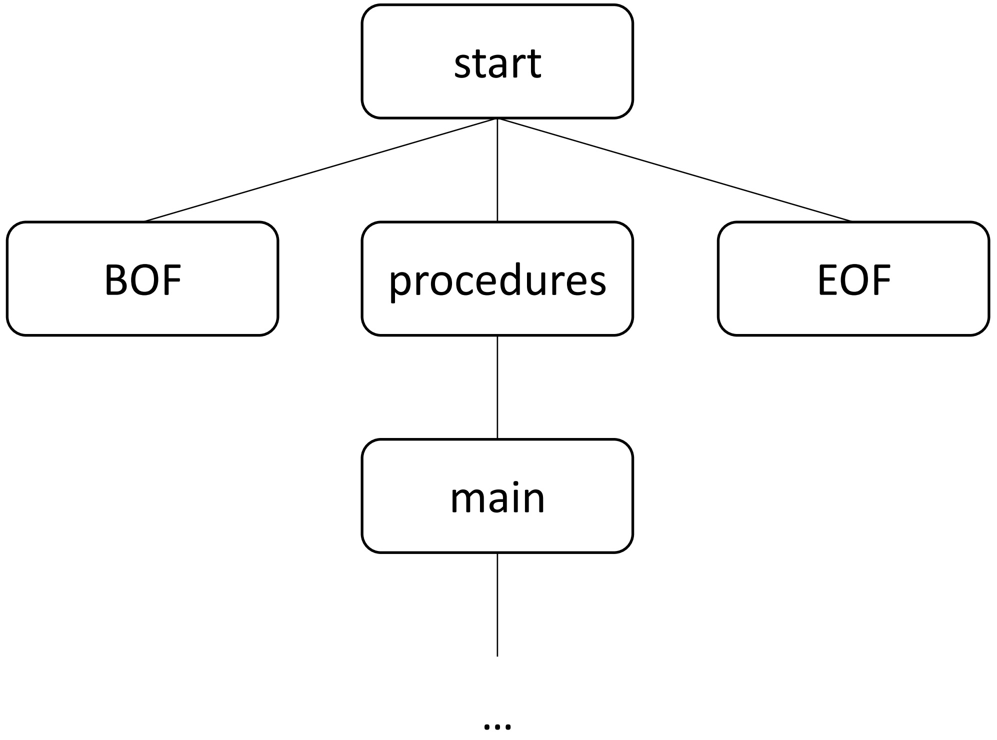
3. Generate code for all statements in the "statements" subtree.

- Statements include assignment statements, if statements, while loops, println statements, and delete (memory deallocation).
- If statements and while loops involve implementing *comparison tests.*
- For println and delete (and new) we will use *external libraries* and linking!

Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int wain(int a, int b) {
  int c = 482;
  int d = 2;
  return a+c/d;
}
```



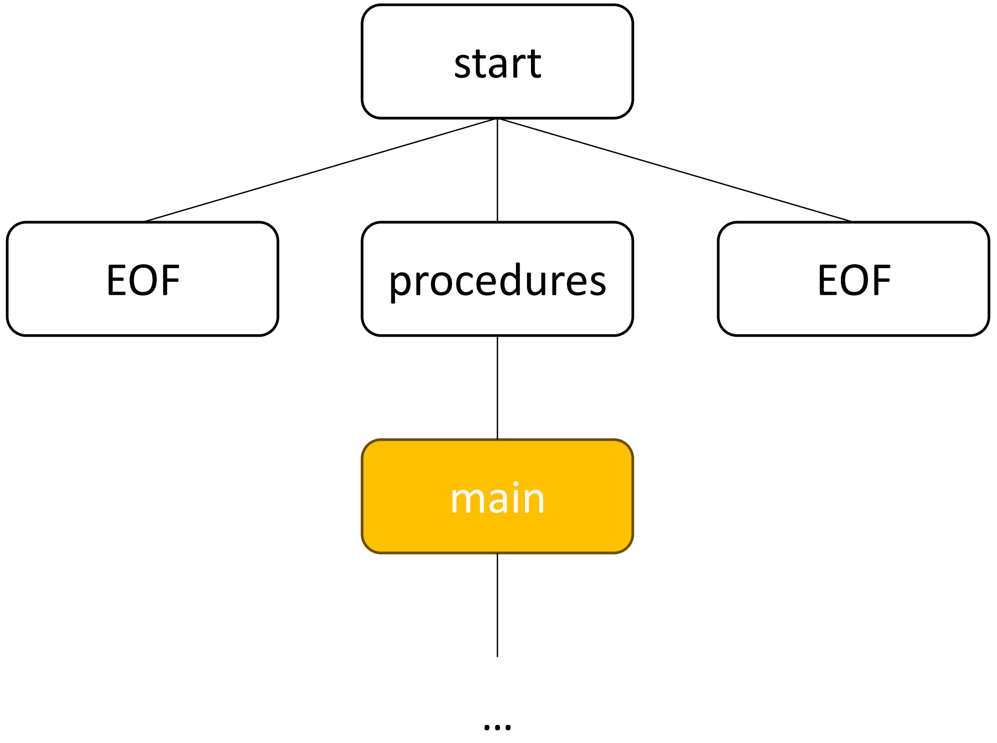
Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int wain(int a, int b) {
  int c = 482;
  int d = 2;
  return a+c/d;
}
```

```
lis $4
.word 4
```

Initialize constants

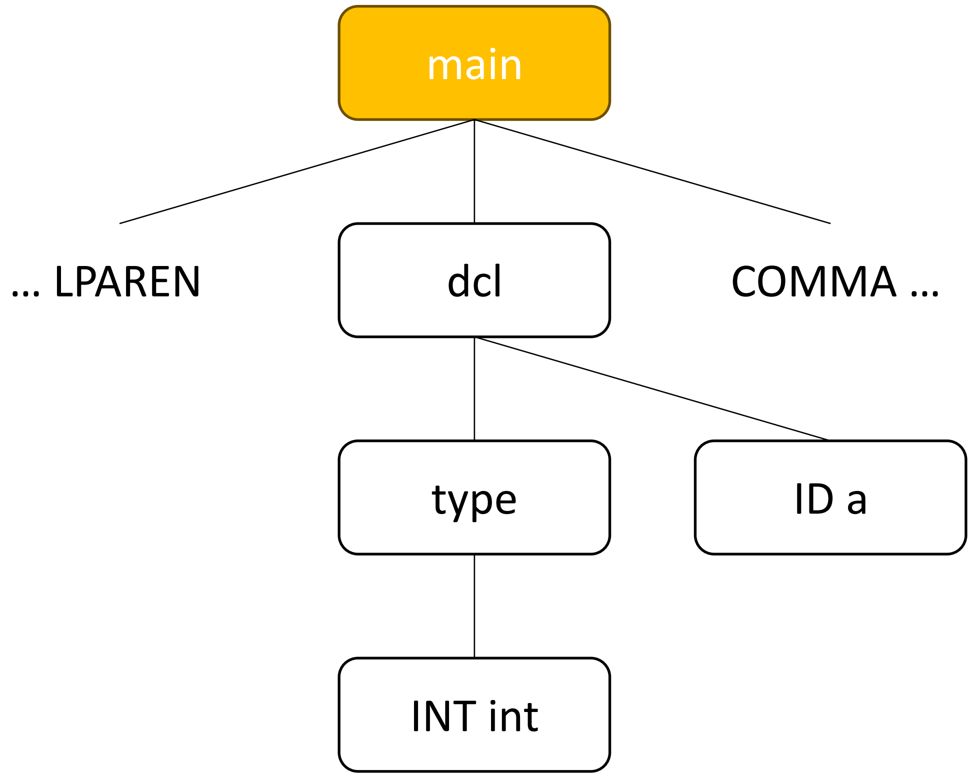


Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int wain(int a, int b) {
  int c = 482;
  int d = 2;
  return a+c/d;
}
```

```
lis $4
.word 4
```



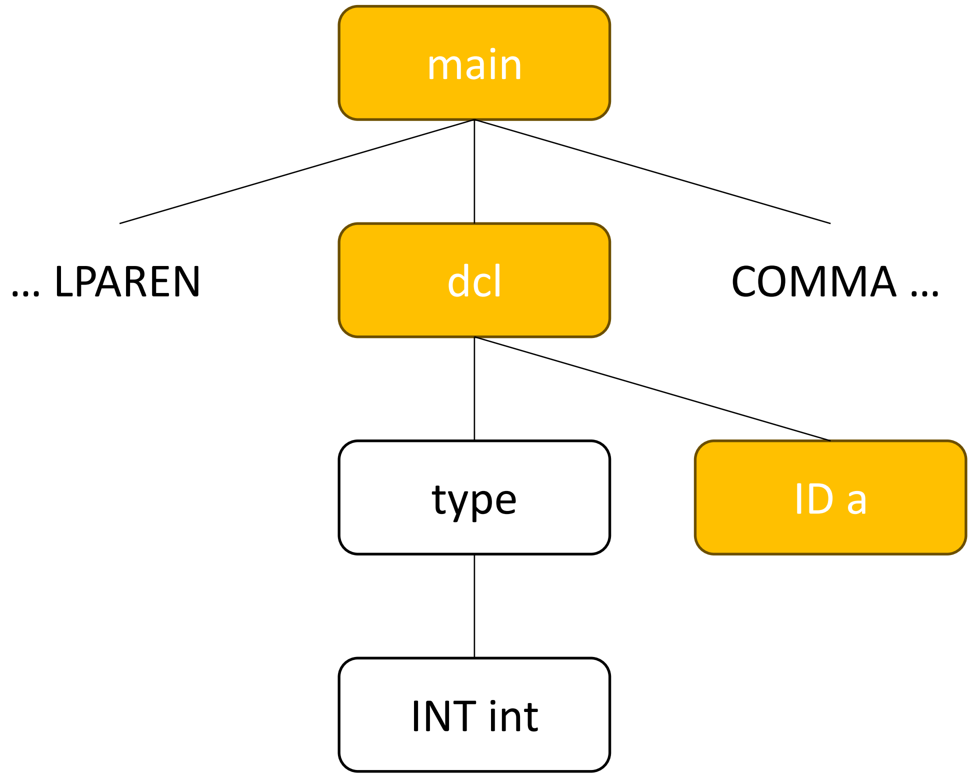
Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int wain(int a, int b) {
  int c = 482;
  int d = 2;
  return a+c/d;
}
```

```
lis $4
.word 4
sw $1, -4($30)
sub $30, $30, $4
```

Add (a, 8) to offset table
Store a = $1 on stack



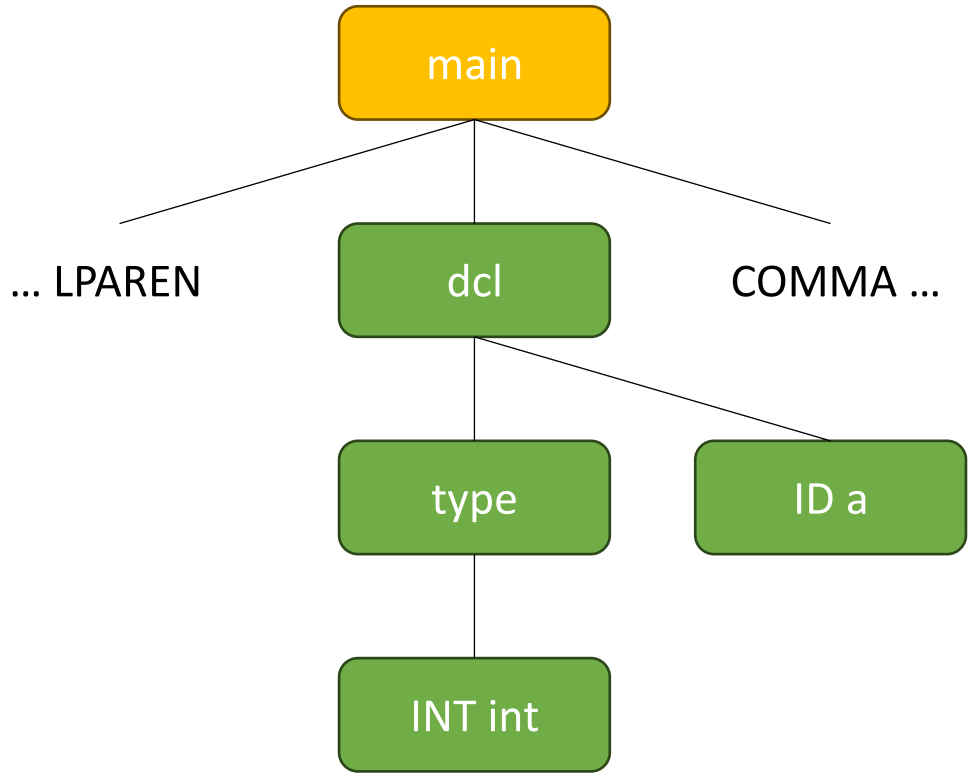
Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int wain(int a, int b) {
  int c = 482;
  int d = 2;
  return a+c/d;
}
```

```
lis $4
.word 4
sw $1, -4($30)
sub $30, $30, $4
```

Add (a, 8) to offset table
Store a = $1 on stack

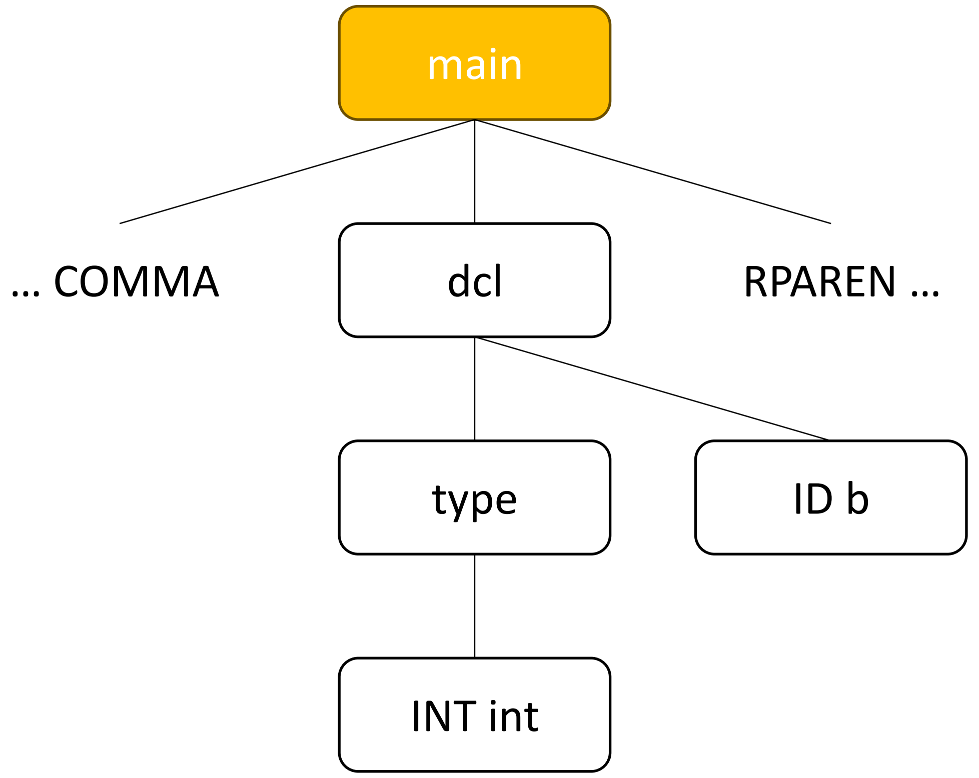


Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int wain(int a, int b) {
  int c = 482;
  int d = 2;
  return a+c/d;
}
```

```
lis $4
.word 4
sw $1, -4($30)
sub $30, $30, $4
```



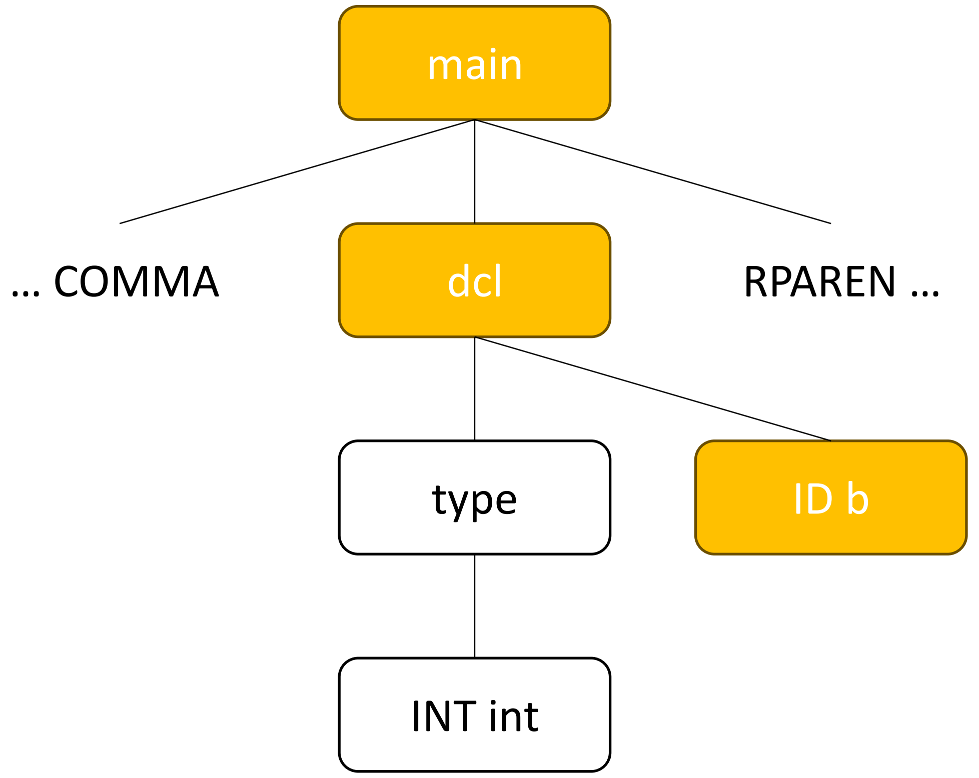
Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int wain(int a, int b) {
  int c = 482;
  int d = 2;
  return a+c/d;
}
```

```
lis $4
.word 4
sw $1, -4($30)
sub $30, $30, $4
sw $2, -4($30)
sub $30, $30, $4
```

Add (b, 4) to offset table
Store b = $2 on the stack



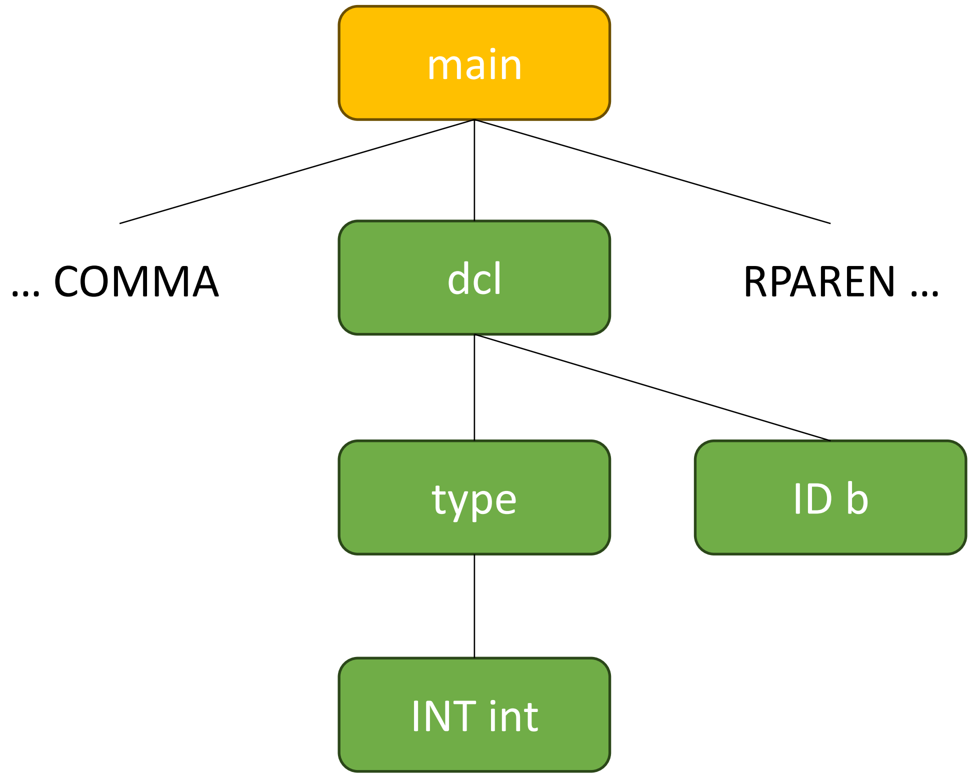
Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int wain(int a, int b) {
  int c = 482;
  int d = 2;
  return a+c/d;
}
```

```
lis $4
.word 4
sw $1, -4($30)
sub $30, $30, $4
sw $2, -4($30)
sub $30, $30, $4
```

Add (b, 4) to offset table
Store b = $2 on the stack



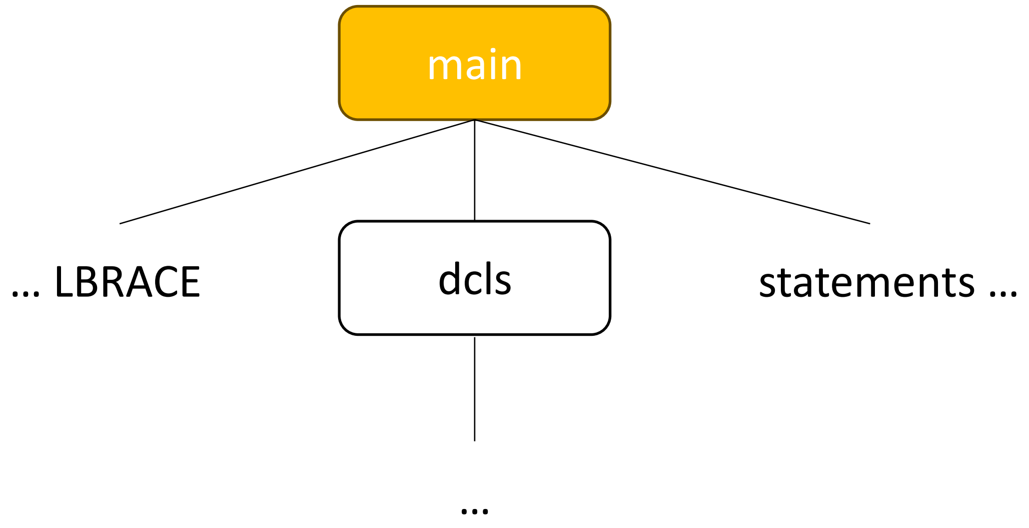
Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int wain(int a, int b) {
  int c = 482;
  int d = 2;
  return a+c/d;
}
```

```
lis $4
.word 4
sw $1, -4($30)
sub $30, $30, $4
sw $2, -4($30)
sub $30, $30, $4
sub $29, $30, $4
```

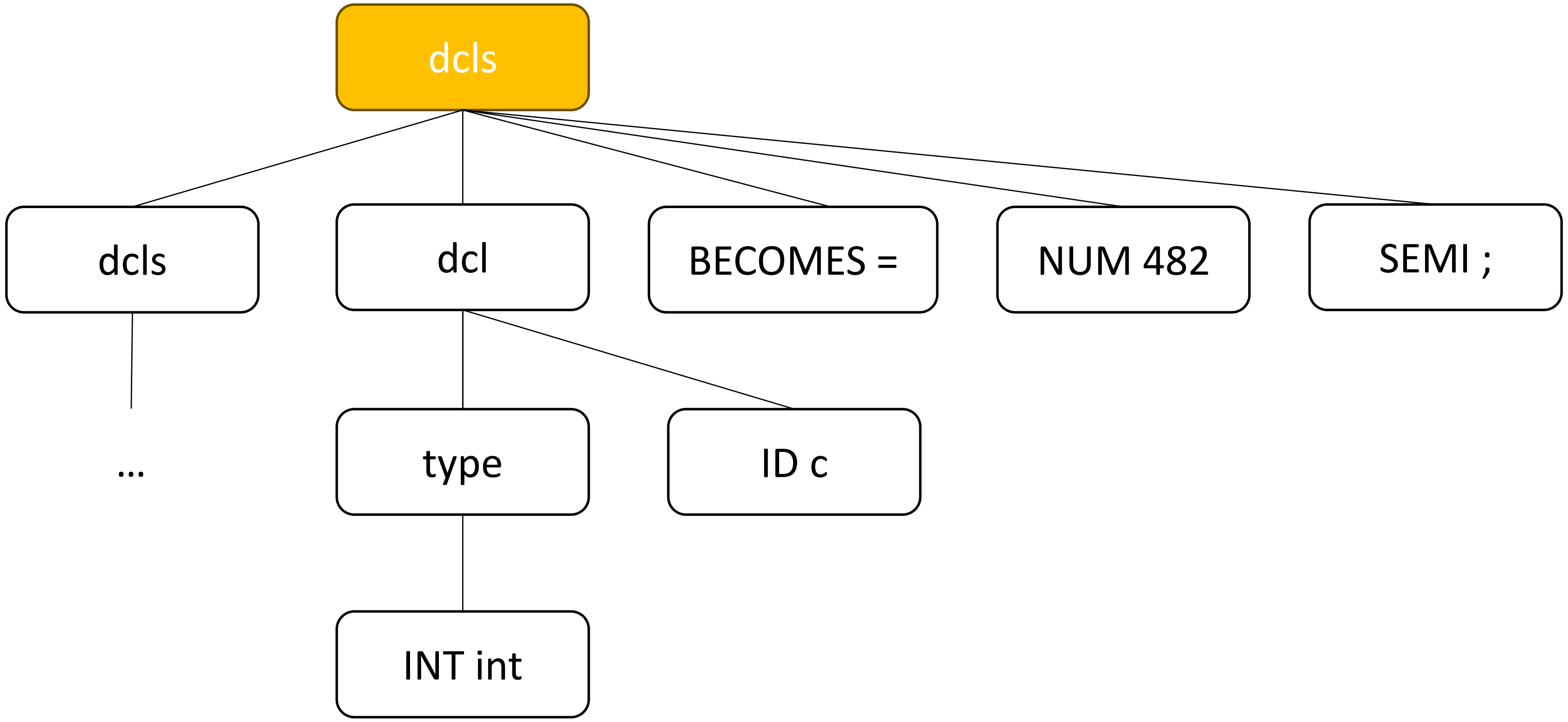
Before processing "dcls", set up the frame pointer



Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int c = 482;
int d = 2;
```

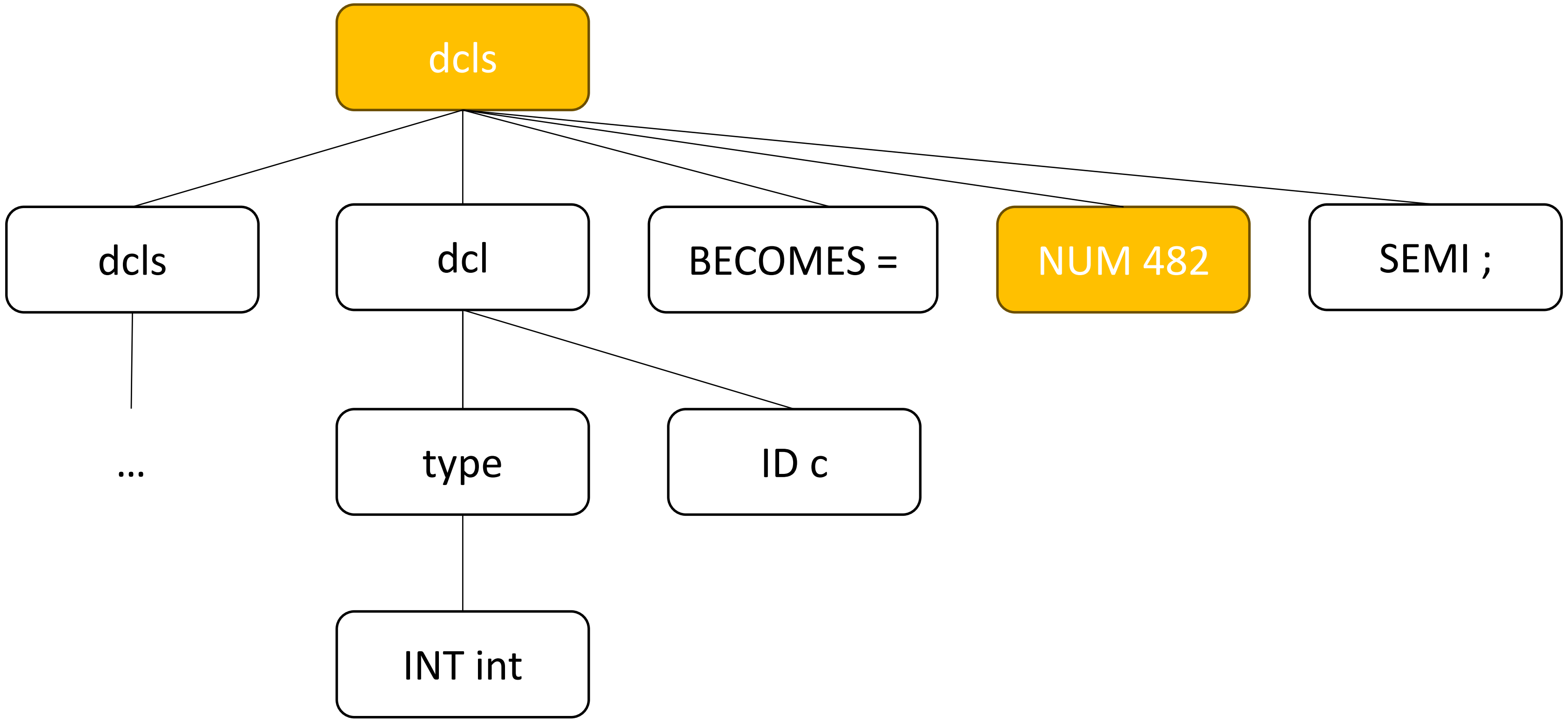


Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int c = 482;
int d = 2;
```

```
lis $3
.word 482
```

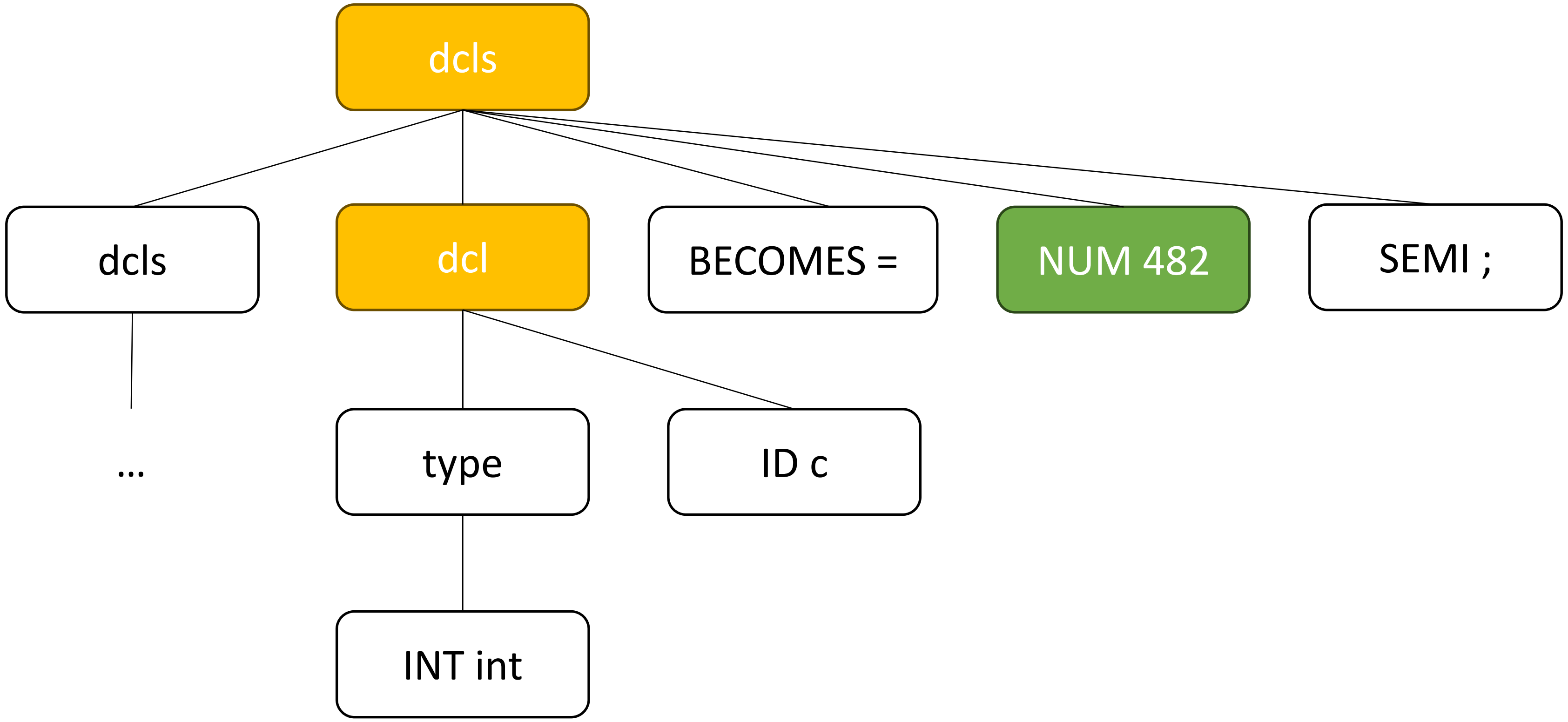


Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int c = 482;
int d = 2;
```

```
lis $3
.word 482
```



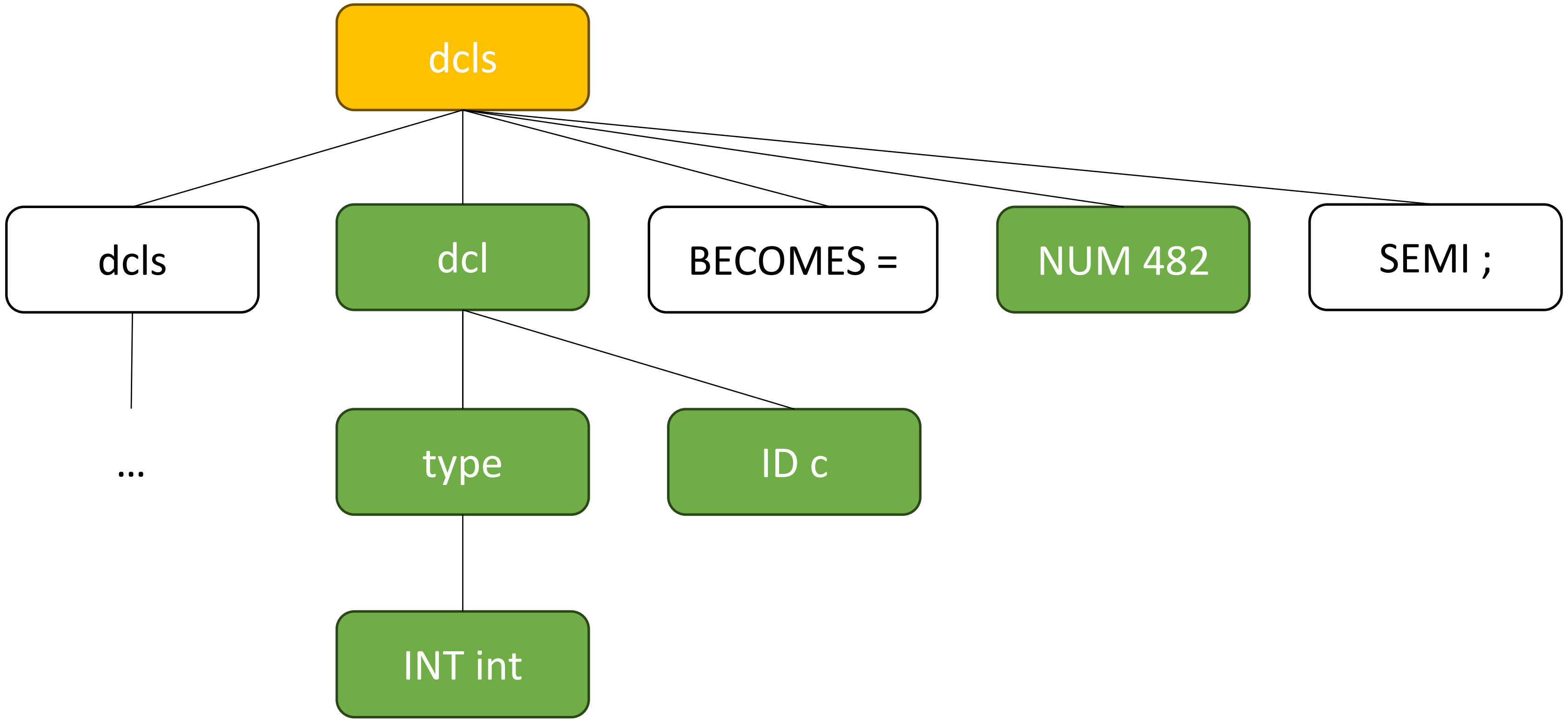
Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int c = 482;
int d = 2;
```

```
lis $3
.word 482
sw $3, -4($30)
sub $30, $30, $4
```

Also add (c, 0) to offset table.



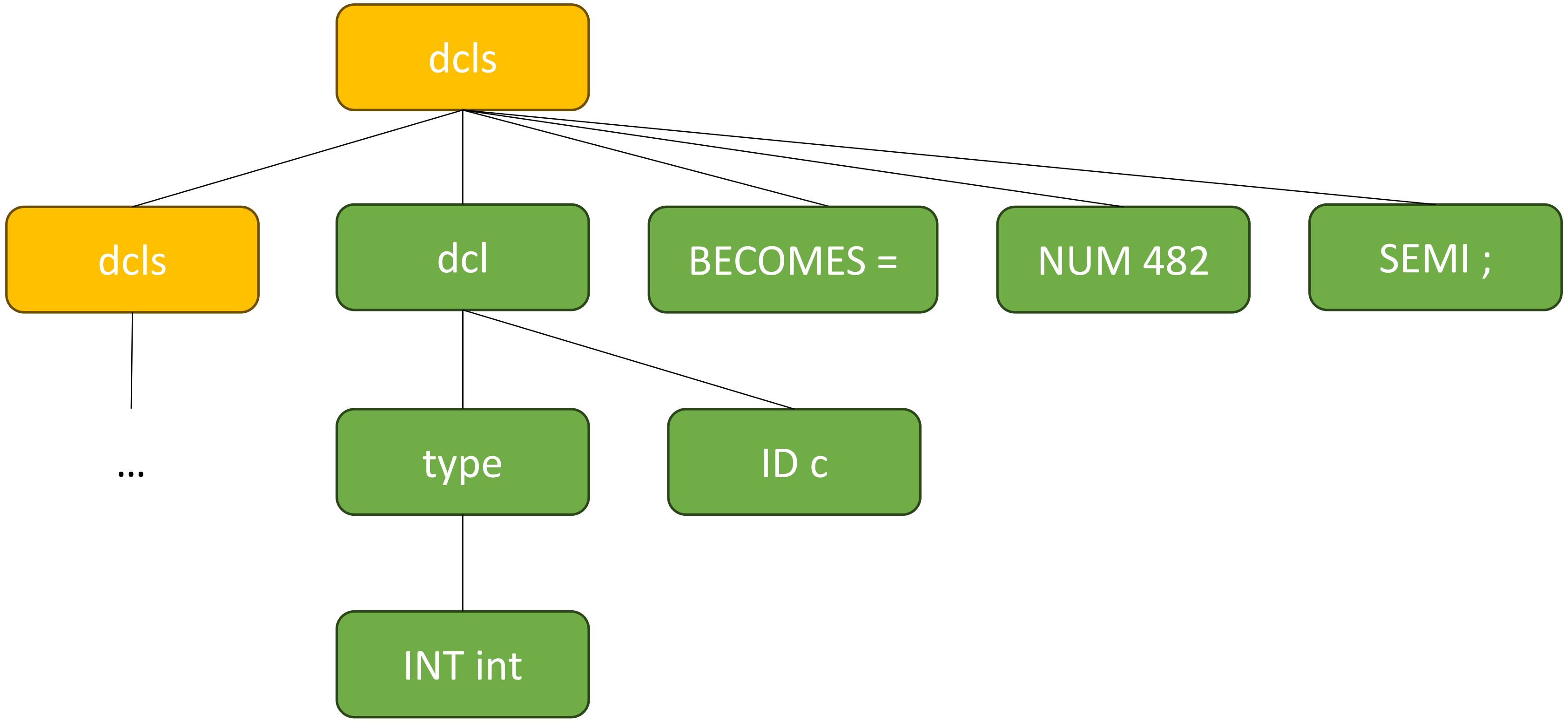
Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int c = 482;
int d = 2;
```

```
lis $3
.word 482
sw $3, -4($30)
sub $30, $30, $4
```

Done with this declaration.
Move on to the next.

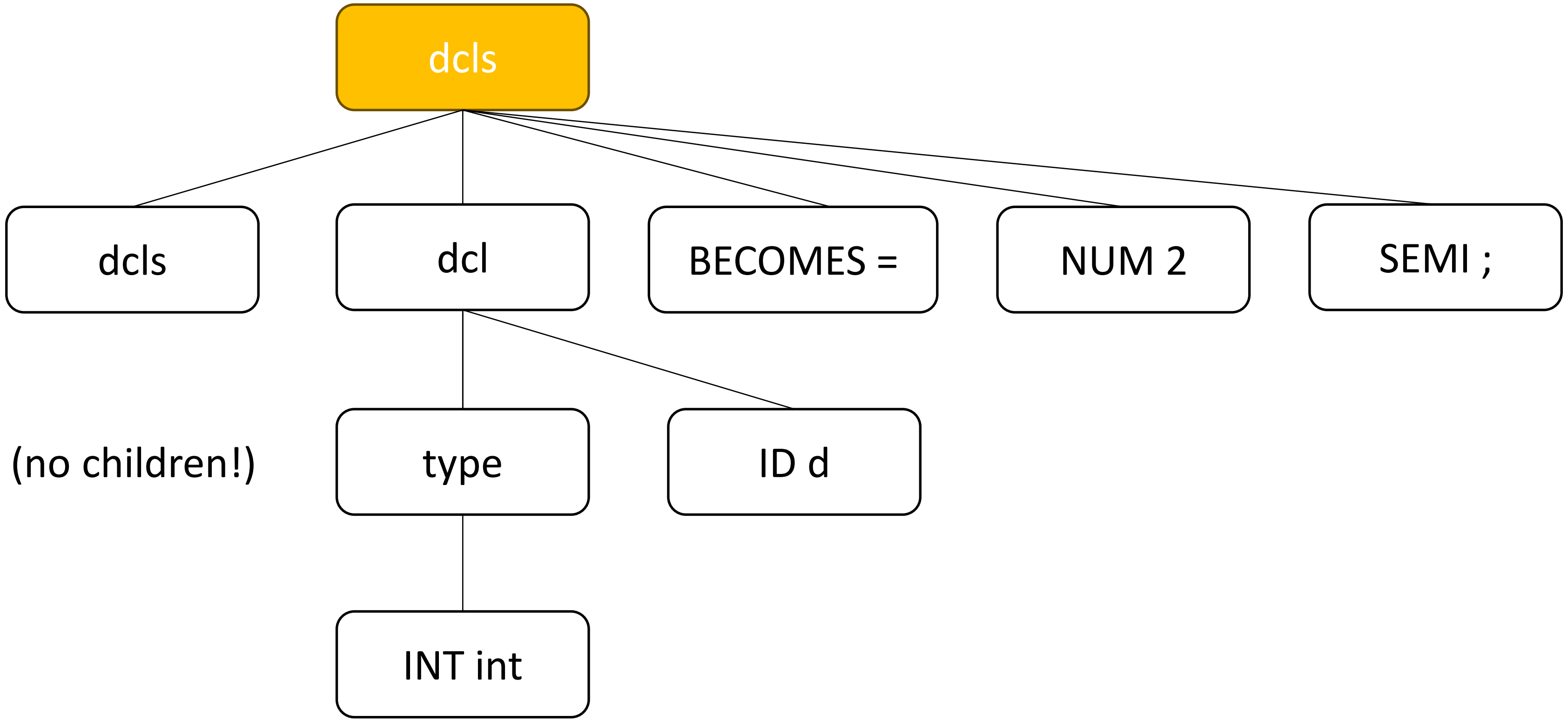


Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int c = 482;
int d = 2;
```

```
lis $3
.word 482
sw $3, -4($30)
sub $30, $30, $4
```

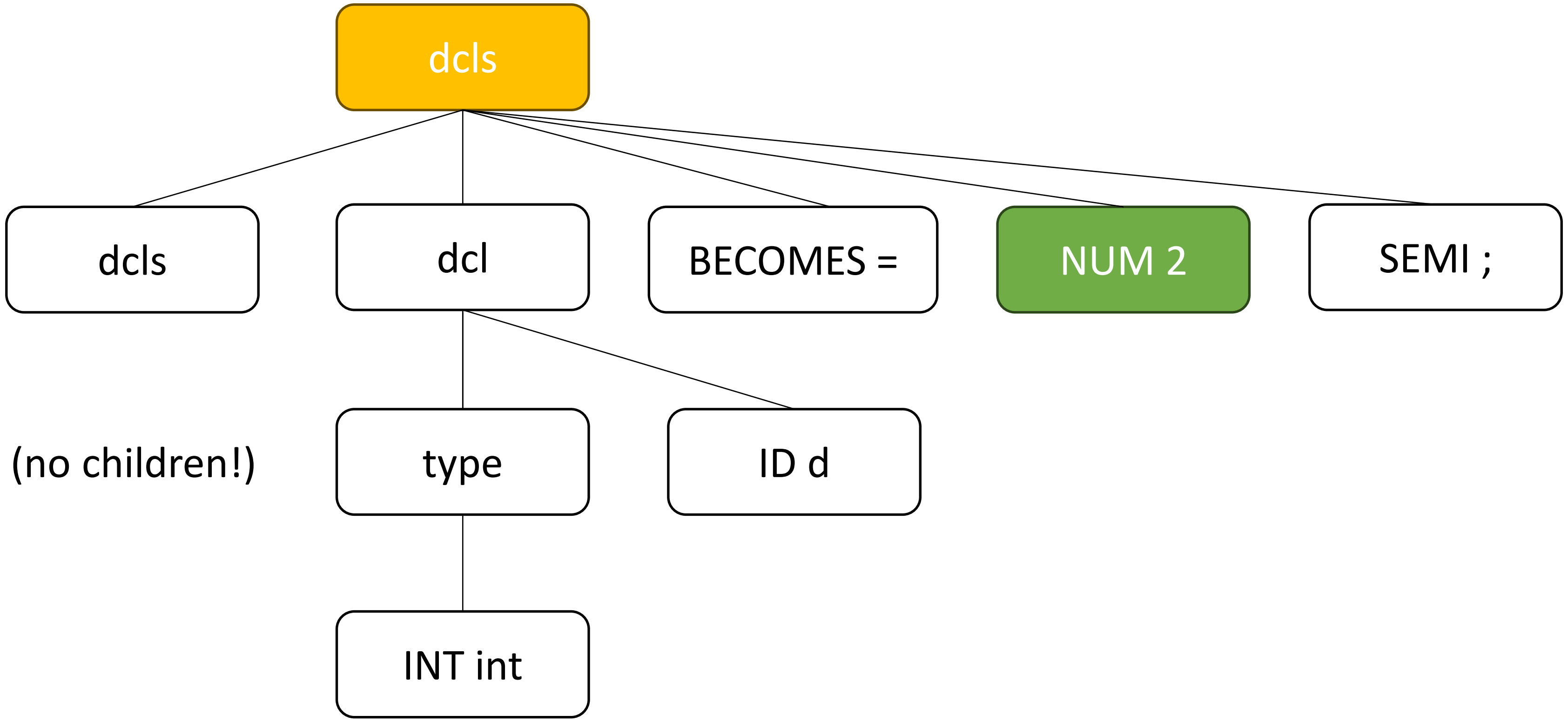


Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int c = 482;
int d = 2;
```

```
lis $3
.word 482
sw $3, -4($30)
sub $30, $30, $4
lis $3
.word 2
```



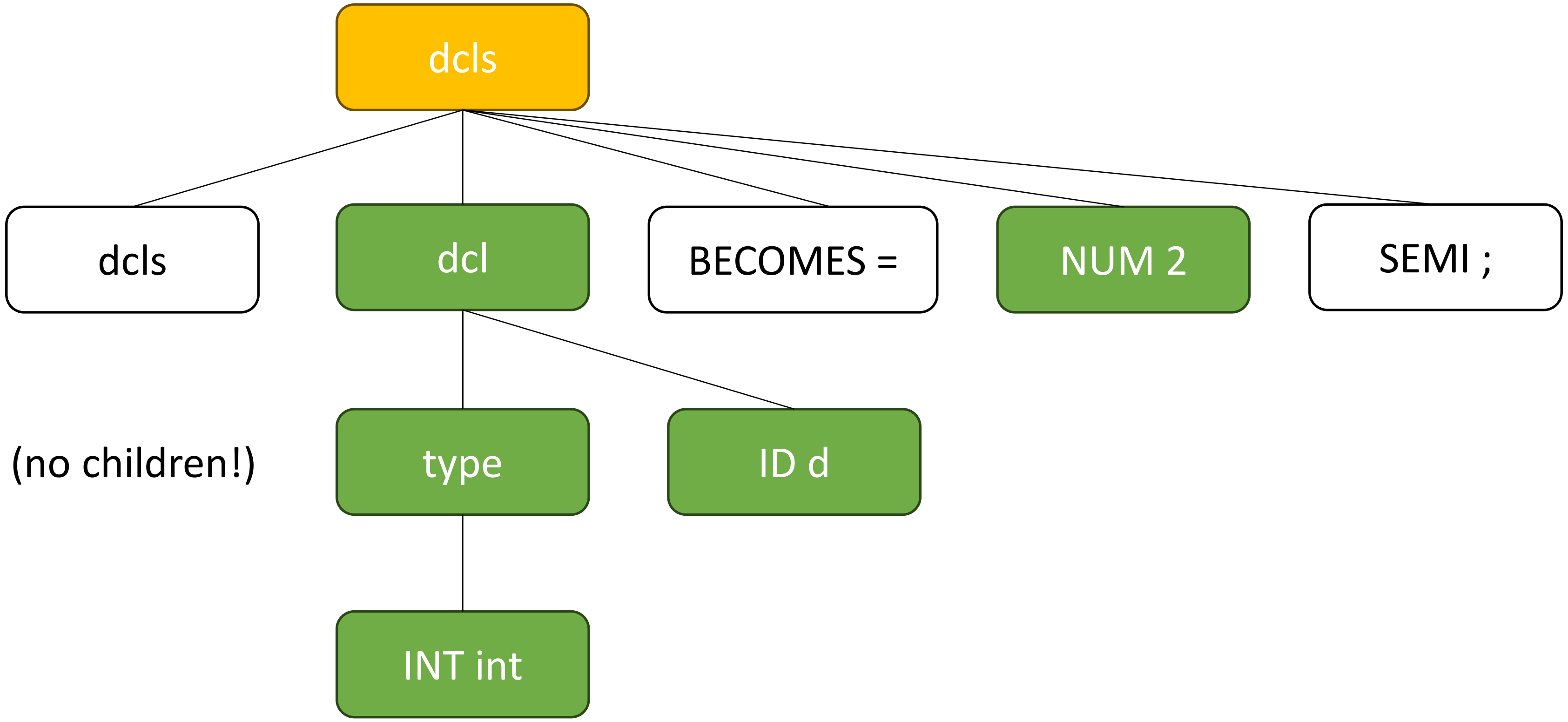
Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int c = 482;
int d = 2;
```

```
lis $3
.word 482
sw $3, -4($30)
sub $30, $30, $4
lis $3
.word 2
sw $3, -4($30)
sub $30, $30, $4
```

Also add (d, -4) to offset table.



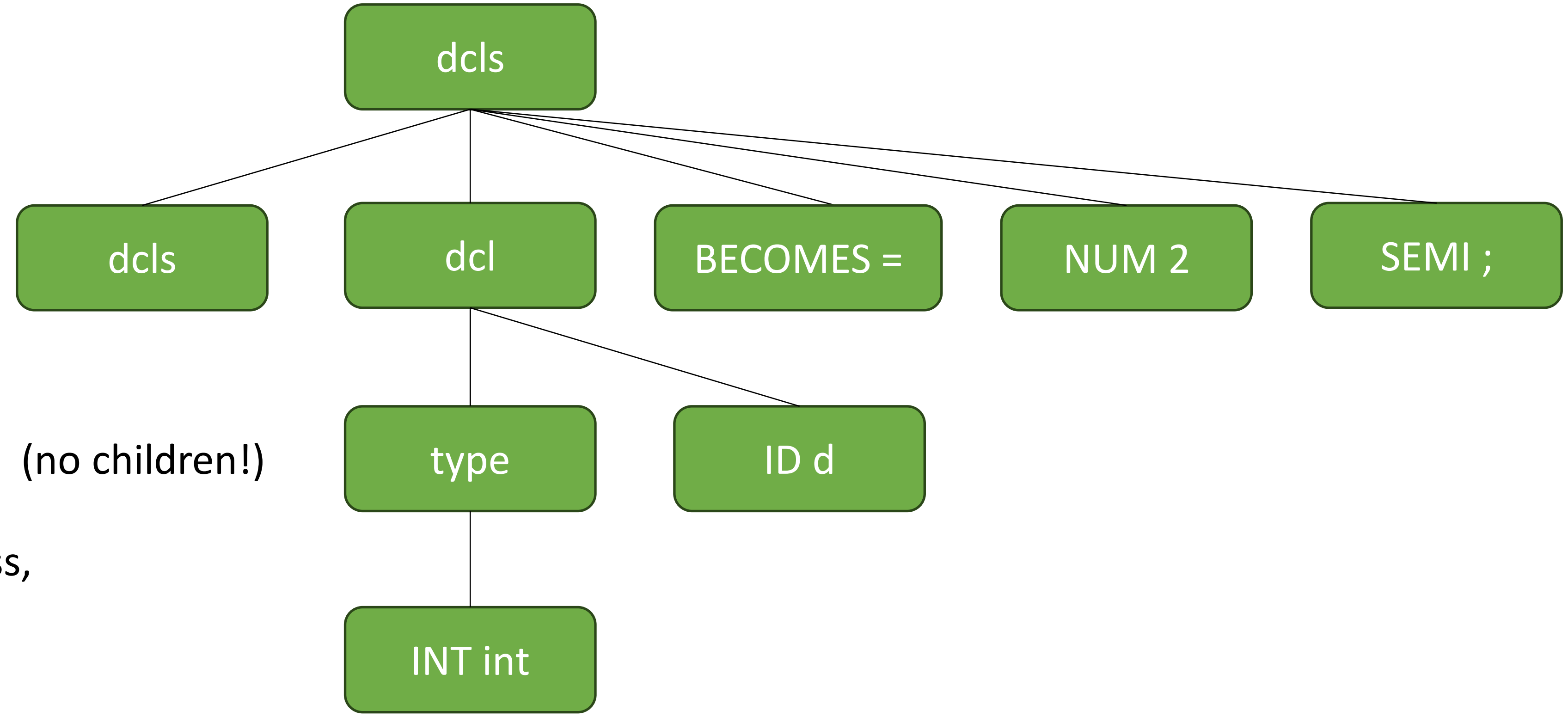
Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int c = 482;
int d = 2;
```

```
lis $3
.word 482
sw $3, -4($30)
sub $30, $30, $4
lis $3
.word 2
sw $3, -4($30)
sub $30, $30, $4
```

No more declarations to process, so we are fully done.



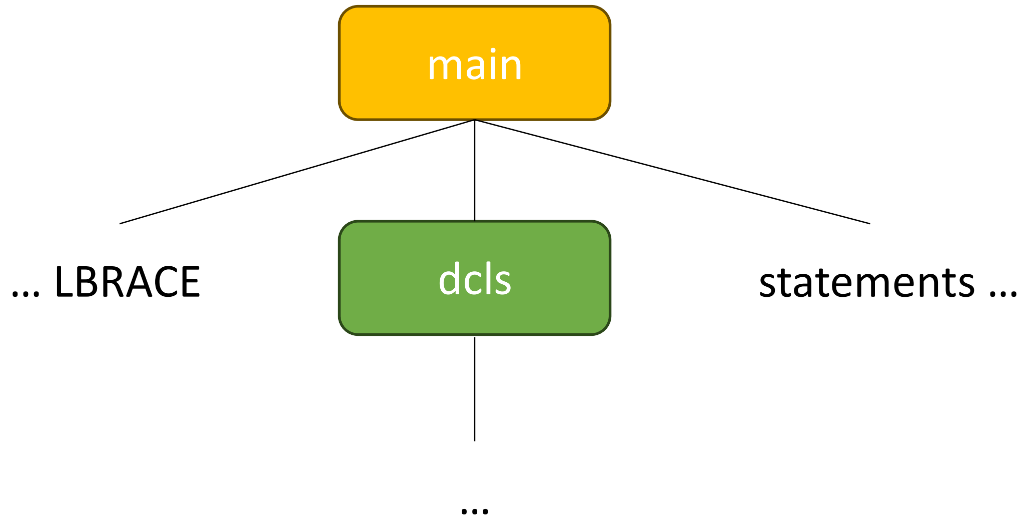
Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int wain(int a, int b) {
  int c = 482;
  int d = 2;
  return a+c/d;
}
```

```
lis $4
.word 4
sw $1, -4($30)
sub $30, $30, $4
sw $2, -4($30)
sub $30, $30, $4
sub $29, $30, $4
```

```
lis $3
.word 482
sw $3, -4($30)
sub $30, $30, $4
lis $3
.word 2
sw $3, -4($30)
sub $30, $30, $4
```

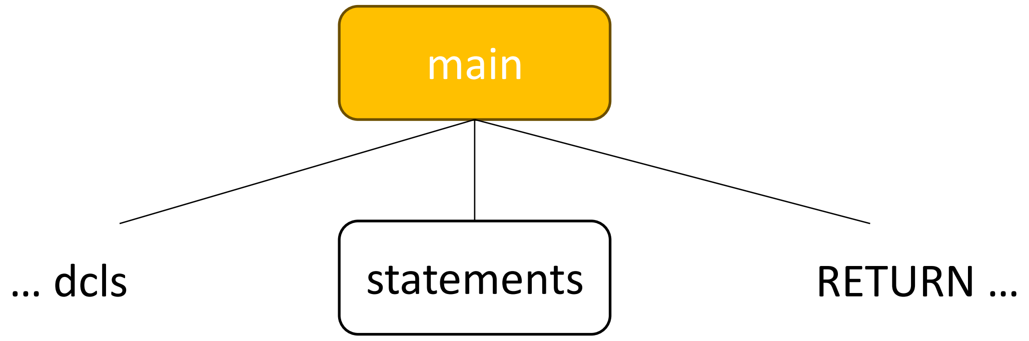


Done with the "prologue" (setting up constants, frame pointer, and storing variables on the stack.)

Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int wain(int a, int b) {
  int c = 482;
  int d = 2;
  return a+c/d;
}
```

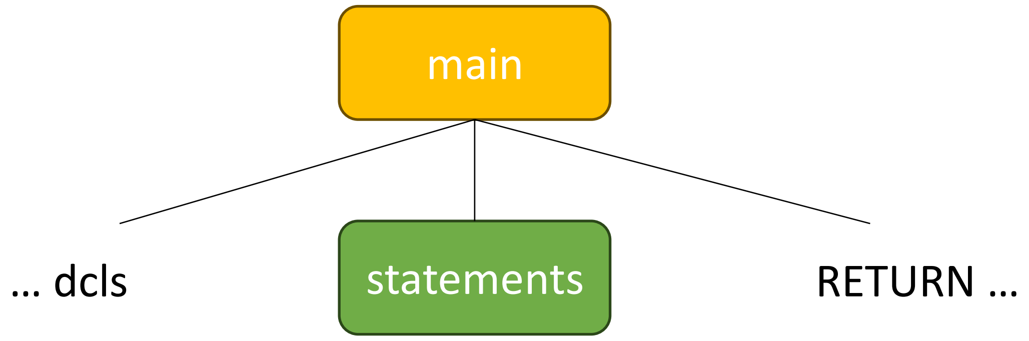


Time for the procedure body (statements and return expression).

Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int wain(int a, int b) {
  int c = 482;
  int d = 2;
  return a+c/d;
}
```



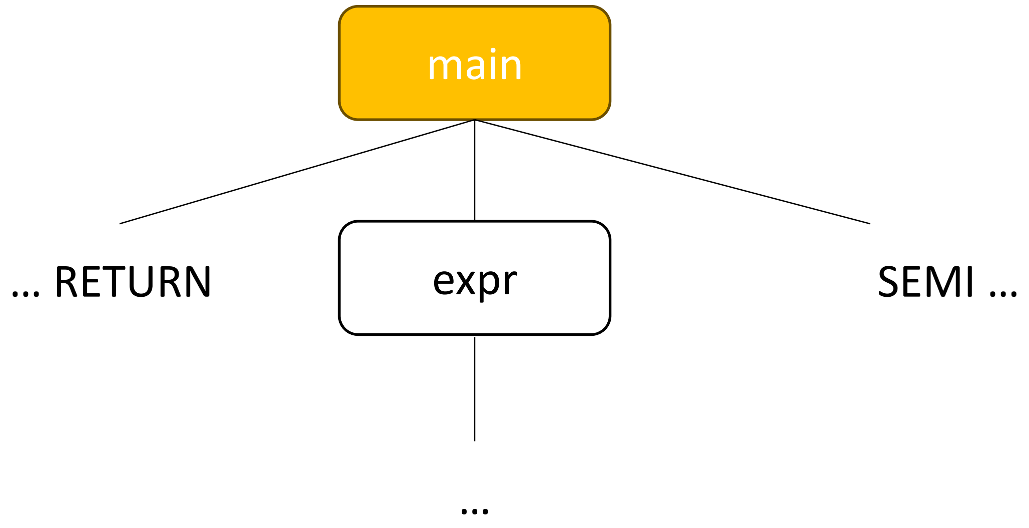
Time for the procedure body (statements and return expression).

This procedure has no statements.

Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int wain(int a, int b) {
  int c = 482;
  int d = 2;
  return a+c/d;
}
```

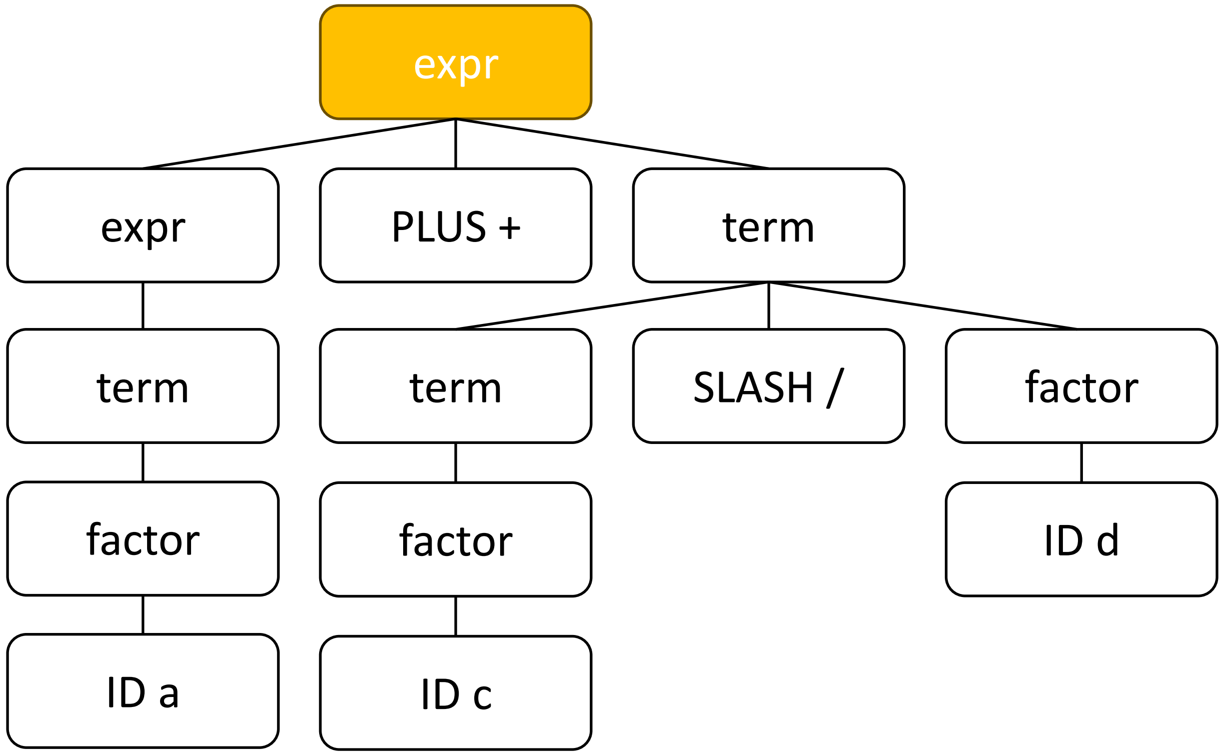


Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
a+c/d
```

Variable	Offset
a	8
b	4
c	0
d	-4



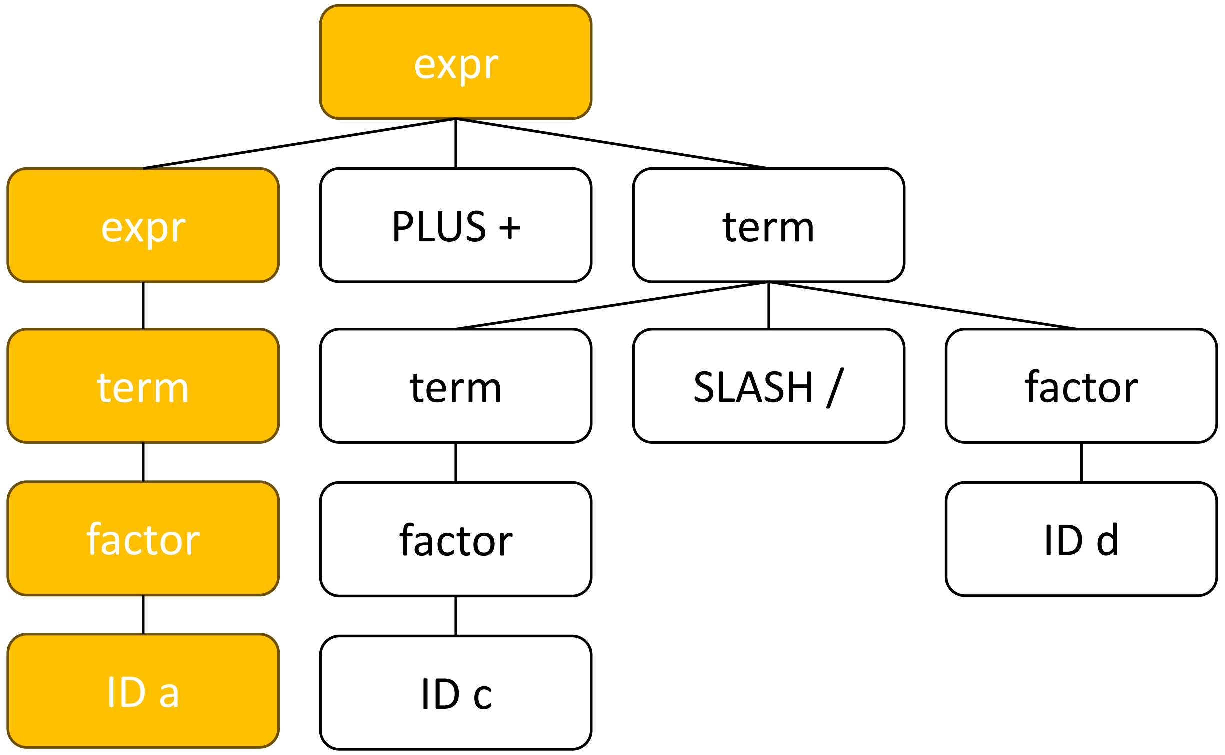
Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
a+c/d

lw $3, 8($29)
```

Variable	Offset
a	8
b	4
c	0
d	-4



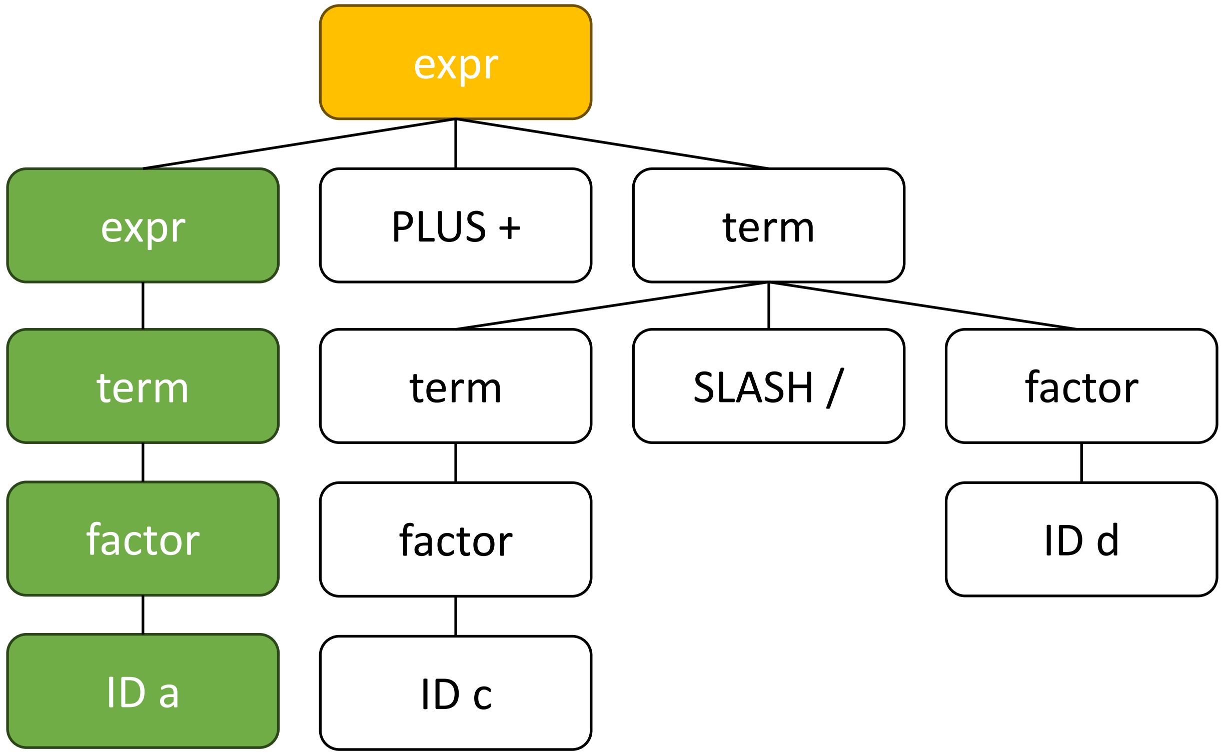
Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
a+c/d
lw $3, 8($29)
```

Load value of a into $3.

Variable	Offset
a	8
b	4
c	0
d	-4



Code Generation: Example

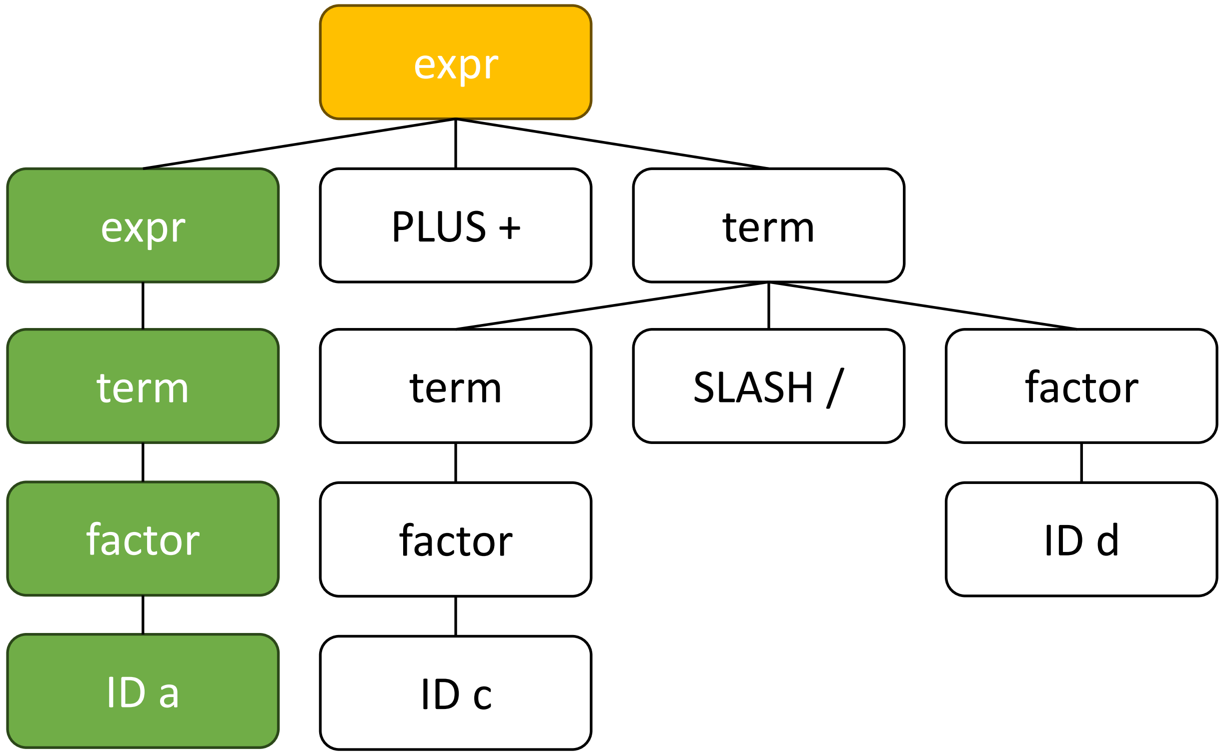
- Before we proceed with further discussion, let's look at an example of code generation in action.

```
a+c/d
```

```
lw $3, 8($29)
sw $3, -4($30)
sub $30, $30, $4
```

Before examining the right subexpression, we push the result of the left subexpression to the stack.

Variable	Offset
a	8
b	4
c	0
d	-4



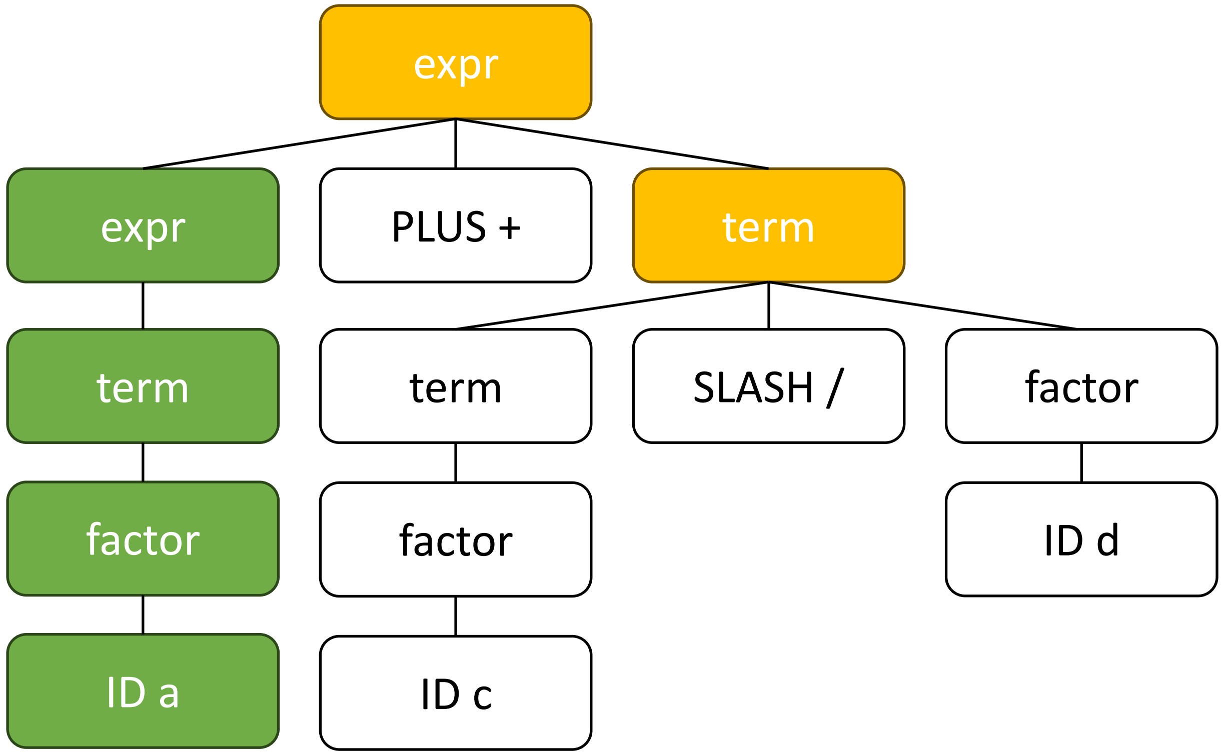
Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
a+c/d

lw $3, 8($29)
sw $3, -4($30)
sub $30, $30, $4
```

Variable	Offset
a	8
b	4
c	0
d	-4



Code Generation: Example

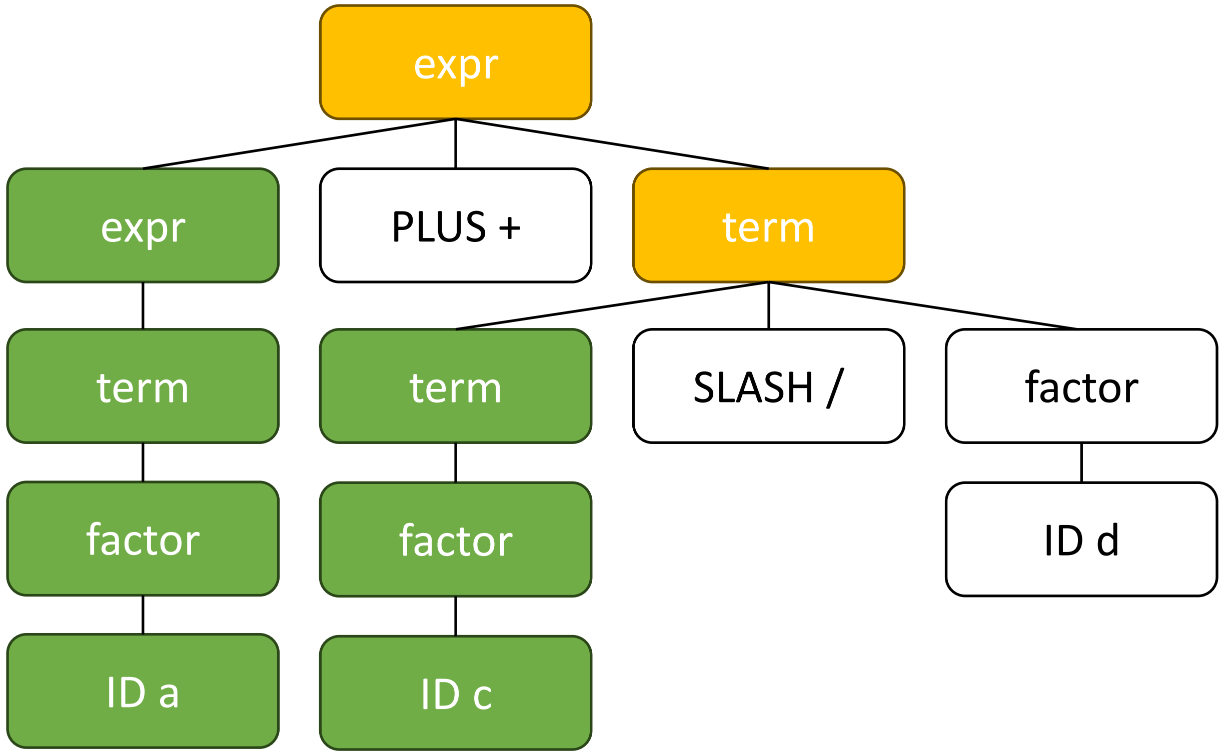
- Before we proceed with further discussion, let's look at an example of code generation in action.

```
a+c/d
```

```
lw $3, 8($29)
sw $3, -4($30)
sub $30, $30, $4
lw $3, 0($29)
```

Load value of c into $3.

Variable	Offset
a	8
b	4
c	0
d	-4



Code Generation: Example

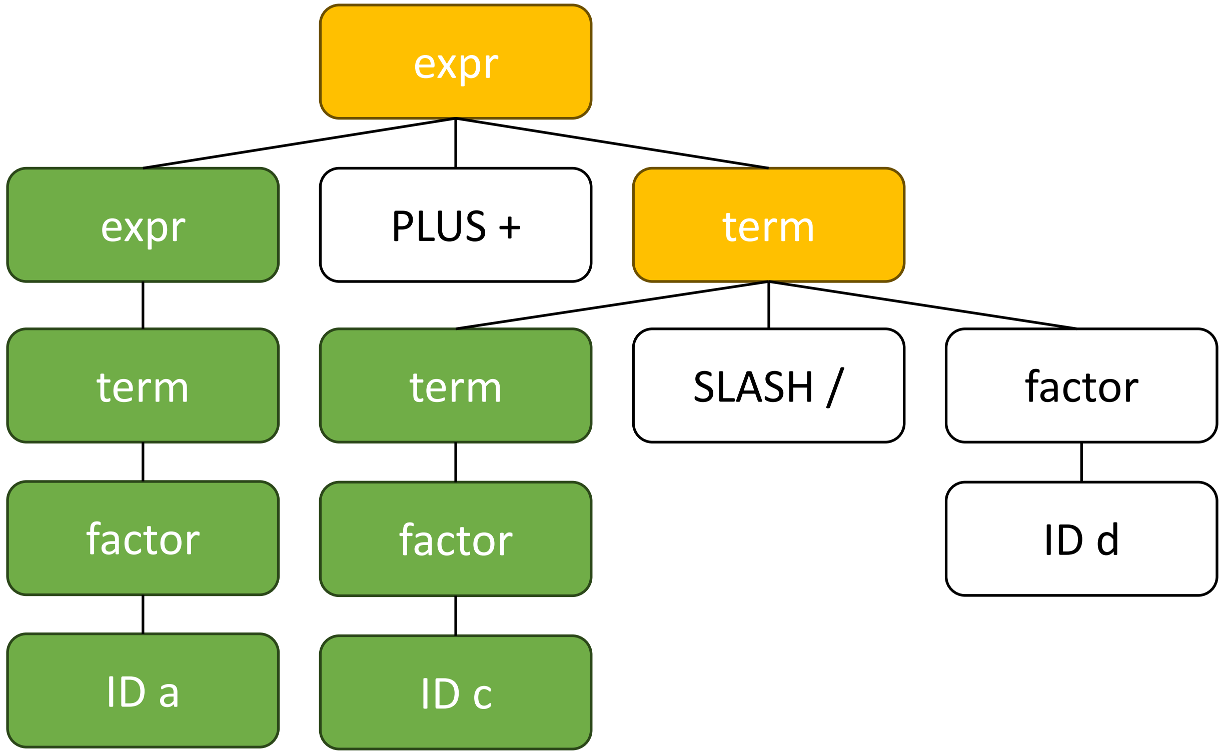
- Before we proceed with further discussion, let's look at an example of code generation in action.

```
a+c/d

lw $3, 8($29)
sw $3, -4($30)
sub $30, $30, $4
lw $3, 0($29)
sw $3, -4($30)
sub $30, $30, $4
```

Variable	Offset
a	8
b	4
c	0
d	-4

Push the result of the left subexpression onto the stack before looking at the right.



Code Generation: Example

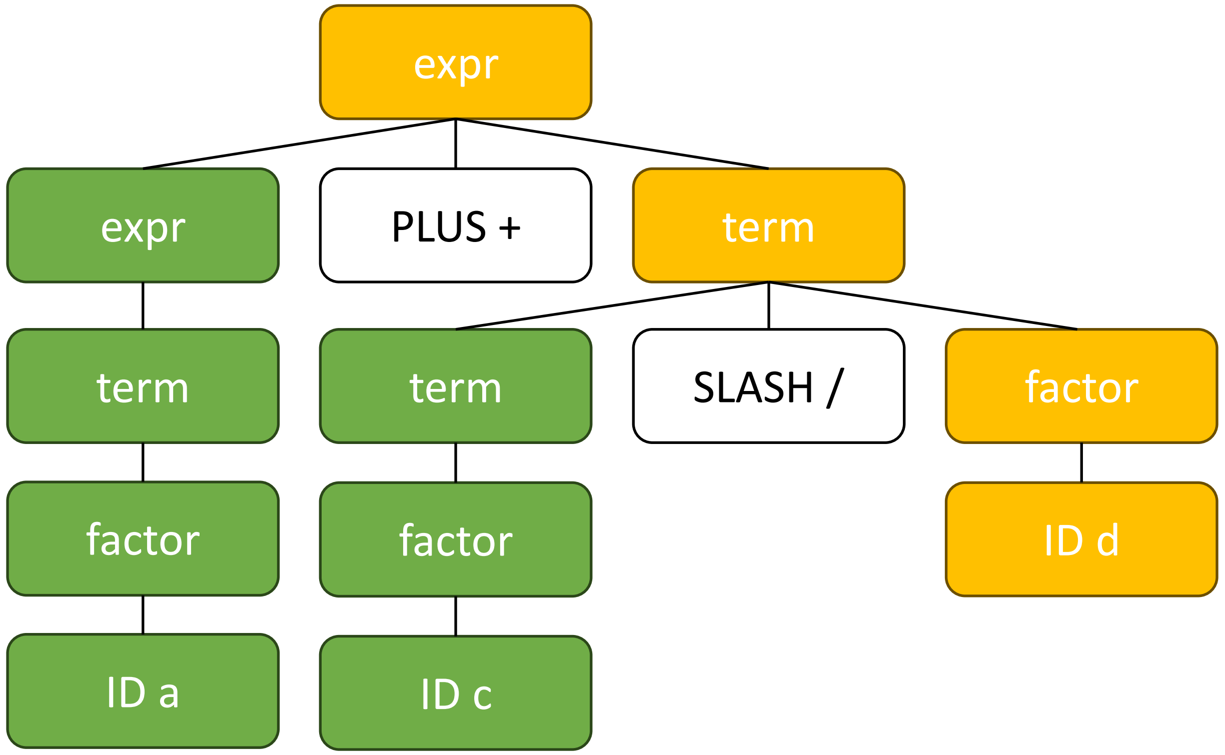
- Before we proceed with further discussion, let's look at an example of code generation in action.

```
a+c/d
```

```
lw $3, 8($29)
sw $3, -4($30)
sub $30, $30, $4
lw $3, 0($29)
sw $3, -4($30)
sub $30, $30, $4
lw $3, -4($29)
```

Load the value of d into $3.

Variable	Offset
a	8
b	4
c	0
d	-4



Code Generation: Example

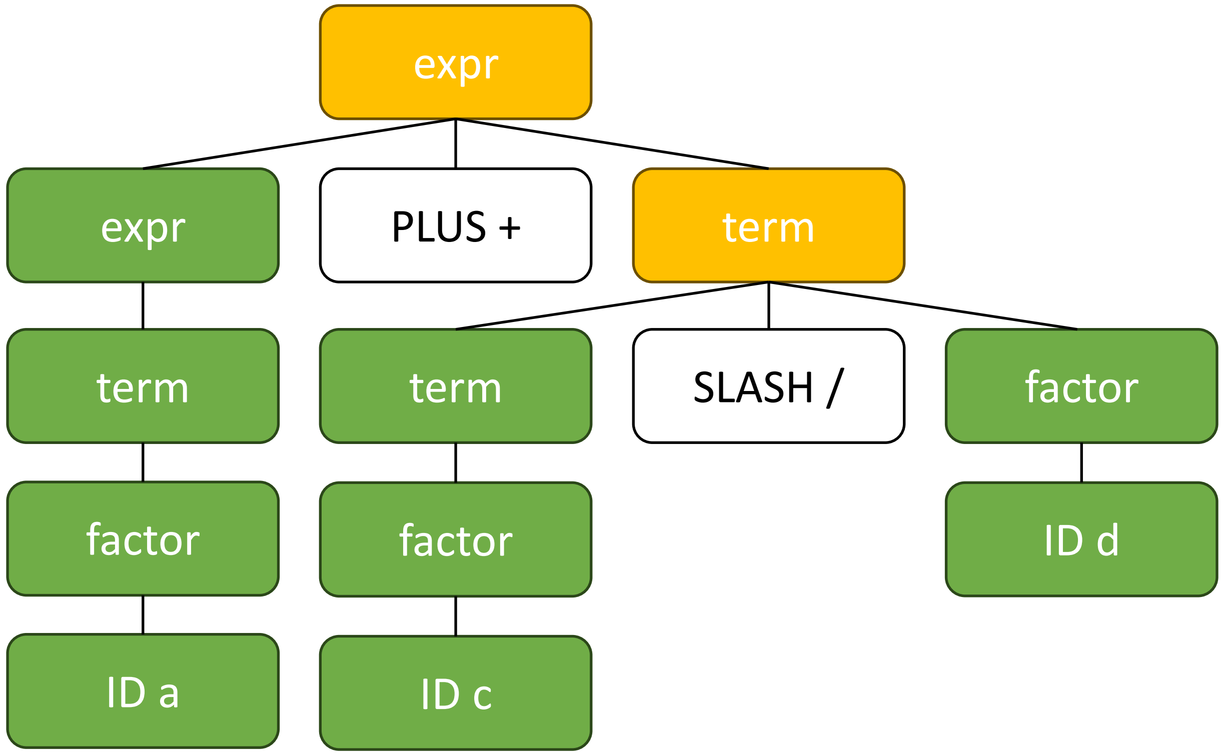
- Before we proceed with further discussion, let's look at an example of code generation in action.

```
a+c/d

lw $3, 8($29)
sw $3, -4($30)
sub $30, $30, $4
lw $3, 0($29)
sw $3, -4($30)
sub $30, $30, $4
lw $3, -4($29)
```

Variable	Offset
a	8
b	4
c	0
d	-4

The left and right subexpressions are both computed now.



Code Generation: Example

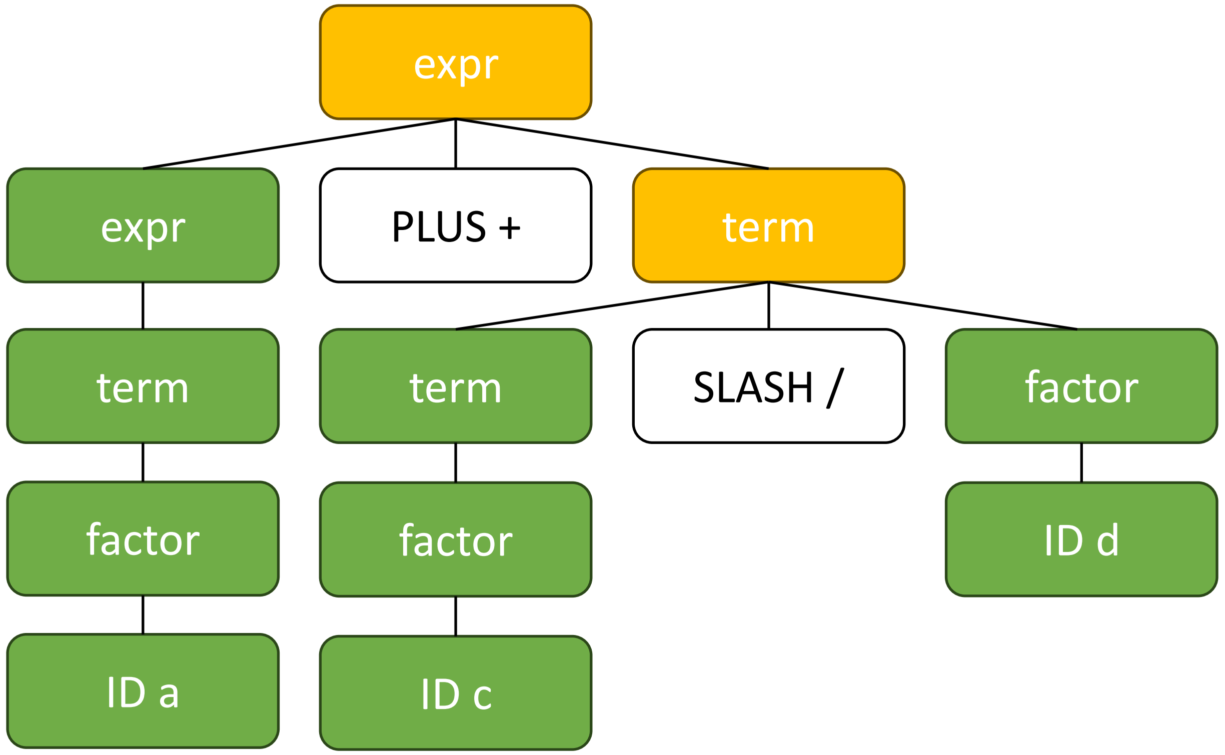
- Before we proceed with further discussion, let's look at an example of code generation in action.

```
a+c/d

lw $3, 8($29)
sw $3, -4($30)
sub $30, $30, $4
lw $3, 0($29)
sw $3, -4($30)
sub $30, $30, $4
lw $3, -4($29)
add $30, $30, $4
lw $5, -4($30)
```

Before looking at the operation, pop the left result off the stack into $5.

Variable	Offset
a	8
b	4
c	0
d	-4



Code Generation: Example

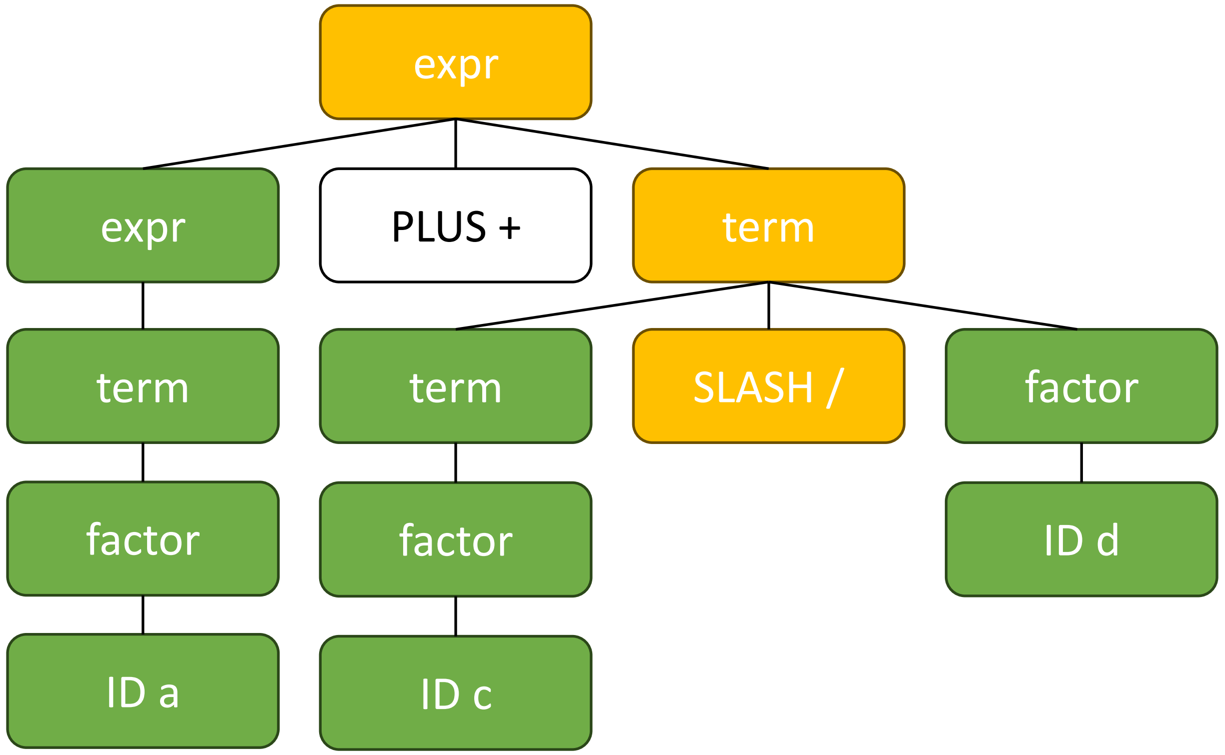
- Before we proceed with further discussion, let's look at an example of code generation in action.

```
a+c/d

lw $3, 8($29)
sw $3, -4($30)
sub $30, $30, $4
lw $3, 0($29)
sw $3, -4($30)
sub $30, $30, $4
lw $3, -4($29)
add $30, $30, $4
lw $5, -4($30)
div $5, $3
mflo $3
```

Variable	Offset
a	8
b	4
c	0
d	-4

Do the operation.



Code Generation: Example

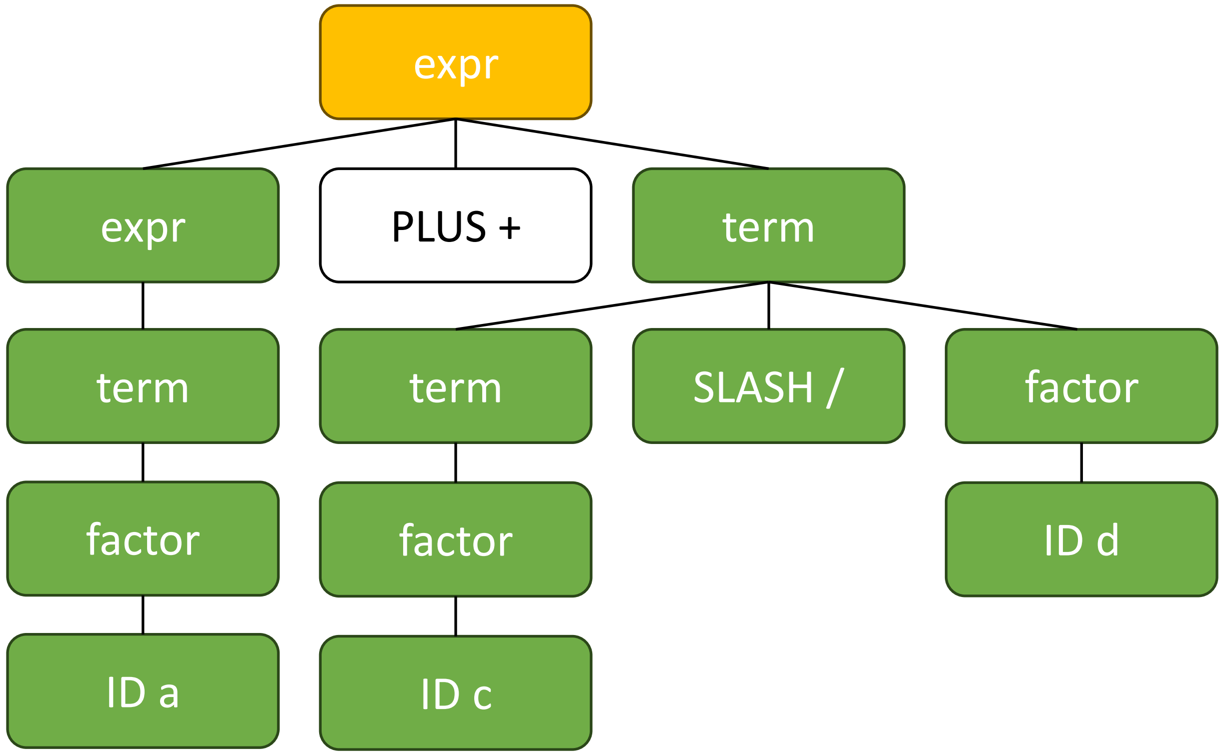
- Before we proceed with further discussion, let's look at an example of code generation in action.

```
a+c/d

lw $3, 8($29)
sw $3, -4($30)
sub $30, $30, $4
lw $3, 0($29)
sw $3, -4($30)
sub $30, $30, $4
lw $3, -4($29)
add $30, $30, $4
lw $5, -4($30)
div $5, $3
mflo $3
```

Variable	Offset
a	8
b	4
c	0
d	-4

Now repeat for the PLUS.



Code Generation: Example

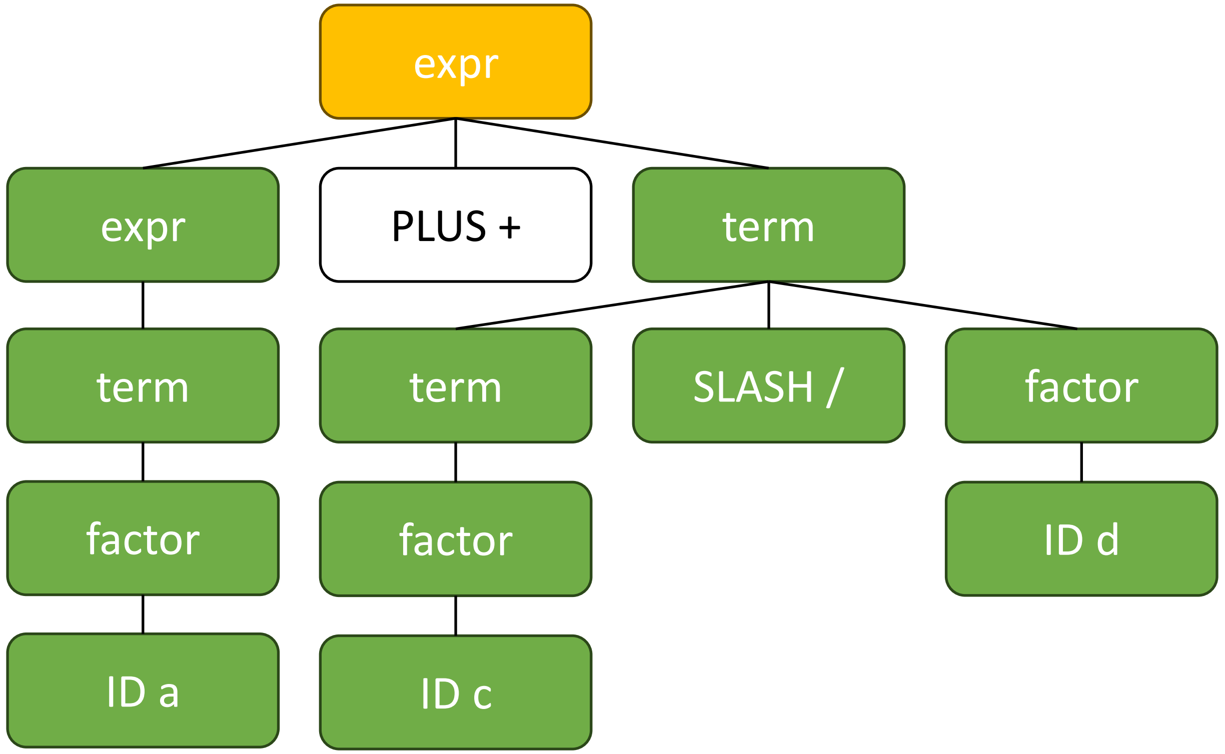
- Before we proceed with further discussion, let's look at an example of code generation in action.

```
a+c/d

lw $3, 8($29)
sw $3, -4($30)
sub $30, $30, $4
lw $3, 0($29)
sw $3, -4($30)
sub $30, $30, $4
lw $3, -4($29)
add $30, $30, $4
lw $5, -4($30)
div $5, $3
mflo $3
add $30, $30, $4
lw $5, -4($30)
```

Variable	Offset
a	8
b	4
c	0
d	-4

First pop into $5.



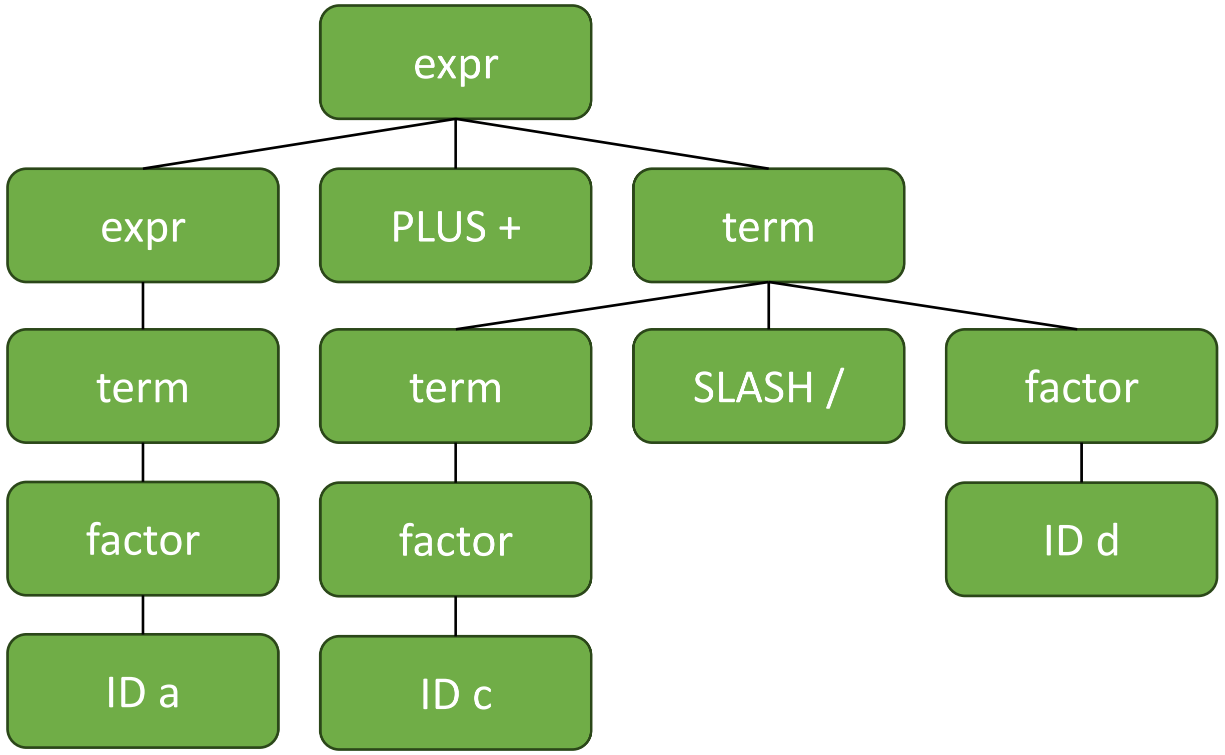
Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
a+c/d

lw $3, 8($29)
sw $3, -4($30)
sub $30, $30, $4
lw $3, 0($29)
sw $3, -4($30)
sub $30, $30, $4
lw $3, -4($29)
add $30, $30, $4
lw $5, -4($30)
div $5, $3
mflo $3
add $30, $30, $4
lw $5, -4($30)
add $3, $5, $3
```

Variable	Offset
a	8
b	4
c	0
d	-4



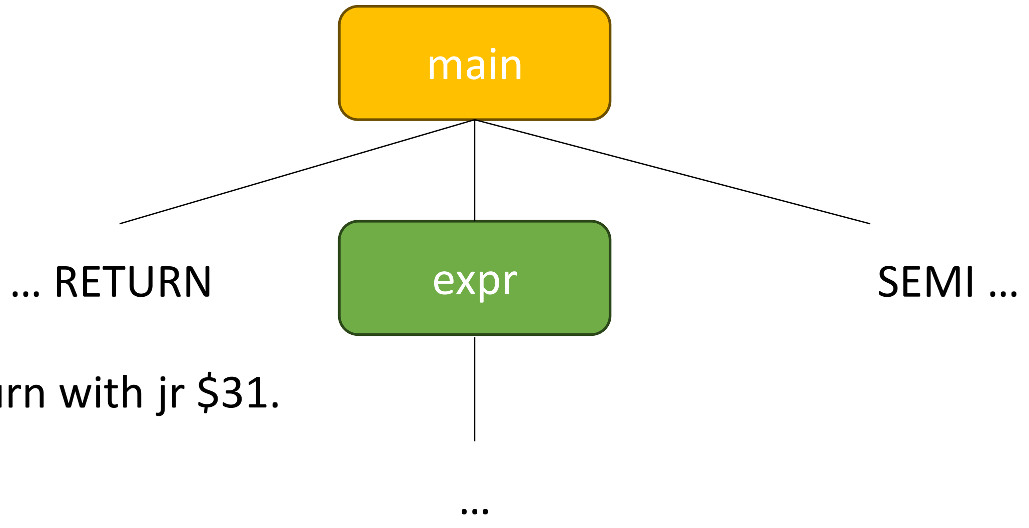
Do the operation. And, we're done.

Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int wain(int a, int b) {
  int c = 482;
  int d = 2;
  return a+c/d;
}
```

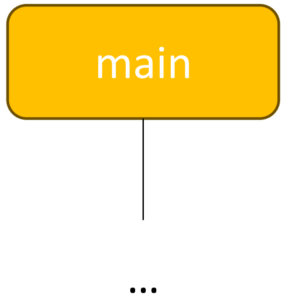
Almost done. We just need to clean up the stack and return with jr $31.



Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.

```
int wain(int a, int b) {
  int c = 482;
  int d = 2;
  return a+c/d;
}
```

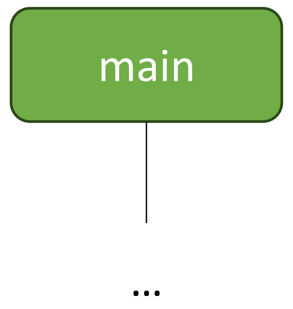


Pop 4 times, since we pushed 4 variables, and then return.

```
add $30, $30, $4    OR    lis $5
add $30, $30, $4          .word 16
add $30, $30, $4          add $30, $30, $5
add $30, $30, $4          jr $31
jr $31
```

Code Generation: Example

- Before we proceed with further discussion, let's look at an example of code generation in action.



```
int wain(int a, int b) {
  int c = 482;
  int d = 2;
  return a+c/d;
}
```

```
; set up constants
lis $4
.word 4
; push variables
sw $1, -4($30)
sub $30, $30, $4
sw $2, -4($30)
sub $30, $30, $4
; set up FP
sub $29, $30, $4
lis $3
.word 482
sw $3, -4($30)
sub $30, $30, $4
lis $3
.word 2
sw $3, -4($30)
sub $30, $30, $4
; procedure body
lw $3, 8($29)
sw $3, -4($30)
sub $30, $30, $4
lw $3, 0($29)
sw $3, -4($30)
sub $30, $30, $4
lw $3, -4($29)
add $30, $30, $4
lw $5, -4($30)
div $5, $3
mflo $3
add $30, $30, $4
lw $5, -4($30)
add $3, $5, $3
; restore stack
add $30, $30, $4
add $30, $30, $4
add $30, $30, $4
add $30, $30, $4
; end program
jr $31
```