Derivation of LL(1) Grammar for C0

Stochastic Batman

January 1st, 2026

1 Original Context-Free Grammar of C0

The original grammar is presented below, with dereference operator changed from * to @.

1.1 Lexical Rules

<table>
<tr><th>Non-terminal</th><th>Production</th><th>Description</th></tr>
<tr><td><Di></td><td>0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9</td><td>Digit</td></tr>
<tr><td><DiS></td><td><Di> | <Di><DiS></td><td>Digit sequence</td></tr>
<tr><td><Le></td><td>a | ... | z | A | ... | Z | _</td><td>Letter</td></tr>
<tr><td><DiLe></td><td><Le> | <Di></td><td>Alphanumeric symbol</td></tr>
<tr><td><DiLeS></td><td><DiLe> | <DiLe><DiLeS></td><td>Sequence of alphanumeric symbols</td></tr>
<tr><td><Na></td><td><Le> | <Le><DiLeS></td><td>Name</td></tr>
<tr><td><C></td><td><DiS> | <DiS>u | null</td><td>int/uint/null constant</td></tr>
<tr><td><CC></td><td>'_' | ... | '~'</td><td>Char-constant with ASCII code</td></tr>
<tr><td><BC></td><td>true | false</td><td>Bool-constant</td></tr>
<tr><td><id></td><td><Na> | <id>.<Na> | <id>[<E>] | <id>@ | <id>&</td><td>Identifier (field, index, deref, addr)</td></tr>
</table>

1.2 Expressions

<table>
<tr><th>Non-terminal</th><th>Production</th><th>Description</th></tr>
<tr><td><F></td><td><id> | -<F> | (<E>) | <C></td><td>Factor</td></tr>
<tr><td><T></td><td><F> | <T>*<F> | <T>/<F></td><td>Term</td></tr>
<tr><td><E></td><td><T> | <E>+<T> | <E>-<T></td><td>Expression</td></tr>
<tr><td><Atom></td><td><E> > <E> | <E> >= <E> | <E> < <E> | <E> <= <E> | <E> == <E> | <E> != <E> | <BC></td><td>Atom</td></tr>
<tr><td><BF></td><td><id> | <Atom> | !<BF> | (<BE>)</td><td>Boolean factor</td></tr>
<tr><td><BT></td><td><BF> | <BT> && <BF></td><td>Boolean term</td></tr>
<tr><td><BE></td><td><BT> | <BE> || <BT></td><td>Boolean expression</td></tr>
<tr><td><Pa></td><td><E> | <BE> | <CC></td><td>Parameter</td></tr>
</table>

<table>
<tr><td><PaS></td><td><Pa> | <Pa>,<PaS></td><td>Parameter sequence</td></tr>
</table>

1.3 Statements

<table>
<tr><th>Non-terminal</th><th>Production</th><th>Description</th></tr>
<tr><td><St></td><td><id> = <E> | <id> = <BE> | <id> = <CC> | if <BE> { <StS> } | if <BE> { <StS> } else { <StS> } | while <BE> { <StS> } | <id> = <Na>(<PaS>) | <id> = <Na>() | <id> = new <Na>@</td><td>Assignment / if-then / if-then-else / while / call / alloc</td></tr>
<tr><td><StS></td><td><St> | <St>; <StS></td><td>Statement sequence</td></tr>
<tr><td><rSt></td><td>return <E> | return <BE> | return <CC></td><td>Return statement</td></tr>
</table>

1.4 Types and Declarations

<table>
<tr><th>Non-terminal</th><th>Production</th><th>Description</th></tr>
<tr><td><Ty></td><td>int | bool | char | uint | <Na></td><td>Basic type</td></tr>
<tr><td><VaD></td><td><Ty> <Na></td><td>Variable declaration</td></tr>
<tr><td><VaDS></td><td><VaD> | <VaD>;<VaDS></td><td>Variable declaration sequence</td></tr>
<tr><td><TE></td><td><Ty>[<DiS>] | <Ty>@ | struct { <VaDS> }</td><td>Type expression (array/pointer/struct)</td></tr>
<tr><td><TyD></td><td>typedef <TE> <Na></td><td>Type declaration</td></tr>
<tr><td><TyDS></td><td><TyD> | <TyD>;<TyDS></td><td>Type declaration sequence</td></tr>
</table>

1.5 Functions and Program

<table>
<tr><th>Non-terminal</th><th>Production</th><th>Description</th></tr>
<tr><td><body></td><td><rSt> | <StS>;<rSt></td><td>Function body</td></tr>
<tr><td><PaDS></td><td><VaD> | <VaD>,<PaDS></td><td>Parameter declarations</td></tr>
<tr><td><FuD></td><td><Ty> <Na>(<PaDS>){<VaDS>;<body>} | <Ty> <Na>(<PaDS>){<body>} | <Ty> <Na>(){<VaDS>;<body>} | <Ty> <Na>(){<body>}</td><td>Function declaration</td></tr>
<tr><td><FuDS></td><td><FuD> | <FuD>;<FuDS></td><td>Function sequence</td></tr>
<tr><td><prog></td><td><TyDS>;<VaDS>;<FuDS> | <VaDS>;<FuDS> | <TyDS>;<FuDS> | <FuDS></td><td>Program</td></tr>
</table>

2 Deriving the LL(1) Grammar

To transform the original grammar into an LL(1) grammar, I address issues such as left recursion, common prefixes in alternatives, and ambiguity. I proceed section by section, deriving each non-terminal in the LL(1) version from the original rules. I introduce new non-terminals for tails to handle recursion and factoring.

2.1 Lexical Rules and Terminals

The lexical rules are mostly regular expressions and not recursive, so they are largely unchanged. In the LL(1) version, I use terminals like ID for <Na>, NUM for <DiS>, etc.

I refactor the identifier <id> into l-value with postfix operators.

Original <id>: <Na> | <id>.<Na> | <id>[<E>] | <id>@ | <id>&.

This has left recursion. To eliminate it, I introduce <lvalue> and <lvalue_tail>.

Derive <lvalue>:

I start with base: <lvalue> -> ID <lvalue_tail>, where ID replaces <Na>.

Now, the recursive parts: . ID, [<Expr>], @, &, and ϵ for no more.

So, <lvalue_tail> -> . ID <lvalue_tail> | [<Expr>] <lvalue_tail> | @ <lvalue_tail> | & <lvalue_tail> | ϵ.

This is left-factored already since each alternative starts with a distinct terminal: ., [, @, &, or ϵ. No left recursion because tail is after the operator. This matches the LL(1) version.

For constants, <C>, <CC>, <BC> become terminals.

2.2 Types and Declarations

I start with types.

Original <Ty>: int | bool | char | uint | <Na>.

I replace <Na> with ID, so <Ty> -> int | bool | char | uint | ID. Unchanged.

Original <TE>: <Ty>[<DiS>] | <Ty>@ | struct { <VaDS> }.

I replace <DiS> with NUM. This has common prefix <Ty> in first two alternatives.

Left-factor: I introduce <TEprime> for the modifier.

<TE> -> <Ty> <TEprime> | struct { <VaDS> }.

<TEprime> -> [NUM] | @ | ϵ.

Now distinct starts: int/bool/char/uint/ID vs struct. Within <TEprime>, starts with [, @, or ϵ. Matches LL(1).

Variable declarations:

Original <VaD>: <Ty> <Na> -> <Ty> ID. Unchanged.

Original <VaDS>: <VaD> | <VaD>; <VaDS>. Left recursive.

Eliminate: <VaDS> -> <VaD> <VaDS_tail>.

<VaDS_tail> -> ; <VaD> <VaDS_tail> | ϵ.

Starts with ; or ϵ. Matches LL(1).

Type declarations follow similarly:

<TyDS> -> <TyD> <TyDS_tail>.

<TyDS_tail> -> ; <TyD> <TyDS_tail> | ϵ.

Optional typedefs: <TDSO> -> <TyDS> | ϵ.

2.3 Expressions

The expressions are separated into arithmetic and boolean in the original grammar, but in LL(1), I combine them into <Expr> with precedence levels.

Multiplicative expressions:

Original <T> -> <F> | <T>*<F> | <T>/<F>. This is left recursive.

I remove it: <MulExpr> -> <Primary> <MulExpr_tail>.

<MulExpr_tail> -> * <Primary> <MulExpr_tail> | / <Primary> <MulExpr_tail> | ϵ.

Additive expressions:

Original <E> -> <T> | <E>+<T> | <E>-<T>.

I transform it to: <AddExpr> -> <MulExpr> <AddExpr_tail>.

<AddExpr_tail> -> + <MulExpr> <AddExpr_tail> | - <MulExpr> <AddExpr_tail> | ϵ.

Relational expressions:

I create: <RelExpr> -> <AddExpr> <RelExpr_tail>.

<RelExpr_tail> -> <rel_op> <AddExpr> <RelExpr_tail> | ϵ.

Boolean expressions:

I build the AND level: <AndExpr> -> <RelExpr> <AndExpr_tail>.
<AndExpr_tail> -> && <RelExpr> <AndExpr_tail> | ϵ.
And the OR level: <Expr> -> <AndExpr> <Expr_tail>.
<Expr_tail> -> || <AndExpr> <Expr_tail> | ϵ.

2.4 Statements

Original <St> has many alternatives. I unify them into:
<lvalue> = <RHS> | if <Expr> { <StS> } <EP> | while <Expr> { <StS> }.
Where I define <RHS> -> <Expr> | new ID @.
For if-then-else, I left factor by introducing:
<EP> -> else { <StS> } | ϵ.
Statement sequences:
Original <StS>: <St> | <St> ; <StS>. Left recursive.
I remove it: <StS> -> <St> <StS_tail>.
<StS_tail> -> ; <St> <StS_tail> | ϵ.
For return: I use **return** <Expr>, since I've unified the expression types.

2.5 Function Body

Original body: <rSt> | <StS> ; <rSt>.
In LL(1), I define: <body> -> <SSO> <rSt>.
<SSO> -> <StS> | ϵ.
For locals, I introduce the local keyword to distinguish local variable declarations:
<locals> -> local <VaDS> | ϵ.

2.6 Function Parameters

Original <PaDS> -> <VaD> | <VaD>, <PaDS>. Left recursive.
I remove it: <PaDS> -> <VaD> <PaDS_tail>.
<PaDS_tail> -> , <VaD> <PaDS_tail> | ϵ.
I define <PDSO> -> <PaDS> | ϵ for empty ().

2.7 Program and Globals

In LL(1), I unify the program structure as optional <TDSO> followed by <GDs>.
I define: <GD> -> <Ty> ID <GDT>.
<GDT> -> ; | (<PDSO>) { <locals> <body> }.
This distinguishes global variables (<Ty> ID ;) from functions (<Ty> ID (...) { ... }).
I define: <GDs> -> <GD> <GDs> | ϵ.
<prog> -> <TDSO> <GDs>.

3 Derived LL(1) Grammar of C0

The derived LL(1) grammar is as follows.

3.1 Types

Non-terminal	Production	Description
<Ty>	int | bool | char | uint | ID	Basic type
<TEprime>	[NUM] | @ | ϵ	Type modifier

<table>
<tr><th></th><th></th><th></th></tr>
<tr><td><TE></td><td><Ty> <TEprime> | struct { <VaDS> }</td><td>Type expression</td></tr>
</table>

3.2 Variable Declarations

<table>
<tr><th>Non-terminal</th><th>Production</th><th>Description</th></tr>
<tr><td><VaD></td><td><Ty> ID</td><td>Variable declaration</td></tr>
<tr><td><VaDS_tail></td><td>; <VaD> <VaDS_tail> | ϵ</td><td>More var decls</td></tr>
<tr><td><VaDS></td><td><VaD> <VaDS_tail></td><td>Var decl sequence</td></tr>
</table>

3.3 Type Declarations

<table>
<tr><th>Non-terminal</th><th>Production</th><th>Description</th></tr>
<tr><td><TyD></td><td>typedef <TE> ID</td><td>Type declaration</td></tr>
<tr><td><TyDS_tail></td><td>; <TyD> <TyDS_tail> | ϵ</td><td>More type decls</td></tr>
<tr><td><TyDS></td><td><TyD> <TyDS_tail></td><td>Type decl sequence</td></tr>
<tr><td><TDSO></td><td><TyDS> | ϵ</td><td>Optional typedefs</td></tr>
</table>

3.4 L-values

<table>
<tr><th>Non-terminal</th><th>Production</th><th>Description</th></tr>
<tr><td><lvalue></td><td>ID <lvalue_tail></td><td>L-value</td></tr>
<tr><td><lvalue_tail></td><td>. ID <lvalue_tail> | [<Expr>] <lvalue_tail> | @ <lvalue_tail> | & <lvalue_tail> | ϵ</td><td>L-value postfix</td></tr>
</table>

3.5 Expressions

<table>
<tr><th>Non-terminal</th><th>Production</th><th>Description</th></tr>
<tr><td><Primary></td><td>ID <primary_tail> | - <Primary> | ! <Primary> | (<Expr>) | <C> | <CC> | <BC></td><td>Primary expression</td></tr>
<tr><td><primary_tail></td><td>(<PSO>) | <lvalue_tail></td><td>Call or postfix ops</td></tr>
<tr><td><MulExpr></td><td><Primary> <MulExpr_tail></td><td>Multiplicative expr</td></tr>
<tr><td><MulExpr_tail></td><td>* <Primary> <MulExpr_tail> | / <Primary> <MulExpr_tail> | ϵ</td><td>Mul/div continuation</td></tr>
<tr><td><AddExpr></td><td><MulExpr> <AddExpr_tail></td><td>Additive expr</td></tr>
<tr><td><AddExpr_tail></td><td>+ <MulExpr> <AddExpr_tail> | - <MulExpr> <AddExpr_tail> | ϵ</td><td>Add/sub continuation</td></tr>
<tr><td><RelExpr></td><td><AddExpr> <RelExpr_tail></td><td>Relational expr</td></tr>
<tr><td><RelExpr_tail></td><td><rel_op> <AddExpr> <RelExpr_tail> | ϵ</td><td>Relational continuation</td></tr>
<tr><td><AndExpr></td><td><RelExpr> <AndExpr_tail></td><td>Logical AND expr</td></tr>
</table>

<AndExpr_tail>	&& <RelExpr> <AndExpr_tail> | ϵ	AND continuation
<Expr>	<AndExpr> <Expr_tail>	Full expression
<Expr_tail>	|| <AndExpr> <Expr_tail> | ϵ	OR continuation

3.6 Call Parameters

Non-terminal	Production	Description
<PaS>	<Expr> <PaS_tail>	Parameter sequence
<PaS_tail>	, <Expr> <PaS_tail> | ϵ	More parameters
<PSO>	<PaS> | ϵ	Optional parameters

3.7 Statements

Non-terminal	Production	Description
<RHS>	<Expr> | new ID @	Assignment RHS
<rSt>	return <Expr>	Return statement
<EP>	else { <StS> } | ϵ	Optional else
<St>	<lvalue> = <RHS> | if <Expr> { <StS> } <EP> | while <Expr> { <StS> }	Statement
<StS_tail>	; <St> <StS_tail> | ϵ	More statements
<StS>	<St> <StS_tail>	Statement sequence

3.8 Function Body

Non-terminal	Production	Description
<locals>	local <VaDS> | ϵ	Local declarations
<SSO>	<StS> | ϵ	Optional statements
<body>	<SSO> <rSt>	Function body

3.9 Function Parameters

Non-terminal	Production	Description
<PaDS>	<VaD> <PaDS_tail>	Param declarations
<PaDS_tail>	, <VaD> <PaDS_tail> | ϵ	More param decls
<PDSO>	<PaDS> | ϵ	Optional param decls

3.10 Program

Non-terminal	Production	Description
<GDT>	; | (<PDSO>) { <locals> <body> }	Var end or function def

<GD>	<Ty> ID <GDT>	Global declaration
<GDs>	<GD> <GDs> | ϵ	Global decl sequence
<prog>	<TDSO> <GDs>	Program

4 Terminals

Terminal	Description
ID	Identifier
NUM	Integer literal (for array sizes)
<C>	Integer constant
<CC>	Character constant
<BC>	Boolean constant (true, false)
<rel_op>	Relational operator (<, >, <=, >=, ==, !=)
int, bool, char, uint	Built-in type keywords
struct, typedef, new	Type-related keywords
if, else, while, return, local	Control and declaration keywords
+, -, *, /	Arithmetic operators
&&, ||, !	Logical operators
@, &	Pointer dereference and address-of
.	Field access
[,]	Array indexing
(,)	Parentheses
{, }	Braces
;	Statement/declaration separator
,	Parameter separator
=	Assignment