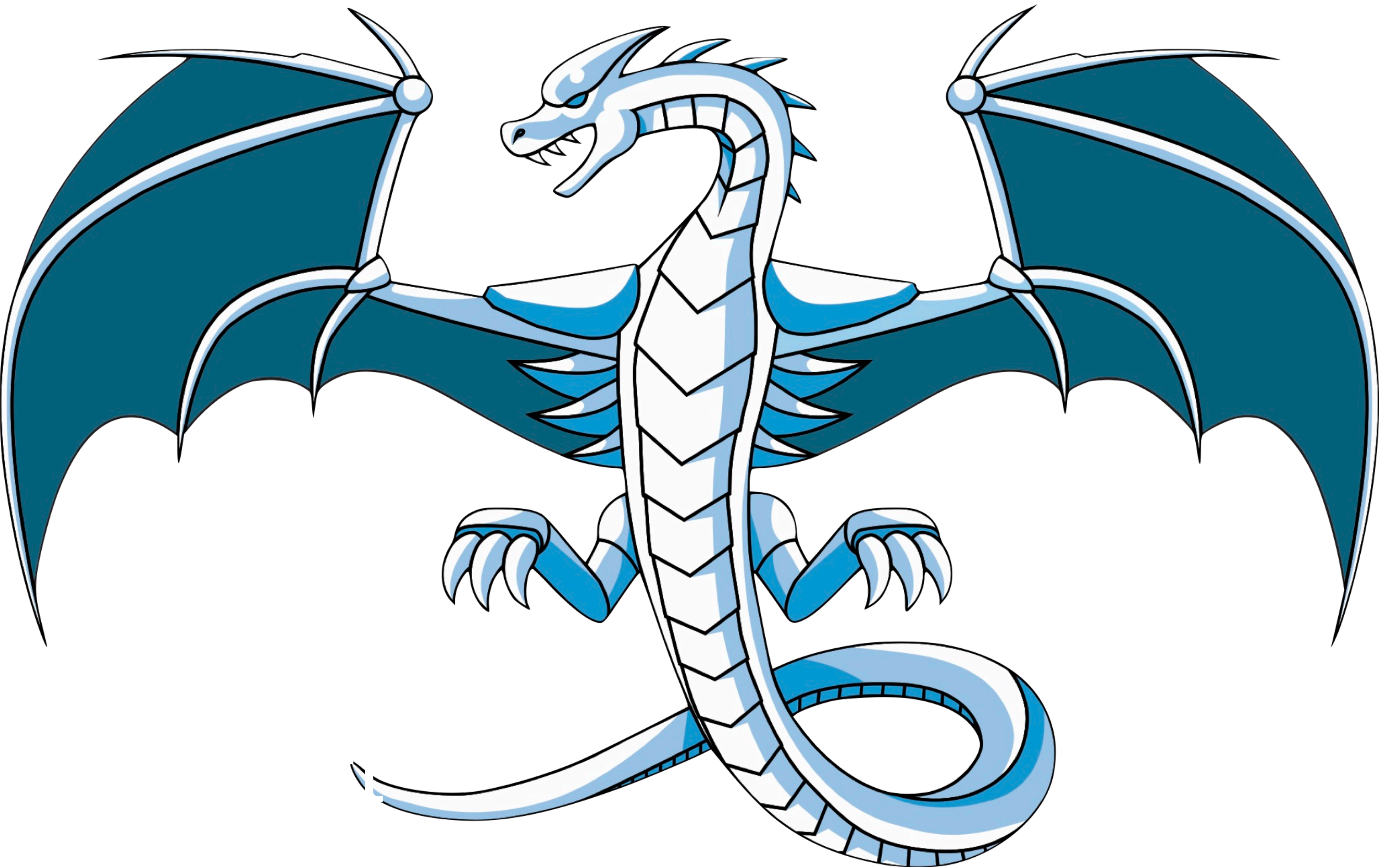
# Academic C Compiler

A compiler for a subset of C

CS4555: Compiler Construction

James Bingen
Dinu Blanovschi
Yigit Colakoglu



# Academic C Compiler

A compiler for a subset of C

by

James Bingen
Dinu Blanovschi
Yigit Colakoglu

| Student Names |
|---|
| James Bingen |
| Dinu Blanovschi |
| Yigit Çolakoğlu |

| | |
|---|---|
| Instructors: | Dr. S.S. Chakraborty, Dr. M.A. Costea |
| Teaching Assistant: | A. Jacques |
| Project Duration: | December, 2025 - January, 2026 |
| Faculty: | Faculty of Electrical Engineering, Mathematics and Computer Science, Delft |

| | |
|---|---|
| Cover: | LLVM logo |
| Style: | TU Delft Report Style, with modifications by Daan Zwaneveld |

TU Delft

# Contents

# 1

# Language

We define a compact C-like language to keep the semantics familiar while keeping the implementation scope manageable. The program consists of zero or more functions, each with an explicit return type, name, optional parameters, and a compound statement serving as the body.

## 1.1. Declarations and statements

Function parameters, local variables, and function return values are all typed using the built-in primitive types `int`, `float`, or `char`. Variable declarations follow the pattern `type identifier [= expression];` and may appear anywhere a statement is allowed. Statements include blocks, conditionals, loops, return/break/continue, and expressions. Conditionals (`if` with optional `else`) and loops (`while`, `for`) all expect a parenthesized expression before their controlled statement, matching standard C-style syntax. One deviation we have from the C standard is that we require all loops and conditionals to be followed by a block (i.e. `{ ... }`). This is to avoid ambiguity and simplify the parsing.

## 1.2. Expressions

Expressions form a hierarchy of operators. Primary expressions are literals (integer, float, and character constants), identifiers, or parenthesized subexpressions. Unary operators (`++`, `--`, `+`, `-`, `!`, , `*`, `&`) can appear in prefix position according to the usual semantics, while postfix increment/decrement and function calls attach to the end of a primary. Binary operators cover arithmetic, bitwise, logical, comparison, and assignment semantics, and argument lists for function calls are comma-separated expressions.

The grammar enforces precedence and associativity to disambiguate expressions. Assignment operators (including `+=`, `-=`, etc.) associate right-to-left, while all other binary operators are evaluated left-to-right. Detailed precedence/associativity information and the formal grammar can be found in Appendix B.

## 1.3. Disambiguation strategy

The grammar is designed so that every production starts with a distinct token set and keywords are reserved (e.g., `if`, `for`, `return`) So the parser can immediately determine which construct it is currently parsing. Expressions rely on the precedence table and associativity rules to resolve binary operator ordering, while parentheses force tighter grouping when needed. Expression statements always end with a semicolon, so there is no overlap between statements and declarations. Finally, all loops and conditionals must be followed by a block (i.e. `{ ... }`) to avoid the possible ambiguity of braceless statements.

By keeping this structure, the language remains close enough to C to leverage existing intuition while trimming unnecessary features for the project scope.

# 2
# Control flow graph

This chapter presents the implementation of Control Flow Graph (CFG) construction and Static Single Assignment (SSA) form transformation in our compiler. The CFG provides an explicit representation of the program control flow by organizing instructions into basic blocks connected by control flow edges. We apply SSA on this structure to ensure that each variable is assigned exactly once, with phi ($\Phi$) nodes inserted at control flow merge points, to merge values from different execution paths. This combination allows for more powerful dataflow analysis and makes many optimizations easier to be applied [1]. We detail the lowering process from AST to CFG and the phi insertion algorithm based on dominance relationships.

## 2.1. The Lowering Pipeline

The transformation from AST to CFG follows a multi-stage pipeline found in the `lower_ast_to_cfg()` function. This process can be divided into several core phases:

1. **Initial CFG Construction:** Basic blocks are created and populated with instructions by traversing the AST
2. **Graph Structure Finalization:** Successor and predecessor relationships are established and unreachable blocks are removed
3. **SSA Transformation:** Phi nodes are inserted and merge points and variables are converted to SSA form
4. **Error Detection:** Malformed phi nodes indicating uninitialized variables are identified and reported. See Chapter 3
5. **Optimization:** A series of cleanup and optimization passes on the CFG. See Chapter 3

## 2.2. Initial CFG Construction

The first phase creates the control flow graph structure by recursively walking the AST and generating basic blocks. We implemented a builder that maintains the growing CFG, allocating new blocks and value references as needed.

**Entry Block and Parameter Handling** The process begins by creating an entry block where function parameters are represented as concrete instructions in the CFG. For each parameter, two instructions are generated:

- An `Assign` instruction that loads the parameter value.
- An `_AssignVar` instruction that associates the parameter with its corresponding variable.

These temporary `_AssignVar` instructions serve as placeholders during the initial construction and will be eliminated during SSA conversion.

**Recursive Statement Lowering** The lowering process recursively traverses statements, creating basic blocks for control flow constructs:

- **Conditional Statements (if/else):** Generate a diamond pattern with separate blocks for the condition evaluation, then-branch, else-branch, and a merge point where control flow reconverges.
- **While Loops:** Create a cyclic structure with an entry block for condition evaluation, a body block, and an exit block. A back edge connects the body to the entry, representing loop iteration.
- **For Loops:** Similar to while loops, but with additional blocks for initialization and update expressions. The update block executes after each iteration before jumping back to the condition check.
- **Expression Lowering:** Expressions are lowered into sequences of assignment instructions. Binary operations like `&&` and `||` use short-circuit evaluation, creating conditional branches.

Each control flow construct updates the current basic block reference, such that subsequent instructions are added to the correct block.

## 2.3. Graph Structure Finalization

After the initial construction, the CFG structure is finalized through some validation and cleanup steps.

**Linking Successors and Predecessors** The graph finalization step traverses all basic blocks and establishes bidirectional edges based on tail instructions. For each unconditional branch, a single edge is added. For conditional branches, two edges are created (then and else paths). This step also validates that all reachable blocks have defined tail instructions.

**Dead Block Elimination** Dead block elimination removes unreachable basic blocks using a worklist algorithm. Starting from the entry block, it marks all reachable blocks through a breadth first traversal. Blocks not marked as reachable are removed, and block IDs are remapped to maintain a contiguous numbering.

## 2.4. SSA Transformation with Phi Insertion

The SSA transformation transforms variable-based code into SSA form where each value has a unique definition. This is done through Phi node insertion at control flow merge points.

In SSA form, variables cannot be reassigned, creating a challenge at control flow merge points where different execution paths may cause different values for the same variable. Phi nodes solve this problem by selecting the correct value based on which predecessor block control flow came.

A Phi node has the form $\%dest = \phi(\%val1@bb1, \%val2@bb2, \ldots)$ where $\%dest$ receives $\%val1$ if execution came from $bb1$, or $\%val2$ if execution came from $bb2$, and so on. This construct allows SSA form to represent variables whose values depend on the control flow path taken.

**Phi Insertion Process** The Phi insertion phase implements a fixed-point algorithm that tracks variable definitions across basic blocks. It maintains three key data structures:

- **before_vars:** Maps variables to their values at block entry
- **after_vars:** Maps variables to their values at block exit
- **phis:** Tracks phi node sources for each block

The algorithm proceeds into two stages. In the first stage, for each basic block, the algorithm processes instructions sequentially, tracking which variables are assigned. When a block assigns a variable, this assignment is propagated to all successor blocks, which may require phi nodes if the variable has different values from different predecessors. In the second stage, the algorithm iteratively propagates variable values from block exits to block entries. When a variable is defined at a block entry (via phi), but not redefined in the block, its value must be available at exit and propagated to successors. This continues until no new phi nodes are added.

## 2.5. Dominance Relationships and Frontiers

Dominance analysis establishes ordering relationships between basic blocks based on control flow paths. We define formal definitions that will be used in subsequent optimization passes.

**Dominance** Node A dominates node B if and only if there is no way to get to B without going through A first. In other words, all execution paths from entry block to B must pass through A. Every block dominates itself. Dominance forms a tree structure called the dominator tree, where the immediate dominator of each block is its parent.

**Post-Dominance** Node A post-dominates node B if and only if there is no way to exit the program from B without passing through A. In other words, all code paths from B to return statements must pass through A.

More generally, we can extend this concept. Node A post dominates a set of nodes S if and only if there is no way to exit the program from any node S without going through A. This extended definition means all code paths from any block in S to program exit must pass through A.

**Dominance Frontiers** The dominance frontier of a block A is the set of blocks where the dominance of A ends. Formally, a block B is in the dominance frontier of A if:

1. A dominates a predecessor of B
2. A does not strictly dominate B (B is not in the dominated set of A, or B = A)

Dominance frontiers are important for SSA construction because they identify exactly where phi nodes must be placed. When a variable is defined in block A, phi nodes for that variable are needed at all blocks in the dominance frontier of A.

While our implementation does not explicitly compute dominance frontiers using the classical algorithm, the phi insertion process implicitly determines these merge points by tracking where variables have different values from different predecessors.

# 3
# Dataflow analyses

## 3.1. Reaching definitions

Happens during SSA conversion. It inserts $\Phi$ nodes to all blocks for all variables in all places where they have a value on at least one code path. This happens in pass `phi_insertion`.

This is a fixpoint algorithm: it continues until there are no more places where we can insert $\Phi$ nodes anymore.

After all the $\Phi$ nodes are inserted, we have a separate pass, `malformed_phi_reduction`, which deletes the $\Phi$ nodes where the set of sources doesn't match the set predecessors of the current block. What that indicates is that there is at least one code path where the corresponding variable is not assigned a value. We keep track of all the SSA values we removed.

When we see an assignment of the form `%a = %b`, where we previously removed `%b`, we will mark `%a` as removed as well. When `%a` is the source of a $\Phi$ node, then the $\Phi$ node gets removed as well. If any of the statements in the original file were deleted as a result of this pass, that means that the program is ill-formed.

We can however remove "plumbing" $\Phi$ nodes / assignments, which are only there to propagate the values of variables into the next blocks. As long as we don't remove assignments that come from the source code, the program is well-formed. We have additional properties on assignments in the CFG to mark where exactly a value comes from / to which variable it belongs to for this purpose.

## 3.2. Dead value elimination

Dead value elimination is one of the trivial passes to implement in SSA. It happens by marking the returned values, as well as the conditional branch condition values, as "alive", and then at every step it attempts to add the operands of computations that give live values as alive, until everything has been marked. Finally, we perform a sweep through all the blocks and remove assignments to dead values, aka unused values / computation results.

The pass responsible for this is `dead_value_elimination`.

## 3.3. Available expressions & very busy expressions

Both those optimizations are implemented by the same pass (`hoist_pass`). The algorithm we implemented can be summarized as follows:

1. The pass first looks through all the blocks for common computations.
2. If it found at least 2 uses of the same computation, it initializes a set of the blocks where they are defined.
3. It determines the set of nodes that is post-dominated by the set computed at step 2.

4. For each use, it finds the “top-most” point (block) in the set computed at step 3, which both dominates that use and is post-dominated by the set of all uses.
5. If a value with that computation was already computed in that “top-most” block (e.g. because it was already available, or inserted by the same computation used somewhere else in the CFG), it reuses that. Otherwise, it hoists the computation to this “top-most” block. In both cases, it then proceeds to replace all the uses of the original value with the new added computation, and deletes the old computation at the original use site.

We assume all functions in our program are pure, i.e. they have no observable behavior, they just run computations on their inputs, and return an output. In that case, changing the order they run in doesn’t affect our compiler, though a real compiler would handle non-pure functions differently.

# 4 Design

## 4.1. SSA

We chose SSA for the structure as it makes all of the later analysis and optimization stages significantly easier. SSA is used by most modern compilers (LLVM, GCC post-2005 with GIMPLE, Rust with MIR, Swift with SIL etc.).

## 4.2. Parser Implementation

For the parser, we decided to implement a custom pratt parser instead of relying on a parser library or implementing a recursive descent parser. The reasoning behind this was to keep full control over the parsing process while supporting left recursive statements in the grammar. It also had the additional benefit of teaching us more about how pratt parsing works.

We implemented a top-down parser for our grammar, which works by using the next token in the input stream to decide which production to apply. We can deterministically decide which production to apply because the grammar has reserved keywords for top-level statements and expressions (`if`, `for`, `return`, `int`, `float`, `char`, etc.).

Once we know which production we want to apply, we call the appropriate function responsible for parsing that production. This function then returns an AST node. We apply this process iteratively until the entire input stream is consumed or we encounter a syntax error.

During parsing, all nodes generated are stored alongside their location in the input stream to provide better error messages during the later phases of the compiler.

Pratt parsing shines especialy when it comes to handling operator precedence and associativity. We handle it by assigning a binding power to each operator. This binding power is used to determine the order in which operators are applied. When parsing an expression, we keep parsing it until we encounter an operator with a higher binding power or we reach the end of the expression. If we encounter an operator with a higher binding power, we pass the “ownership” of the current operand to the new operator, whose result becomes the new operand for the current operator.

For example, the expression `1 + 2 * 3` is parsed as `1 + (2 * 3)` because the binding power of the `+` operator is lower than the binding power of the `*` operator.

One deviation we take from the traditional Pratt parser is that instead of verbosely defining the left and right binding power for each operator and capturing the associativity in the grammar through that, we simplify things by giving a single binding power to each operator and an associativity (`left` or `right`). We then derive the left and right binding power from these values via the following formula:

$$\begin{gathered}\text{left BP} = \text{BP}\ (+1 \text{ if associativity = left})\\ \text{right BP} = \text{BP}\ (+1 \text{ if associativity = right})\end{gathered}$$

# A

# Task Division

This section has the task division among our group.

| Task | Yigit | Dinu | James |
|---|---|---|---|
| Language Design | 🟦 | 🟦 | 🟦 |
| Parser | 🟦 | | |
| Control Flow Graph | | | 🟦 |
| SSA Transformation | | | 🟦 |
| Data Flow Analysis | | 🟦 | |
| Optimizations | | 🟦 | |
| LLVM Compilation | | 🟦 | |
| Testing | 🟦 | 🟦 | 🟦 |
| Report | 🟦 | 🟦 | 🟦 |

# B

# Language grammar

This appendix specifies the concrete grammar of the language and documents the nonterminals used by the compiler frontend. Operator precedence is handled by a Pratt parser and is therefore defined separately from the grammar structure. Railroad diagrams are used to visualize productions and were generated using an external tool[1]

## B.1. Operator precedence and associativity

The parser relies on Pratt parsing, so every operator is assigned a binding power that determines how tightly it binds its operands. Table B.1 lists the powers used in the compiler frontend together with each operator's associativity.

**Table B.1:** Operator binding powers used in the Pratt parser.

| BP | Operators | Location | Associativity |
|---|---|---|---|
| 90 | `++, --` | Postfix | — |
| 80 | `!, \~{}, -, +, *, \&, ++, --` | Prefix | — |
| 70 | `*, /, \%` | Infix | Left |
| 60 | `+, -` | Infix | Left |
| 55 | `<<, >>` | Infix | Left |
| 50 | `<, >, <=, >=` | Infix | Left |
| 45 | `==, !=` | Infix | Left |
| 40 | `\&` | Infix | Left |
| 35 | `\^` | Infix | Left |
| 30 | `\|` | Infix | Left |
| 25 | `\&\&` | Infix | Left |
| 20 | `\|\|` | Infix | Left |
| 10 | `=, +=, -=, *=, /=, \%=` | Infix | Right |

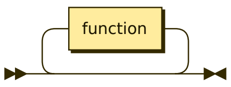
## B.2. Grammar specification

Each subsection below documents a nonterminal in the language grammar. After every rule we list the nonterminals that reference the production.

### B.2.1. Program

This is the root nonterminal of the grammar. It represents the entire program and consists of zero or more functions.

[1]RR – Railroad Diagram Generator

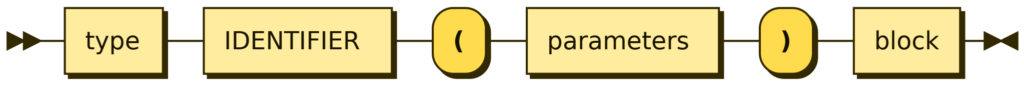


```
program ::= function*
```

### B.2.2. Declaration forms

Declarations introduce named entities into the program and define their type and scope. The language currently supports function and variable declarations.

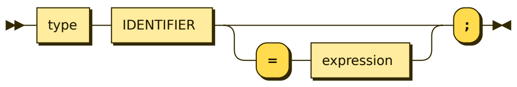
Function



```
function ::= type IDENTIFIER '(' parameters ')' block
```

Referenced by: `program`.

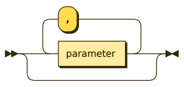
Variable declaration



```
var_decl ::= type IDENTIFIER ( '=' expression )? ';'
```

Referenced by: `for_init`, `declaration_stmt`.

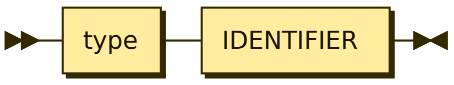
### B.2.3. Parameters



```
parameters ::= ( parameter ( ',' parameter )* )?
```

Referenced by: `function`.

### B.2.4. Parameter



```
parameter ::= type IDENTIFIER
```

Referenced by: `parameters`.

### B.2.5. Type

```
type ::= 'int'
       | 'float'
       | 'char'
```

Referenced by: `function`, `parameter`, `var_decl`.

### B.2.6. Statement classes

Statements are grouped by their control-flow behavior and syntactic role. This classification mirrors the structure commonly used in imperative language specifications.

Statement

```
statement ::= compound_stmt
            | selection_stmt
            | iteration_stmt
            | jump_stmt
            | declaration_stmt
            | expr_stmt
```

Referenced by: `block`.

### Compound statement

```
compound_stmt ::= block
```

Referenced by: `statement`, `if_stmt`, `for_stmt`, `while_stmt`.

### Selection statement

```
selection_stmt ::= if_stmt
```

Referenced by: `statement`.

### Iteration statement

```
iteration_stmt ::= while_stmt
                 | for_stmt
```

Referenced by: `statement`.

### Jump statement

```
jump_stmt ::= return_stmt
            | break_stmt
            | continue_stmt
```

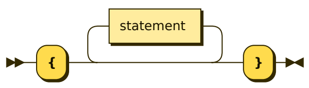
Referenced by: `statement`.

### Declaration statement

```
declaration_stmt ::= var_decl
```

Referenced by: `statement`.

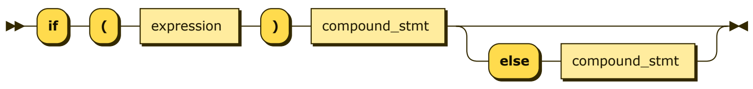
## B.2.7. Block



```
block ::= '{' statement* '}'
```

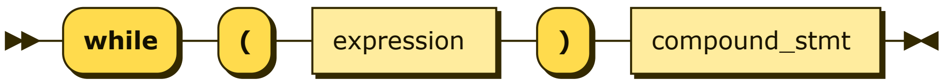
Referenced by: `function`, `compound_stmt`.

## B.2.8. If statement



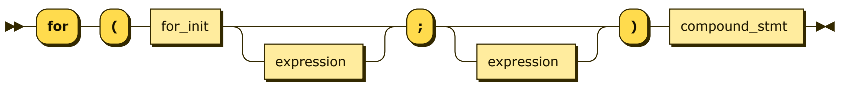
```
if_stmt ::= 'if' '(' expression ')' compound_stmt ( 'else' compound_stmt )?
```

## B.2.9. While statement



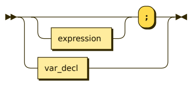
```
while_stmt ::= 'while' '(' expression ')' compound_stmt
```

## B.2.10. For statement



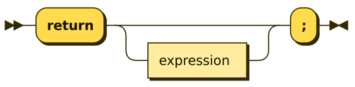
```
for_stmt ::= 'for' '(' for_init expression? ';' expression? ')' compound_stmt
```

### B.2.11. For initializer



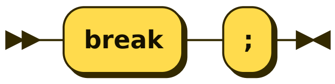
```
for_init ::= expression? ';'
           | var_decl
```

### B.2.12. Return statement



```
return_stmt ::= 'return' expression? ';'
```

### B.2.13. Break statement



```
break_stmt ::= 'break' ';'
```

### B.2.14. Continue statement

```
continue_stmt ::= 'continue' ';'
```

### B.2.15. Expression statement

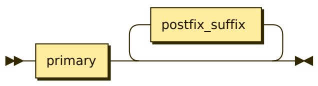
expression
;

```
expr_stmt ::= expression ';'
```

### B.2.16. Postfix constructs

Postfix constructs apply operations to an already-formed primary expression. They include function calls and postfix increment and decrement operations.

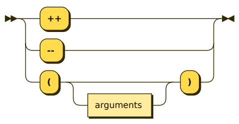
#### Postfix expression



```
postfix_expr ::= primary postfix_suffix*
```

Referenced by: `unary_expr`.

#### Postfix suffix



```
postfix_suffix ::= increment_op
                 | call_expr
```

Referenced by: `postfix_expr`.

#### Increment operator

```
increment_op ::= '++' | '--'
```

Referenced by: `postfix_suffix`.

#### Call expression

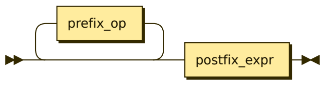
```
call_expr ::= '(' arguments? ')'
```

Referenced by: `postfix_suffix`.

### B.2.17. Unary expressions

Unary expressions apply prefix or postfix operators to a single operand. They include both value-producing and side-effecting operations.

#### Unary expression



```
unary_expr ::= prefix_op* postfix_expr
```

Referenced by: `expression`, `binary_expr`.

#### Unary operators

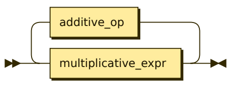
```
prefix_op ::= '++' | '--' | '+' | '-' | '!' | '~' | '*' | '&'
```

Referenced by: `unary_expr`.

### B.2.18. Binary expressions

Binary expressions combine two operands using infix operators. Operator precedence and associativity are defined by the Pratt parser binding powers.

#### Binary expression



```
binary_expr ::= unary_expr ( binary_op unary_expr )*
```

Referenced by: `expression`.

#### Binary operators

```
binary_op ::= '=' | '+=' | '-=' | '*=' | '/=' | '%='
            | '||' | '&&'
            | '|' | '^' | '&'
            | '==' | '!='
            | '<' | '<=' | '>' | '>='
            | '<<' | '>>'
            | '+' | '-'
            | '*' | '/' | '%'
```

Referenced by: `binary_expr`.

# C Other optimization passes

## C.1. Constant propagation (`constant_propagation`)

The `constant_propagation` pass is responsible for performing “constexpr” evaluations, e.g. evaluations of pure values, where the operands are also constant, can be computed at compile time. This pass might enable the `dead_value_elimination` pass later for the operands, if any.

This pass also evaluates conditional branches, and if the condition is constant, it transforms them into unconditional branches, which might enable the `block_inliner` pass, and cleaning up the condition in a `dead_value_elimination` pass if unused elsewhere.

## C.2. Block inliner (`block_inliner`)

This pass is responsible for inlining blocks into their predecessors. The conditions for this pass are:

1. The predecessor block has an uncoditional branch to its successor.
2. The successor block has a single predecessor.

In this case, the `block_inliner` pass will move all the instructions from the successor block into the predecessor, replace the tail of the predecessor with the tail of the successor, and then delete the successor block, as its not used anywhere else.

## C.3. Block deduplicator (`block_dedup`)

In the case where multiple blocks don’t have any instructions, but have the same tail, this pass will deduplicate them. This works for pairs where the 2 duplicate blocks don’t share the same successor, or if they do, then that successor **must not** have $\Phi$ nodes. This enables the `tail_unification` pass.

## C.4. Tail unification (`tail_unification`)

This pass is responsible for turning conditional branches where both the branches lead to the same target block into unconditional branches to that block, removing the need for a condition. This might enable `dead_value_elimination` for the condition.

## C.5. $\Phi$ to select (`phi_to_sel`)

Consider the diamond CFG (see Figure C.1). The `phi_to_sel` pass is responsible for converting the $\Phi$ nodes in D to select statements in A, when B and C don’t have any computations (instructions) of their own, their only predecessor is A, and D’s only predecessors are B and C.

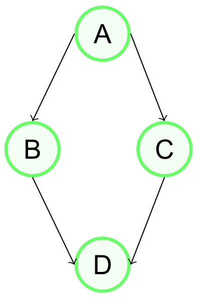


**Figure C.1:** Diamond CFG

# D
# LLVM Backend

We have implemented an (optional) LLVM backend for our compiler. Its job is to lower the CFG produced and turn it into an LLVM module, which we pass on to LLVM to optimize and then output.

LLVM has been around for decades, and has received a long series of improvements over the years. It is used as a backend by multiple compilers (clang, rustc, swift, Kotlin/Native etc.), giving us the perfect foundation for our backend.

Refer to the README for how to build the compiler with the LLVM backend (disabled by default).

# E

# Testing

Our testing strategy is to have test cases as source files in a directory, and test harnesses that read those test cases and execute the tests, via libtest-mimic[1].

The test cases start with comments that are parsed by the harness. After that, the harness runs the compiler on the source file and with the configuration parsed from the comments, and asserts that the output of the compiler is the same as the expected output (available in different files living next to the sources they test), effectively basing our testing approach on snapshot testing. To update the snapshots, the tests are run with a `BLESS=1` environment variable.

[1]https://rust-exercises.com/advanced-testing/09_test_harness/00_intro

# References

[1] J. Singer, P. Brisk, F. Rastello, et al. *Static Single Assignment Book*. Accessed on [add date you accessed it]. 2018. URL: `https://pfalcon.github.io/ssabook/latest/book-full.pdf`.