ASL Syntax Reference
DDI 0620

Arm Architecture Technology Group

September 25, 2024

Contents

Chapter 1

Non-Confidential Proprietary Notice

is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

Chapter 2

Disclaimer

This document is part of the ASLRef material.

This material covers ASLv1, a new, experimental, and as yet unreleased version of ASL.

The development version of ASLRef can be found here:
https://github.com/herd/herdtools7.

A list of open items being worked on can be found here:
https://github.com/herd/herdtools7/blob/master/asllib/doc/ASLRefProgress.tex.

This material is work in progress, more precisely at Alpha quality as per Arm's quality standards. In particular, this means that it would be premature to base any production tool development on this material.

However, any feedback, question, query and feature request would be most welcome; those can be sent to Arm's Architecture Formal Team Lead Jade Alglave (`jade.alglave@arm.com`) or by raising issues or PRs to the herdtools7 github repository.

Chapter 3

Introduction

This document defines how an ASL specification, given as text, can be transformed into an *abstract syntax tree*, which is a tree-like data structure. This transformation occurs in three stages:

Lexical Analysis The text is first transformed into a list of *tokens*. This stage is defined in Chapter 5;

Parsing The list of tokens is transformed into a *parse tree*. This stage is defined in Chapter 6;

Abstraction The parse tree is transformed into an abstract syntax tree. This is a conceptual stage. In actuality, the parsing stage transforms the list of tokens directly into an abstract syntax tree. However, it is useful to distinguish between the parsing state and the abstraction stage. ASL abstract syntax trees are defined in Chapter 7. This stage is defined in Chapter 8.

Chapter 4

Formal System

In this chapter, we define the mathematical concepts and notations used throughout. This chapter appears in all of the ASL references as its content it used in all of them. We start by defining general mathematical concepts and then describe how sets of rules formally define functions and relations.

4.1 Mathematical Definitions and Notations

We use $\triangleq$ to define mathematical concepts.

We define the following sets:

- $\mathbb{N}$ is the set of natural numbers, including 0.
- $\mathbb{N}^+$ is the set of natural numbers, excluding 0.
- $\mathbb{Z}$ is the set of integers.
- $\mathbb{Q}$ is the set of rationals.
- $\mathbb{B}$ is the set of ASL Boolean literals, which consists of `TRUE` and `FALSE`. We employ these literals to represent the corresponding mathematical truth values, which are used to denote whether logical assertions hold or not. We also employ the mathematical meaning of logical conjunction $\wedge$, logical disjunction $\vee$, and logical negation $\neg$, given next. For a set of Boolean values A:

$$\wedge A \triangleq \begin{cases} \texttt{TRUE} & \text{if all values in A are } \texttt{TRUE} \\ \texttt{FALSE} & \text{otherwise} \end{cases}$$

$$\vee A \triangleq \begin{cases} \texttt{FALSE} & \text{if all values in A are } \texttt{FALSE} \\ \texttt{TRUE} & \text{otherwise} \end{cases}$$

For a pair of Boolean values $a, b \in \mathbb{B}$, we define $a \wedge b \triangleq \wedge\{a, b\}$ and $a \vee b \triangleq \vee\{a, b\}$. Finally, $\neg\texttt{TRUE} \triangleq \texttt{FALSE}$ and $\neg\texttt{FALSE} \triangleq \texttt{TRUE}$.

- $\mathbb{I}$ is the set of all ASL identifiers.
- $\mathbb{L}$ is the set of all labels of Abstract Syntax Tree (AST) nodes.
- $\mathbb{S}$ is the set of all ASCII strings.

We utilize the notation $\overbrace{a}^{b}$ to enable us to name the mathematical term a as b so that we can refer to it in text. We especially use this to name the input arguments and output results of functions and relations. For example, the input argument of `sign`, which is defined next is named q.

Definition 1 (Sign of a Rational Number) *The function* $\mathtt{sign} : \overbrace{\mathbb{Q}}^{q} \to \{-1, 0, 1\}$ *returns the sign of* $\mathtt{q}$*:*

$$\mathtt{sign}(q) \triangleq \begin{cases} 1 & \text{if } q > 0 \\ 0 & \text{if } q = 0 \\ -1 & \text{if } q < 0 \end{cases}$$

Definition 2 (Empty Set) *The* empty set *— the set that does not contain any element — is denoted as* $\emptyset$*.*

Definition 3 (Set Cardinality) *For a set* S*, the notation* $|S|$ *stands for the number of elements in* S*.*

Definition 4 (Powerset) *The* powerset *of a set* A*, denoted as* $\mathcal{P}(A)$*, is the set of all subsets of* A*, including the empty set and* A *itself:*

$$\mathcal{P}(A) \triangleq \{B \mid B \subseteq A\} \ .$$

Definition 5 (Powerset of Finite Subsets) *The* powerset of finite subsets *of a set* A*, denoted as* $\mathcal{P}_{fin}(A)$*, is the set of all finite subsets (including the empty set) of* A*:*

$$\mathcal{P}_{fin}(A) \triangleq \{B \mid B \subseteq A, |B| \in \mathbb{N}\} \ .$$

Definition 6 (Cartesian Product) *The* Cartesian product *of sets* A *and* B*, denoted* $A \times B$ *is* $A \times B \triangleq \{(a, b) \mid a \in A, b \in B\}$*.*

Definition 7 (Partial Function) *A* partial function*, denoted* $f : A \rightharpoonup B$*, is a function from a subset of* A *to* B*. The* domain *of a partial function* f*, denoted* $\mathrm{dom}(f)$*, is the subset of* A *for which it is defined. We write* $f(x) = \perp$ *to denote that* x *is not in the domain of* f*, that is,* $x \notin \mathrm{dom}(f)$*.*

Notice that the domain of a partial function need not be finite, which is what the following definition covers.

Definition 8 (Finite-domain Function) *The notation* $\to_{fin}$ *stands for a function whose domain is finite.*

Definition 9 (Empty Function) *The function with an empty domain is denoted as* $\emptyset_\lambda$.

Definition 10 (Function Update) *The function denoted as* $f[x \mapsto v]$ *is a function identical to* f*, except that* x *is bound to* v*. That is, if* $g = f[x \mapsto v]$ *then*

$$g(z) = \begin{cases} v & \text{if } z = x \\ f(z) & \text{otherwise} \end{cases} .$$

The notation $\{i = 1..k : a_i \mapsto b_i\}$ *stands for the function formed from the corresponding input-output pairs:* $\emptyset_\lambda[a_1 \mapsto b_1] \ldots [a_k \mapsto b_k]$.

Definition 11 (Function Restriction) *The* restriction *of a function* $f : X \to Y$ *to a subset of its domain* $A \subseteq \mathrm{dom}(f)$*, denoted as* $f|_A$*, is defined in terms of the set of input-output pairs:*

$$f|_A \triangleq \{(x, f(x)) \mid x \in A\} .$$

Definition 12 (Function Graph) *The* graph *of a finite-domain function* $f : X \to_{fin} Y$ *is the list of input-output pairs for* f*, given in any order:*

$$\mathit{func\_graph}(f) \triangleq \{(x, f(x)) \mid x \in \mathrm{dom}(f)\} .$$

Throughout this document, we will annotate arguments of relations and functions, wherever it is useful, by writing a name or an expression above the corresponding argument type. This makes convenient to refer to arguments by referring to the corresponding names and helps identify the expressions corresponding to the arguments. For example,

$$\mathsf{choice} : \overbrace{\mathbb{B}}^{b} \times \overbrace{T}^{x} \times \overbrace{T}^{y} \to \overbrace{T}^{z}$$

defines a function type and lets us refer to the first argument as b, the second argument as x, the third argument as y, and to the result as z.

A *parametric function* is a function whose domain is not a priori fixed but rather parameterized by the type of its arguments. An example is the choice function where the type T of x, y, and z is unspecified and inferred from the context where the function is used.

Definition 13 (Choice) *The parametric function* $\mathsf{choice} : \overbrace{\mathbb{B}}^{b} \times \overbrace{T}^{x} \times \overbrace{T}^{y} \to \overbrace{T}^{z}$*, is defined as follows:*

$$\mathit{choice}(\mathtt{b}, x, y) \triangleq \begin{cases} x & \text{if } \mathtt{b} \text{ is } \mathit{TRUE} \\ y & \text{otherwise} \end{cases}$$

4.1.1 Lists

In the remainder of this document, we use the term *list* and *sequence* interchangeably.

A list of elements is either empty, denoted by [ ], or non-empty. A non-empty list is either denoted by listing the elements in sequence, $v_1 \ldots v_k$, or in bracketed form,

$[v_1, \ldots, v_k]$, which is used to aesthetically separate it from surrounding mathematical expressions. The commas carry no special meaning.

For a non-empty list $v_1 \ldots v_k$, the ***head*** of the list is the first element — v_1 — and the ***tail*** of the list is the suffix obtained by removing v_1 from the list.

We refer to individual elements of a non-empty list V by the index notation $V[i]$ where $i \in \mathbb{N}^+$.

Definition 14 (List Length) *The* length *of a list is the number of elements in that list:* $|[\,]| \triangleq 0$ *and* $|v_1, \ldots, v_k| = k$.

We use the notation $a..b$, where $a, b \in \mathbb{Z}$ and $a \leq b$, as a shorthand for the interval $[a \ldots b]$. We write $x_{a..b}$ as a shorthand for the sequence $x_a \ldots x_b$. We write $i = 1..k : V(i)$, where $V(i)$ is a mathematical expression parameterized by i, to denote the sequence of expressions $V(1) \ldots V(k)$. The notation $a \in A : V(a)$, where A is a set and V is an expression parameterized by the free variable a, stands for $V(a_1) \ldots V(a_k)$ where $a_{1..k}$ is an arbitrary ordering of the elements of A.

We write T^* to denote a the type of a possibly-empty list of elements of type T, and T^+ for a non-empty list of elements of type T.

Definition 15 (List Concatenation) *The parametric function* $+ : T^* \times T^* \to T^*$ *concatenates two lists:*

$$\begin{array}{rcl} [\,] + L & \triangleq & L \\ L + [\,] & \triangleq & L \\ l_{1..k} + m_{1..n} & \triangleq & [l_{1..k}, m_{1..n}] \end{array}$$

Definition 16 (Equating List Lengths) *The parametric function*

$$\mathit{equal\_length} : \overbrace{L}^{a} \times \overbrace{L}^{b} \to \mathbb{B}$$

compares the length of two lists:

$$\mathit{equal\_length}(a, b) \triangleq |a| = |b| \ .$$

Definition 17 (Indices of a List) *The parametric function* ***indices*** $: T^* \to \mathbb{N}^*$ *returns the (1-based) list of indices for a given list:*

$$\begin{array}{rcl} \mathit{indices}([\,]) & \triangleq & [\,] \\ \mathit{indices}(v_{1..k}) & \triangleq & [1..k] \ . \end{array}$$

Definition 18 (Unzipping a List of Pairs) *The parametric function*

$$\mathit{unzip} : (T_1 \times T_2)^* \to (T_1^* \times T_2^*)$$

transforms a list of pairs into the corresponding pair of lists:

$$\mathit{unzip}(\mathit{pairs}) \triangleq \begin{cases} ([\,], [\,]) & \text{if } \mathit{pairs} = [\,] \\ (a_{1..k}, b_{1..k}) & \text{else } \mathit{pairs} = (a_1, b_1) \ldots (a_k, b_k) \end{cases} \ .$$

4.1.2 OCaml-style Notations

We use the following notations, which are in the style of the OCaml programming language, to facilitate correspondence with our reference implementation.

The notation $L(v_{1..k})$ is a compound term where L is a label and $v_{1..k}$ is a (possibly singleton) list of mathematical values. We also write $L(T_{1..k})$, where $T_{1..k}$ denotes mathematical types of values, to stand for the type $\{L(v_{1..k}) \mid v_1 \in T_1, \ldots, v_k \in T_k\}$.

Definition 19 (Optional) *The notation $\langle\cdot\rangle$ stands for either an empty set or a singleton set, where* `None` $\triangleq \langle\rangle$ *denotes an empty set and $\langle v\rangle$ denotes a set containing the single element v. The notation $\langle T\rangle$, where T denotes a mathematical type, stands for $\{\langle\rangle\} \cup \{\langle v\rangle \mid v \in T\}$.*

We refer to $\langle T\rangle$ as an optional.

4.2 How we use Rules

An *inference rule* (rule, for short) is an implication between a set of logical assertions, called the *premises* of the rule, and a *conclusion* assertion. The conclusion holds when the conjunction of its premises holds.

We use the following rule notation, where $P_{1..k}$ are the rule premises and C is the conclusion:

$$\frac{P_1 \qquad \ldots \qquad P_k}{C}$$

For example, the rule TypingRule.ELit has one premise:

$$\frac{\texttt{annotate\_literal}(\texttt{v}) \xrightarrow{\text{type}} \texttt{t}}{\texttt{annotate\_expr}(\texttt{tenv}, \texttt{E\_Literal}(\texttt{v})) \xrightarrow{\text{type}} (\texttt{t}, \texttt{E\_Literal}(\texttt{v}))}$$

and the rule TypingRule.Binop (somewhat simplified here) has three premises:

$$\frac{\begin{array}{c}\texttt{annotate\_expr}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\text{type}} (\texttt{t1}, \texttt{e1}') \\ \texttt{annotate\_expr}(\texttt{tenv}, \texttt{e2}) \xrightarrow{\text{type}} (\texttt{t2}, \texttt{e2}') \\ \texttt{check\_binop}(\texttt{tenv}, \texttt{op}, \texttt{t1}, \texttt{t2}) \xrightarrow{\text{type}} \texttt{t}\end{array}}{\texttt{annotate\_expr}(\texttt{tenv}, \texttt{E\_Binop}(\texttt{op}, \texttt{e1}, \texttt{e2})) \xrightarrow{\text{type}} (\texttt{t}, \texttt{E\_Binop}(\texttt{op}, \texttt{e1}', \texttt{e2}'))}$$

The free variables appearing in the premises and conclusion are interpreted universally. That is, the rules apply to any values (of the appropriate types) assigned to their free variables. For example, the rule TypingRule.Binop applies to any choice of values for the free variables `tenv` (a static environment), `e1`, `e2`, `e1'`, `e2'` (expressions), `t`, `t1`, and `t2` (types).

Definition 20 (Grounding) *Assertions can be* grounded *by substituting their free variables with values. A* ground rule *is a rule with all its assertions (premises and conclusion) grounded.*

For example, the following is a grounding of TypingRule.Binop

$$\frac{\begin{array}{c}\texttt{annotate\_expr}(\emptyset_{\text{tenv}}, \text{E\_Literal}(\text{L\_Int}(2))) \xrightarrow{\text{type}} (\text{T\_Int}, \text{E\_Literal}(\text{L\_Int}(2))) \\ \texttt{annotate\_expr}(\emptyset_{\text{tenv}}, \text{E\_Literal}(\text{L\_Int}(3))) \xrightarrow{\text{type}} (\text{T\_Int}, \text{E\_Literal}(\text{L\_Int}(3))) \\ \texttt{check\_binop}(\emptyset_{\text{tenv}}, \texttt{MUL}, \text{T\_Int}, \text{T\_Int}) \xrightarrow{\text{type}} \text{T\_Int}\end{array}}{\begin{array}{c}\texttt{annotate\_expr}(\emptyset_{\text{tenv}}, \text{E\_Binop}(\texttt{MUL}, \text{E\_Literal}(\text{L\_Int}(2)), \text{E\_Literal}(\text{L\_Int}(3)))) \xrightarrow{\text{type}} \\ (\text{T\_Int}, \text{E\_Binop}(\texttt{MUL}, \text{E\_Literal}(\text{L\_Int}(2)), \text{E\_Literal}(\text{L\_Int}(3))))\end{array}}$$

obtained by the following substitutions:

free variable	value
tenv	$\emptyset_{\text{tenv}}$
e1	E_Literal(L_Int(2))
e1′	E_Literal(L_Int(2))
e2	E_Literal(L_Int(3))
e2′	E_Literal(L_Int(3))
t	T_Int
t1	T_Int
t2	T_Int
op	MUL

A set of rules is interpreted disjunctively. That is, each rule is used to determine whether its conclusion holds independently of other rules.

Definition 21 (Axiom) *An* axiom *is a rule with an empty set of premises. An axiom is denoted by simply stating its conclusion.*

An example of an axiom in the ASL type system is TypingRule.SPass:

$$\texttt{annotate\_stmt}(\text{tenv}, \text{S\_Pass}) \xrightarrow{\text{type}} (\text{S\_Pass}, \text{tenv})$$

An example of an axiom in the ASL semantics is SemanticsRule.PAll:

$$\texttt{eval\_pattern}(\texttt{env}, \_, \text{Pattern\_All}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{Bool}(\texttt{TRUE}), \emptyset_{\text{g}})$$

To show that a specification is correct, with respect to the set of type rules, or to show that a specification evaluates to a certain value, with respect to the set of semantic rules, we must apply rules to form a *derivation tree*.

Definition 22 (Derivation Tree) *A* derivation tree *is a tree whose vertices correspond to ground assertions. More specifically, the leaves of a derivation tree correspond to ground axioms, and an internal vertex corresponds to a ground conclusion of a rule with its children corresponding to the ground premises of the same rule.*

4.2.1 Transitions

We use rules as a structured way for defining relations (and therefore functions, as a special case).

To define a relation $R \subseteq X \times Y$, we use assertions of the form $tx \xrightarrow{R} ty$ where tx and ty are logical terms denoting sets of elements from X and Y, respectively. We call such assertions *transitions*. A set of rules M with transition assertions defines the relation

$$R = \{(x, y) \mid x \xrightarrow{R} y \text{ can be derived from rules in } M\} \ .$$

For example, the rule TypingRule.ELit defines a relation between the infinite set of elements of the form `annotate_expr(tenv, E_Literal(v))` (for the infinite choice of values for the free variables `tenv` and `v`) to the infinite set of pairs of the form `(t, E_Literal(v))`, such that the premise holds.

Mutual Exclusion Principle: Our rules follow (with very few deviations, which we point out in context) a mutual exclusion principle, where each rule defines a relation disjoint from the ones defined by the other rules. This makes it easy to determine the rule responsible for a given transition.

4.2.2 Configurations

Our relations range over compound values. That is, values that often nest tuples and lists inside other tuples and lists. We refer to such values as *configurations*. To make it easier to distinguish between different configurations, we will sometimes attach labels to tuples using the OCaml-style notation discussed earlier. We refer to those labels as *configuration domains*. The domain of a configuration $C = L(\ldots)$, denoted `config_domain`(C), is the label L.

We refer to configurations at the origin of a transition as *input configurations* and to the configurations at the destination of a transition as *output transitions*.

For example, the conclusion of the rule TypingRule.ELit has `annotate_expr(tenv, E_Literal(v))` as its input configuration and `(t, E_Literal(v))` as its output configuration. Further, `config_domain(annotate_expr(tenv, E_Literal(v)))` = `annotate_expr`, while the output configuration does not have a configuration domain, since it is an unlabelled pair.

Our rules always make use of labelled input configurations. This makes it easier to ensure the mutual exclusion rule principle.

Our rules always define relations whose sets of input configurations and output configurations are disjoint.

Definition 23 (Fresh Element) *Premises of the form $x \in T$ `is fresh` mean that in any instantiation in a derivation tree, the value of x is unique. That is, different from all other values instantiated for any other variable.*

Definition 24 (Ignore Variable) *To keep rules succinct, we write _ for a mathematical variable whose name is irrelevant for understanding the rule, and can thus be omitted. Each occurrence of _ represents a variable whose name is different from any other free variable in the rule.*

For example, the rule SemanticsRule.PAll, shown above, uses an ignore variable to stand for the value being matched by a `-` pattern. Since the rule does not need to refer to the value, we do not name it and use an ignore variable instead.

4.2.3 Flavors of Equality In Rules

We now explain equality notations in rules, two of which are used in SemanticsRule.Lit, shown here:

$$\frac{\mathsf{env} \overset{\text{is}}{=} (\_, \mathsf{denv}) \qquad \mathtt{x} \in \mathrm{dom}(L^{\mathsf{denv}}) \qquad \mathtt{v} := L^{\mathsf{denv}}(\mathtt{x}) \qquad \mathtt{g} := \mathsf{ReadEffect}(\mathtt{x})}{\mathtt{eval\_expr}(\mathtt{env}, \mathsf{E\_Var}(\mathtt{x})) \xrightarrow{\text{eval}} \mathsf{Normal}((\mathtt{v}, \mathtt{g}), \mathtt{env})}$$

Range: we write $i = 1..k$ to allow listing premises parameterized by i or constructing lists from expressions parameterized by i. For example, given two lists a and b,

$$i = 1..k : a[i] > b[i]$$

is the list of premises

$$\begin{array}{l} a[0] > b[0] \\ \ldots \\ a[k] > b[k] \ . \end{array}$$

Predicate: we write $a = b$ as an assertion of the equality of a and b. For example, the mathematical identity $x \times (y + z) = x \times y + x \times z$.

Deconstruction / "View as": some values, such as tuples, are compound. In order to refer to the structure of compound values, we write $v \overset{\text{is}}{=} f(u_{1..k})$ where the expression on the right hand side exposes the internal structure of v by introducing the variables $u_{1..k}$, allowing us to alias internal components of v. Intuitively, v is re-interpreted as $f(u_{1..k})$. For example, suppose we know that v is a pair of values. Then, $v \overset{\text{is}}{=} (a, b)$ allows us to alias a and b. In SemanticsRule.Lit, we know that the environment `env` is a pair where the first component is a static environment and the second component is a dynamic environment. Therefore, writing $\mathtt{env} \overset{\text{is}}{=} (\_, \mathsf{denv})$ allows us to name the dynamic environment component and then refer to it, while ignoring the static environment component. Similarly, if v is a non-empty list, then $v \overset{\text{is}}{=} [h]{+}t$ deconstructs the list into the head of the list h and its tail t. Given that a variable v represents a list, we write $v \overset{\text{is}}{=} v_{1..k}$ to list its elements and allow referring to them by index.

Definition / "Define as": the notation $\mathtt{x} := \mathtt{e}$ denotes that `x` is a new name serving as an alias for the expression `e`. For example, in the rule SemanticsRule.Lit, we use `g` to name $\mathsf{ReadEffect}(\mathtt{x})$. Aliases allow us to break down complex expressions, but rules can always be rewritten without them, by inlining their right-hand sides:

$$\frac{\mathsf{env} \overset{\text{is}}{=} (\_, \mathsf{denv}) \qquad \mathtt{x} \in \mathrm{dom}(L^{\mathsf{denv}})}{\mathtt{eval\_expr}(\mathtt{env}, \mathsf{E\_Var}(\mathtt{x})) \xrightarrow{\text{eval}} \mathsf{Normal}((L^{\mathsf{denv}}(\mathtt{x}), \mathsf{ReadEffect}(\mathtt{x})), \mathtt{env})}$$

4.2.4 AST-related Notations

When deconstructing AST record nodes such as $\{f_1 : t_2, \ldots, f_k : t_k\}$, we sometimes only care about a subset of the fields $\{f_{i_1}, \ldots, f_{i_m}\} \subset \{f_{1..k}\}$. In such cases, we write $\{f_{i_1} : t_{i_m}, \ldots, f_{i_m} : t_{i_m}, \ldots\}$, where $\ldots$ stands for fields that are irrelevant for the rule.

For example[1], the func non-terminal is of a record type and has the following fields: name, parameters, args, body, return_type, and subprogram_type. The notation {body : SB_ASL(body), args : arg_decls, ...} allows us to deconstruct a given func node by matching only the body and args fields.

Recall that a subset of AST nodes are either labels or labelled tuples. The partial function *ast_label* returns the label $\texttt{l} \in \mathbb{L}$ an AST node, when it exists. For example, *ast_label*(T_Bool) = T_Bool and *ast_label*(T_Named(x)) = T_Named.

4.2.5 How to Parse Rules Efficiently

Consider the following examples, which is a simplified version of SemanticsRule.Binop

$$\frac{\begin{array}{c}\texttt{op} \notin \{\texttt{BAND}, \texttt{BOR}, \texttt{IMPL}\} \\ \texttt{eval\_expr}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m1}, \texttt{env1}) \\ \texttt{eval\_expr}(\texttt{env1}, \texttt{e2}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m2}, \texttt{new\_env}) \\ \texttt{m1} \overset{\text{is}}{=} (\texttt{v1}, \texttt{g1}) \qquad \texttt{m2} \overset{\text{is}}{=} (\texttt{v2}, \texttt{g2}) \qquad \texttt{binop}(\texttt{op}, \texttt{v1}, \texttt{v2}) \xrightarrow{\text{eval}} \texttt{v} \\ \texttt{g} := \texttt{g1} \parallel \texttt{g2}\end{array}}{\texttt{eval\_expr}(\texttt{env}, \text{E\_Binop}(\texttt{op}, \texttt{e1}, \texttt{e2})) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

To parse a rule, start by examining the conclusion and the variables appearing in the rule. In this case, the rule describes a transition from an input configuration eval_expr(env, E_Binop(op, e1, e2)), whose configuration domain is eval_expr, to an output configuration Normal((v, g), new_env) whose configuration domain is Normal. A rule uses the free variables appearing in the input configuration of the conclusion (env, op, e1, and e2 in our example), with the goal of assigning values to the free variables in the output configuration of the conclusion (v, g, and new_env, in our example).

Now, scan the premises in order to see where env, op, e1, and e2 are used and how premises assign values to v, g, and new_env. In this case, v is assigned as the result of the transition assertion $\texttt{binop}(\texttt{op}, \texttt{v1}, \texttt{v2}) \xrightarrow{\text{eval}} \texttt{v}$, g is assigned the expression g1 ‖ g2, and new_env is assigned as the result of the transition assertion $\texttt{eval\_expr}(\texttt{env1}, \texttt{e2}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m2}, \texttt{new\_env})$. Notice that to assign values to the variables v, g, and new_env, intermediate values have to be assigned first. For example, $\texttt{eval\_expr}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m1}, \texttt{env1})$ assigned values to env1, which is then used by the transition $\texttt{eval\_expr}(\texttt{env1}, \texttt{e2}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m2}, \texttt{new\_env})$. Similarly, g is requires first assigning values to g1 and g2, which are components of the previously assigned variables m1 and m2.

[1] This example is from SemanticsRule.FCall.

4.2.6 Short-Circuit Rule Macros

Short-circuit rule macros, or *rule macros*, for short, allow us to succinctly define sets of rules. Specifically, they allow us to capture situations where transitions have two alternative output configurations. If the transition results in the first of the alternative output configurations, the following premises are considered. However, if the result is the second, short-circuit output configuration, then the following premises are ignored and the conclusion transitions into the short-circuit output configuration. These short-circuit output configurations are typically, but not always, due to (type or dynamic) errors.

In the following, XP and XQ stand for, possibly empty, sequences of premises. A rule macro includes the special premise form $C \xrightarrow{R} C' /\!\!/ E$, which introduces alternative output configurations C' and short-circuit E:

$$\frac{\begin{array}{c} XP \\ C \xrightarrow{R} C' /\!\!/ E \\ XQ \end{array}}{V \xrightarrow{R} V'}$$

Such a rule macro expands to the following pair of rules:

(Option 1)
$$\frac{\begin{array}{c} XP \\ C \xrightarrow{R} C' \\ XQ \end{array}}{V \xrightarrow{R} V'}$$

(Option 2:Short-circuited)
$$\frac{\begin{array}{c} XP \\ C \xrightarrow{R} E \end{array}}{V \xrightarrow{R} E}$$

Intuitively, if C transitions to C' then $/\!\!/E$ can be ignored and the rule is interpreted as usual (Option 1). However, if C transitions into E (Option 2) then the premises XQ are ignored, thereby short-circuiting the rule, and the input configuration in the conclusion also transitions into E.

We allow more than one premise to include short-circuiting alternatives and also a single premise to include several alternatives. That is, a rule macro of the form

$$\frac{\begin{array}{c} XP \\ C \xrightarrow{R} C' /\!\!/ E_{1\ldots m} \\ XQ \end{array}}{V \xrightarrow{R} V'}$$

Stands for the set of rule macros

$$\frac{\begin{array}{c} XP \\ C \xrightarrow{R} C' /\!\!/ E_1 \\ XQ \end{array}}{V \xrightarrow{R} V'} \quad \cdots \quad \frac{\begin{array}{c} XP \\ C \xrightarrow{R} C' /\!\!/ E_m \\ XQ \end{array}}{V \xrightarrow{R} V'}$$

Notice that after all rule macros are expanded, in a top-to-bottom and left-to-right order, into normal rules, they behave like normal rules where the order of premises does not matter.

Alternative Outcomes Expressed in English Prose: In English prose, we use $/\!/_{x,y,\ldots}$ to mean "if the outcome is one of $x, y, \ldots$ then the result short-circuits the rule.

As an example, consider the rule SemanticsRule.Binop. This time, not simplified:

$$\frac{\begin{array}{c}\texttt{op} \notin \{\texttt{BAND}, \texttt{BOR}, \texttt{IMPL}\} \\ \texttt{eval\_expr}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m1}, \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{eval\_expr}(\texttt{env1}, \texttt{e2}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m2}, \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{m1} \overset{\text{is}}{=} (\texttt{v1}, \texttt{g1}) \qquad \texttt{m2} \overset{\text{is}}{=} (\texttt{v2}, \texttt{g2}) \qquad \texttt{binop}(\texttt{op}, \texttt{v1}, \texttt{v2}) \xrightarrow{\text{eval}} \texttt{v} \;/\!/\; \texttt{\#DE} \\ \texttt{g} := \texttt{g1} \parallel \texttt{g2}\end{array}}{\texttt{eval\_expr}(\texttt{env}, \text{E\_Binop}(\texttt{op}, \texttt{e1}, \texttt{e2})) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

In this rule, `#T` and `#DE` are just shorthand notations for actual configurations, which are properly defined in the semantics reference. Intuitively, the alternative configurations `#T` and `#DE` represent situations where a transition may result in a raised exception and a dynamic error, respectively.

One may first read the rule ignoring these alternative configurations, to see how the goal of transitioning into the output configuration appearing in the conclusion — $\text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})$ — is achieved. Then, re-reading the rule would indicate where exceptions and dynamic errors may result in other output configurations. For example, if the first transition assertion results in a throwing configuration `#T` then the output configuration of the conclusion is also `#T`. This corresponds to the following rule in the expanded macro:

$$\frac{\begin{array}{c}\texttt{op} \notin \{\texttt{BAND}, \texttt{BOR}, \texttt{IMPL}\} \\ \texttt{eval\_expr}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} \texttt{\#T}\end{array}}{\texttt{eval\_expr}(\texttt{env}, \text{E\_Binop}(\texttt{op}, \texttt{e1}, \texttt{e2})) \xrightarrow{\text{eval}} \texttt{\#T}}$$

Similarly, if the first transition assertion results in a dynamic error, the output configuration of the conclusion is that dynamic error, which corresponds to the following rule in the expansion:

$$\frac{\begin{array}{c}\texttt{op} \notin \{\texttt{BAND}, \texttt{BOR}, \texttt{IMPL}\} \\ \texttt{eval\_expr}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} \texttt{\#DE}\end{array}}{\texttt{eval\_expr}(\texttt{env}, \text{E\_Binop}(\texttt{op}, \texttt{e1}, \texttt{e2})) \xrightarrow{\text{eval}} \texttt{\#DE}}$$

The following rules correspond to the cases where the first transition results in $\text{Normal}(\texttt{m1}, \texttt{env1})$, but the second transition assertion results in either `#T` or `#DE`, respec-

tively:

$$\frac{\begin{array}{c}\texttt{op} \notin \{\texttt{BAND}, \texttt{BOR}, \texttt{IMPL}\} \\ \texttt{eval\_expr}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{m1}, \texttt{env1}) \\ \texttt{eval\_expr}(\texttt{env1}, \texttt{e2}) \xrightarrow{\text{eval}} \texttt{\#T}\end{array}}{\texttt{eval\_expr}(\texttt{env}, \mathsf{E\_Binop}(\texttt{op}, \texttt{e1}, \texttt{e2})) \xrightarrow{\text{eval}} \texttt{\#T}}$$

$$\frac{\begin{array}{c}\texttt{op} \notin \{\texttt{BAND}, \texttt{BOR}, \texttt{IMPL}\} \\ \texttt{eval\_expr}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{m1}, \texttt{env1}) \\ \texttt{eval\_expr}(\texttt{env1}, \texttt{e2}) \xrightarrow{\text{eval}} \texttt{\#DE}\end{array}}{\texttt{eval\_expr}(\texttt{env}, \mathsf{E\_Binop}(\texttt{op}, \texttt{e1}, \texttt{e2})) \xrightarrow{\text{eval}} \texttt{\#DE}}$$

Expanding the last transition assertion, gives us the case:

$$\frac{\begin{array}{c}\texttt{op} \notin \{\texttt{BAND}, \texttt{BOR}, \texttt{IMPL}\} \\ \texttt{eval\_expr}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{m1}, \texttt{env1}) \\ \texttt{eval\_expr}(\texttt{env1}, \texttt{e2}) \xrightarrow{\text{eval}} \mathsf{Normal}(\texttt{m2}, \texttt{new\_env}) \\ \texttt{m1} \overset{\text{is}}{=} (\texttt{v1}, \texttt{g1}) \qquad \texttt{m2} \overset{\text{is}}{=} (\texttt{v2}, \texttt{g2}) \qquad \texttt{binop}(\texttt{op}, \texttt{v1}, \texttt{v2}) \xrightarrow{\text{eval}} \texttt{\#DE}\end{array}}{\texttt{eval\_expr}(\texttt{env}, \mathsf{E\_Binop}(\texttt{op}, \texttt{e1}, \texttt{e2})) \xrightarrow{\text{eval}} \texttt{\#DE}}$$

All these cases are succinctly encoded in a single rule with the alternative output configurations.

4.2.7 Boolean Transition Assertions

We define the following rules to allow us to treat assertions as transition assertions:

BOOL_TRANS_TRUE
$$\textit{bool\_transition}(\texttt{TRUE}) \longrightarrow \texttt{TRUE}$$

BOOL_TRANS_FALSE
$$\textit{bool\_transition}(\texttt{FALSE}) \longrightarrow \texttt{FALSE}$$

This is useful in that it allows us to use assertions in rule macros.

4.2.8 Rule Naming

To name a rule, we place it in a section with its name. However, some relations are defined by a group of rules. In such cases, we refer to the individual rules in a group as *case rules*, or simply *cases*. We annotate case rules by names appearing above and to the left of the rule. The name of these case rules is the name of the group, given by its section, followed by the name of the case.

For example, the ASL Semantics Reference defines the rule SemanticsRule.BaseValue using 11 cases of which two are the following:

BOOL
$$\frac{\textit{get\_structure}(\texttt{t}) \xrightarrow{\text{type}} \mathsf{T\_Bool}}{\texttt{base\_value}(\texttt{env}, \texttt{t}) \xrightarrow{\text{eval}} (\texttt{Bool}(\texttt{TRUE}), \emptyset_{\texttt{g}})}$$

REAL
$$\frac{\textit{get\_structure}(\texttt{t}) \xrightarrow{\text{type}} \mathsf{T\_Real}}{\texttt{base\_value}(\texttt{env}, \texttt{t}) \xrightarrow{\text{eval}} (\texttt{Real}(0), \emptyset_{\texttt{g}})}$$

The full name of the first case is then SemanticsRule.BaseValue.BOOL and the full name of the second case is SemanticsRule.BaseValue.REAL.

When explaining rules in English prose, we include the name of the case rules in parenthesis to make it easier to relate the prose to the corresponding mathematical definitions (see, for example, the Prose paragraph of SemanticsRule.BaseValue or that of TypingRule.CheckUnop).

4.2.9 Generic Notations

- The notation ↪ denotes that a line that is longer than the page width continues on the next line.
- The notation ******** common prefix ******** serves as a visual aid to delimit a common prefix of premises shared by rule cases.
- The notation ******** common suffix ******** serves as a visual aid to delimit a common suffix of premises shared by rule cases.
- **Missing definition:** Red hyperlinks indicate items that are yet to be defined.

Chapter 5

ASL Lexical Definition

This chapter defines the various elements of an ASL specification text in a high-level way and then formalizes the lexical analysis as a function that takes a text and returns a list of *tokens* or a lexical error.

5.1 ASL Specification Text

An ASL specification is a string — a list of ASCII characters — consisting of a *content text* followed by an *end-of-file*. The content text is a list of ASCII characters that have the decimal encoding of 32 through 126 (inclusive), which includes the space character (decimal encoding 32), as well as carriage return (decimal encoding 13) and line feed (decimal encoding 10). The end of file character is denote by eof. The content text does not contain an end-of-file character.

In particular, it is an error to use a tab character in ASL specification text (decimal encoding 9).

5.2 Lexical Regular Expressions

Table 5.1 defines the regular expressions RegExp used to define *lexemes* — substrings of the ASL specification text that are used to form *tokens*. We use the notation c for characters such as the apostrophe.

Let <ascii_char> stand for any ASCII character:

$$\texttt{<ascii\_char>} \triangleq \texttt{ASCII\{0-255\}}$$

Let <char> stand for an ASCII character that may appear in the content text:

$$\texttt{<char>} \triangleq \texttt{ASCII\{10\}} \mid \texttt{ASCII\{13\}} \mid \texttt{ASCII\{32-126\}}$$

The notation Lang(e) stands for *formal language* of a regular expression e. That is, the set of strings that match that regular expression.

Table 5.1: Lexical Regular Expressions

RegExp	Matches
`a_string`	Any character in `a_string`
□	The space character (decimal 32)
ASCII{a}	The ASCII with decimal 'a'
ASCII{a-b}	The ASCII range between decimals 'a' and 'b'
(*A*)	*A*
A B	*A* followed by *B*
A | *B*	A or B
A - *B*	*A* but not *B*
A∗	Zero or more repetitions of A
A+	One or more repetitions of A
`"a_string"`	The string `a_string` verbatim
<r>	The lexical regular expression defined for <r>

5.3 Whitespace

Comments, newlines and space characters are treated as whitespace.

5.4 Comments

ASL supports comments in the style of C++:

- Single-line comments: the text from // until the end of the line is a comment (ASCII{10} is the line feed character `\n`).
- Multi-line comments: the text between `/*` and `*/` is a comment.

Comments do not nest and the two styles of comments do not interact with each other.

<line_comment> ≜ "//" (<char> - ASCII{10})* | "/*" <char>* "*/"

5.5 Integer Literals

Integers are written either in decimal using one or more of the characters `0-9` and underscore, or in hexadecimal using `0x` at the start followed by the characters `0-9, a-f, A-F` and underscore. An integer literal cannot start with an underscore.

This is formalized by the following lexical regular expression:

<digit> ≜ 0123456789
<int_lit> ≜ <digit> (□ | <digit>)*
<hex_lit> ≜ 0x (<digit> | abcdefABCDEF) (_ | <digit> | abcdefABCDEF)*

5.6 Real Number Literals

Real numbers are written in decimal and consist of one or more decimal digits, a decimal point and one or more decimal digits. Underscores can be added between digits to aid readability

Underscores in numbers are not significant, and their only purpose is to separate groups of digits to make constants such as `0xefff_fffe`, `1_000_000` or `3.141_592_654` easier to read,

This is formalized by the following lexical regular expression:

<real_lit> ≜ <digit> (_ | <digit>)* '.' <digit> (_ | <digit>)*

5.7 Boolean Literals

Boolean literals are written using `TRUE` or `FALSE`.

5.8 Bitvector Literals

Constant bit-vectors are written using 1, 0 and spaces surrounded by single-quotes.

<bitvector_lit> ≜ ' (01␣)* '

The spaces in a bitvector are not significant and are only used to improve readability. For example, `'1111 1111 1111 1111'` is the same as `'1111111111111111'`.

5.9 Bitmasks

Constant bitmasks are written using `1`, `0`, `x` and spaces surrounded by single-quotes. The `x` represents a don't care character.

<bitmask_lit> ≜ ' (01x␣)* '

The spaces in a constant bitmask are not significant and are only used to improve readability.

5.10 String Literals

String literals consist of printable characters surrounded by double quotes. They are used to create string values, which are strings of zero or more characters, where a character is a printable ASCII character, tab (ASCII code `10`), newline (ASCII code `10`), the backslash character (ASCII code `92`), and double-quote character (ASCII code `34`). Unprintable characters (tabs and newlines) are not permitted in string literals, so they are represented by treating the backslash character \, as an escape character. Note therefore that string literals cannot span multiple source lines.

The escape sequences allowed in string literals appear in Table 5.2.

Table 5.2: Escape Sequences in String Literals

Escape sequence	Meaning
`\n`	The newline, ASCII code `10`
`\t`	The tab, ASCII code `9`
`\\`	The backslash character, \, ASCII code `92`
`\"`	The double-quote character, ", ASCII code `34`

```
<str_char>    ≜  ASCII{32-126}
<string_lit>  ≜  " ( (<str_char> - " \ ) | (\ " n t \ ) )* "
```

5.11 Identifiers

Identifiers start with a letter or underscore and continue with zero or more letters, underscores or digits. Identifiers are case sensitive. To improve readability, it is recommended to avoid the use of identifiers that differ only by the case of some characters.

By convention, identifiers that begin with double-underscore are reserved for use in the implementation and should not be used in specifications.

```
<letter>      ≜  'a-z' | 'A-Z'
<identifier>  ≜  (<letter> | _) (<letter> | _ | <digit>)*
```

Tuple element selectors are classed as identifiers. That is, in cases like `(1, 2).item0`, the selector `item0` is classed as an identifier.

5.12 Lexical Analysis

Lexical analysis, which is also referred to as *scanning*, is defined via the function

$$scan : \mathsf{LexSpec} \times \langle \mathsf{ascii\_char} \rangle^* \longrightarrow (\mathbb{TOKEN}^* \cup \{\#\mathsf{LE}\})$$

which takes a *lexical specification* (explained soon), an ASL specification string (where characters are simply numbers representing ASCII characters) and returns a sequence of tokens (tokens are defined below) or a *lexical error* #LE.

Tokens have one of two forms:

Value-carrying Tokens that carry value have the form $L(v)$ where L is a token label, signifying the meaning of the token, and v is a value carried by the token, which is used to construct the respective Abstract Syntax Tree nodes.

Valueless Tokens that do not carry values have the form L where L is a token label.

The set of tokens used for the lexical analysis of ASL strings is defined below.

$$
\begin{array}{lll}
\mathbb{TOKEN} \triangleq & \{\ \textbf{INT\_LIT}(n) \mid n \in \mathbb{Z}\ \} & \cup \\
& \{\ \textbf{REAL\_LIT}(q) \mid q \in \mathbb{Q}\ \} & \cup \\
& \{\ \textbf{STRING\_LIT}(s) \mid s \in \mathsf{Lang}(\texttt{<string\_lit>})\} & \cup \\
& \{\ \textbf{STRING\_CHAR}(c) \mid c \in \mathsf{Lang}(\texttt{<char>})\} & \cup \\
& \{\ \textbf{STRING\_END}\} & \cup \\
& \{\ \textbf{BITVECTOR\_LIT}(b) \mid b \in \{0,1\}^*\ \} & \cup \\
& \{\ \textbf{MASK\_LIT}(m) \mid m \in \{0,1,x\}^*\ \} & \cup \\
& \{\ \textbf{BOOL\_LIT}(\texttt{TRUE}), \textbf{BOOL\_LIT}(\texttt{FALSE})\ \} & \cup \\
& \{\ \textbf{ID}(\texttt{id}) \mid \texttt{id} \in \mathsf{Lang}(\texttt{<identifier>})\ \} & \cup \\
& \{\ \textbf{LEXEME}(s) \mid s \in \mathbb{S}\ \} & \cup \\
& \{\ \mathsf{WHITE\_SPACE}, \mathsf{EOF}, \textbf{T\_ERR}\ \} &
\end{array}
$$

- Tokens of the form **INT_LIT**(n) represent integer literals;
- Tokens of the form **REAL_LIT**(q) represent real literals;
- Tokens of the form **STRING_LIT**(s) represent string literals;
- Tokens of the form **STRING_CHAR**(c) represent a single character in a string literal;
- The token **STRING_END** represents the closing quotes of a string literal;
- Tokens of the form **BITVECTOR_LIT**(b) represent bitvector literals;
- Tokens of the form **MASK_LIT**(m) represent constant bitmasks;
- Tokens of the form **BOOL_LIT**(b) represent Boolean literals;
- Tokens of the form **ID**(i) represent identifiers;
- Tokens with the label **LEXEME** are ones where the value s is simply the *lexeme* for that token. That is, the substring representing that token. Later we will refer to such token by simply quoting the lexeme of the token and dropping the label, for brevity. For example, instead of **LEXEME**(`for`), we will write `"for"`.
- The valueless token WHITE_SPACE represents white spaces;
- The valueless token **T_ERR** represents an illegal lexeme such as the use of a reserved keyword;
- The valueless token EOF represents eof.

Definition 25 (Lexical Specification) *A* lexical specification *consists of a list of pairs* $[(r_1, a_1), \ldots, (r_k, a_k)] \in \mathit{LexSpec}$ *where each pair* (r_i, a_i) *consists of a lexical regular expression* r_i *and a* lexeme action $a_i : \mathbb{S} \times \mathbb{S} \longrightarrow \mathbb{TOKEN}^*$.

The function

$$re\_max\_match : \overbrace{\mathsf{RegExp}}^{e} \times \overbrace{\mathbb{S}}^{s} \longrightarrow (\overbrace{\mathbb{S}}^{s_1} \times \overbrace{\mathbb{S}}^{s_2}) \cup \{\bot\}$$

returns the *longest* match of a regular expression e for a prefix of a string s. More precisely: $re\_max\_match(e, s) = (s_1, s_2)$ means that $s_1 \in \mathsf{Lang}(e)$ and $s = s_1 + s_2$. If no match exists, it is indicated by returning $\bot$.

The function $max\_matches : \overbrace{\mathsf{LexSpec}}^{R} \times \overbrace{\mathbb{S}}^{s} \longrightarrow \overbrace{\mathsf{LexSpec}}^{R'}$ returns the sublist of R consisting of pairs whose maximal matches for s are equal. Importantly, the result sublist R' maintains the order of pairs in R. If all expressions in R do not match (that is re_max_match returns $\bot$ for all pairs in R), then R' is the empty list.

The function $scan$ is constructively defined via the following inference rules:

$$\text{NO\_MATCH}\quad \frac{max\_matches(R, s) = [\,]}{scan(R, s) \xrightarrow{\mathsf{scan}} \#\mathsf{LE}}$$

$$\text{TOKEN}\quad \frac{\begin{array}{c} max\_matches(R, s) = [(e_1, a_1), \ldots, (e_n, a_n)] \\ re\_max\_match(s, e_1) = (s_1, s_2) \qquad a_1(s_1, s_2) \xrightarrow{\mathsf{scan}} ts \mathbin{/\!/} \#\mathsf{LE} \end{array}}{scan(R, s) \xrightarrow{\mathsf{scan}} ts}$$

This form of scanning is referred to as "Maximal Munch" in Compiler Theory and is the most common form of scanning. See "Compilers: Principles, Techniques, and Tools" [**?**] for more details.

While Maximal Munch is a useful policy for scanning of most tokens, it does not work well for string literals and multi-line comments, which require identifying the respective tokens via shortest match. For this purpose, most lexical analyzers split the analysis into separate "states" — one for keywords, symbols, single-line comments, and identifiers, one for string literals, and one for multi-line comments. The lexical analyzers switches between the states as needed, and analyzing string literals involves concatenating the individual characters of the string literal into a single token.

Lexical analysis of ASL follows this approach by defining three specifications:

- SPEC_TOKEN: For keywords, symbols, single-line comments, and identifiers;
- SPEC_COMMENT: For multi-line comments;
- SPEC_STRING: For string literals.

Additionally, lexical analysis of string literals carries the extra state — the string characters encountered along the way.

We now define each of the lexical specifications and related lexeme actions.

Each lexical specification is depicted by a table where the order of elements of a specification corresponds to the order of rows in the table.

5.12.1 Scanning Regular Tokens

To scan keywords, symbols, single-line comments, and identifiers, we define the following lexeme actions:

- The lexeme action

$$\mathit{discard}(s_1, s_2) \triangleq \mathit{scan}(\text{SPEC\_TOKEN}, s_2)$$

  discards the string s_1 and continues scanning s_2 with SPEC_TOKEN. This is used for whitespace.

- The lexeme action

$$\mathit{return\_token}(f) \triangleq \lambda(s_1, s_2).$$
$$\hookrightarrow \begin{cases} \#\mathsf{LE} & \text{if } f(s_1) = \mathbf{T\_ERR} \text{ or} \\ & \mathit{scan}(\text{SPEC\_TOKEN}, s_2) = \mathbf{T\_ERR} \\ [f(s_1)] + \mathit{scan}(\text{SPEC\_TOKEN}, s_2) & \text{else} \end{cases}$$

  is parameterized by a function f that converts strings into corresponding tokens. It applies f to convert s_1 into a token and then continues scanning s_2 with SPEC_TOKEN. If at any point a lexical error is encountered, the entire result is a lexical error.

- The lexeme action

$$\mathit{start\_string}(s_1, s_2) \triangleq \mathit{scan\_string}([\,], s_2)$$

  switches to scanning literal strings via *scan_string*.

- The lexeme action

$$\mathit{start\_comment}(s_1, s_2) \triangleq \mathit{scan}(\text{SPEC\_COMMENT}, s_2)$$

  switches to scanning multi-line comments by changing the lexical specification to SPEC_COMMENT.

- The function $\mathit{dec\_to\_lit}(s)$ returns $\mathbf{INT\_LIT}(n)$ where n is the integer represented by s by decimal representation.

- The function $\mathit{hex\_to\_lit}(s)$ returns $\mathbf{INT\_LIT}(n)$ where n is the integer represented by s by hexadecimal representation.

- The function $\mathit{real\_to\_lit}(s)$ returns $\mathbf{REAL\_LIT}(q)$ where q is the real value represented by s by floating point representation.

- The function $\mathit{str\_to\_lit}(s)$ returns $\mathbf{STRING\_LIT}(s')$ where s' is the string value represented by s.

- The function *bits_to_lit*(s) returns **BITVECTOR_LIT**(b) where b is the sequence of bits given by s.
- The function *mask_to_lit*(s) returns **MASK_LIT**(m) where m is the bitmask given by s.
- The function *false_to_lit*(s) returns **BOOL_LIT**(`FALSE`) (s is ensured to be `FALSE`).
- The function *true_to_lit*(s) returns **BOOL_LIT**(`TRUE`) (s is ensured to be `TRUE`).
- The function *token_id*(s) returns **LEXEME**(s).
- The function *lexical_error* returns **T_ERR**.
- The function *to_identifier*(s) returns **ID**(s).
- The lexeme action

$$\mathit{eof\_token}(s_1, s_2) \triangleq \begin{cases} [\,] & s_2 = [\,] \\ \#\mathsf{LE} & \text{else} \end{cases}$$

  checks whether eof is not followed by more characters and returns a lexical error otherwise.

The lexical specification SPEC_TOKEN is given by the following four tables. Splitting the lexical specification into four tables is done for presentation purposes — the ordering between the entries is induced by the order between the tables and the order of entries in each table. When several regular expressions are listed in a row, it means that they are all associated with the same token function.

Lexical Regular Expressions	**Lexeme Action**
(ASCII{10} | ASCII{13} | ASCII{32})+	*discard*
`"/*"`	*start_comment*
`"` ␣	*start_string*
<int_lit>	*return_token*(*dec_to_lit*)
<hex_lit>	*return_token*(*hex_to_lit*)
<real_lit>	*return_token*(*real_to_lit*)
<string_lit>	*return_token*(*str_to_lit*)
<bitvector_lit>	*return_token*(*bits_to_lit*)
<bitmask_lit>	*return_token*(*mask_to_lit*)
`'!', ',', '<', ">>", "&&", "-->", "<<"`	*return_token*(*token_id*)
`']', ')', "..", '=', '{', "!=", '-', "<->"`	*return_token*(*token_id*)
`'[', '(', '.', "<=", '^', '*', '/'`	*return_token*(*token_id*)
`"==", "||", '+', ':', "=>",`	*return_token*(*token_id*)
`'}', "++", '>', "+:", "*:", ';', ">="`	*return_token*(*token_id*)
`"@looplimit"`	*return_token*(*token_id*)

Lexical Regular Expressions	**Lexeme Action**
`"AND"`, `"array"`, `"as"`, `"assert"`,	*return_token*(*token_id*)
`"begin"`, `"bit"`, `"bits"`, `"boolean"`	*return_token*(*token_id*)
`"case"`, `"catch"`, `"config"`, `"constant"`	*return_token*(*token_id*)
`"DIV"`, `"DIVRM"`, `"do"`,`"downto"`	*return_token*(*token_id*)
`"else"`, `"elsif"`, `"end"`, `"enumeration"`	*return_token*(*token_id*)
`"XOR"`	*return_token*(*token_id*)
`"exception"`	*return_token*(*token_id*)
`"FALSE"`	*return_token*(*false_to_lit*)
`"for"`, `"func"`	*return_token*(*token_id*)
`"getter"`	*return_token*(*token_id*)
`"if"`, `"IN"`, `"integer"`	*return_token*(*token_id*)
`"let"`	*return_token*(*token_id*)
`"MOD"`	*return_token*(*token_id*)
`"NOT"`	*return_token*(*token_id*)
`"of"`, `"OR"`, `"otherwise"`	*return_token*(*token_id*)
`"pass"`, `"pragma"`, `"print"`	*return_token*(*token_id*)
`"real"`, `"record"`, `"repeat"`, `"return"`	*return_token*(*token_id*)
`"setter"`, `"string"`, `"subtypes"`	*return_token*(*token_id*)
`"then"`, `"throw"`, `"to"`, `"try"`	*return_token*(*token_id*)
`"TRUE"`	*return_token*(*true_to_lit*)
`"type"`	*return_token*(*token_id*)
`"UNKNOWN"`, `"until"`	*return_token*(*token_id*)
`"var"`	*return_token*(*token_id*)
`"when"`, `"where"`, `"while"`, `"with"`	*return_token*(*token_id*)

The following list represents keywords that are reserved for future use.

Lexical Regular Expressions	Lexeme Action
"SAMPLE", "UNSTABLE"	*lexical_error*
"_", "access", "advice", "after"	*lexical_error*
"any", "aspect"	*lexical_error*
"assume", "assumes", "before"	*lexical_error*
"call", "cast"	*lexical_error*
"class", "dict"	*lexical_error*
"endcase", "endcatch", "endclass"	*lexical_error*
"endevent", "endfor", "endfunc", "endgetter"	*lexical_error*
"endif", "endmodule", "endnamespace", "endpackage"	*lexical_error*
"endproperty", "endrule", "endsetter", "endtemplate"	*lexical_error*
"endtry", "endwhile", "entry"	*lexical_error*
"event", "export", "expression"	*lexical_error*
"extends", "extern", "feature"	*lexical_error*
"get", "gives"	*lexical_error*
"iff", "implies", "import"	*lexical_error*
"intersect", "intrinsic"	*lexical_error*
"invariant", "is", "list"	*lexical_error*
"map", "module", "namespace", "newevent"	*lexical_error*
"newmap", "original"	*lexical_error*
"package", "parallel"	*lexical_error*
"pointcut", "port", "private"	*lexical_error*
"profile", "property", "protected", "public"	*lexical_error*
"replace"	*lexical_error*
"requires", "rethrow", "rule"	*lexical_error*
"set", "shared", "signal"	*lexical_error*
"statements", "template"	*lexical_error*
"typeof", "union"	*lexical_error*
"using", "watch"	*lexical_error*
"ztype"	*lexical_error*

Lexical Regular Expression	Lexeme Action
<identifier>	*return_token*(*to_identifier*)
eof	*eof_token*

5.12.2 Scanning Strings

To scan string literals, we define the following specialized scanning function. The function

$$scan\_string : \overbrace{\langle ascii\_char \rangle^*}^{buf} \times \overbrace{\langle ascii\_char \rangle^*}^{s} \longrightarrow (\mathbb{TOKEN}^* \cup \{\#LE\})$$

scans string with the SPEC_STRING specification while building the final string literal in `buf`. It is defined via the following rules:

$$\text{NO\_MATCH}\quad \frac{\mathit{max\_matches}(\text{SPEC\_STRING}, s) = [\,]}{\mathit{scan\_string}(\texttt{buf}, s) \xrightarrow{\text{scan}} \#\text{LE}}$$

$$\text{CHAR}\quad \frac{\mathit{max\_matches}(R, s) = [(e_1, a_1), \ldots, (e_n, a_n)] \qquad \mathit{re\_max\_match}(s, e_1) = (s_1, s_2) \\ a_1(s_1, s_2) = \textbf{STRING\_CHAR}(t) \qquad \mathit{scan\_string}(\texttt{buf} + t, s_2) \xrightarrow{\text{scan}} \mathit{ts2} \mathbin{/\!\!/} \#\text{LE}}{\mathit{scan\_string}(\texttt{buf}, s) \xrightarrow{\text{scan}} \mathit{ts2}}$$

$$\text{END}\quad \frac{\mathit{max\_matches}(R, s) = [(e_1, a_1), \ldots, (e_n, a_n)] \qquad \mathit{re\_max\_match}(s, e_1) = (s_1, s_2) \\ a_1(s_1, s_2) = \textbf{STRING\_END} \qquad \mathit{scan}(\text{SPEC\_TOKEN}, s_2) \xrightarrow{\text{scan}} \mathit{ts2} \mathbin{/\!\!/} \#\text{LE}}{\mathit{scan\_string}(\texttt{buf}, s) \xrightarrow{\text{scan}} [\textbf{STRING\_LIT}(\texttt{buf})] + \mathit{ts2}}$$

We also employ the following lexeme actions:

- The lexeme action

$$\mathit{string\_char}(s_1, s_2) \triangleq \textbf{STRING\_CHAR}(s_1)$$

  returns s_1, which is always a single character, as a **STRING_CHAR** token, which is added to the characters that make up the final string literal.

- The lexeme action

$$\mathit{string\_escape}(s_1, s_2) \triangleq \begin{cases} \textbf{STRING\_CHAR}(10) & s_1 = \backslash\ \texttt{n} \\ \textbf{STRING\_CHAR}(9) & s_1 = \backslash\ \texttt{t} \\ \textbf{STRING\_CHAR}(34) & s_1 = \backslash\ \texttt{"} \\ \textbf{STRING\_CHAR}(92) & s_1 = \backslash\ \backslash \end{cases}$$

  returns the ASCII character for the corresponding escape string, in decimal encoding, as a **STRING_CHAR** token, which is added to the characters that make up the final string literal.

- The lexeme action

$$\mathit{string\_finish}(s_1, s_2) \triangleq \textbf{STRING\_END}$$

  signals that the string literal has ended, which makes *scan_string* switch to scanning via *scan* and SPEC_TOKEN.

The lexical specification for string literals — SPEC_STRING — is given by the following table:

Lexical Regular Expression	Lexeme Action
`\n`	*string_escape*
`\t`	*string_escape*
`\"`	*string_escape*
`\\`	*string_escape*
`"`	*string_finish*
<char>	*string_char*

5.12.3 Scanning Multi-line Comments

The lexeme action

$$\mathit{discard\_comment\_char}(s_1, s_2) \triangleq \mathit{scan}(\text{SPEC\_TOKEN}, s_2)$$

discards the string s_1 (which is always a single character) and continues scanning s_2 with SPEC_COMMENT. This is the same as *discard*, except that s_2 is scanned with SPEC_COMMENT instead of SPEC_TOKEN.

The lexical specification for multi-line comments — SPEC_COMMENT — is given by the table below. Notice that here, *discard* below is used to discard the closing of the multi-line comment and to switch to scanning with SPEC_TOKEN.

Lexical Regular Expression	Lexeme Action
`"*/"`	*discard*
<char>	*discard_comment_char*

Chapter 6

ASL Concrete Syntax

This chapter defines the grammar of ASL. The grammar is presented via two extensions to context-free grammars — *inlined derivations* and *parametric productions*, inspired by the Menhir Parser Generator [**?**] for the OCaml language. Those extensions can be viewed as macros over context-free grammars, which can be expanded to yield a standard context-free grammar.

Our definition of the grammar and description of the parsing mechanism heavily relies on the theory of parsing via LR(1) grammars and LR(1) parser generators. See "Compilers: Principles, Techniques, and Tools" [**?**] for a detailed definition of LR(1) grammars and parser construction.

The expanded context-free grammar is an LR(1) grammar, modulo shift-reduce conflicts that are resolved via appropriate precedence definitions. That is, given a list of token, returned from *scan*, it is possible to apply an LR(1) parser to obtain a parse tree if the list of tokens is in the formal language of the grammar and return a parse error otherwise.

The outline of this chapter is as follows:

- Definition of inlined derivations (see Section 6.1)
- Definition of parametric productions (see Section 6.2)
- ASL Parametric Productions (see Section 6.3)
- Definition of the ASL grammar (see Section 6.4)
- Definition of parse trees (see Section 6.5)
- Definition of priority and associativity of operators (see Section 6.6)

6.1 Inlined Derivations

Context-free grammars consist of a list of *derivations* $N \longrightarrow S^*$ where N is a non-terminal symbol and S is a list of non-terminal symbols and terminal symbols, which correspond

to tokens. We refer to a list of such symbols as a *sentence*. A special form of a sentence is the *empty sentence*, written ϵ.

As commonly done, we aggregate all derivations associated with the same non-terminal symbol by writing $N \longrightarrow R_1 \mid \ldots \mid R_k$. We refer to the right-hand-side sentences $R_{1..k}$ as the *alternatives* of N.

Our grammar contains another form of derivation — *inlined derivation* — written as $N \xrightarrow{\text{inline}} R_1 \mid \ldots \mid R_k$. Expanding an inlined derivation consists of replacing each instance of N in a right-hand-side sentence of a derivation with each of $R_{1..k}$, thereby creating k variations of it (and removing $N \xrightarrow{\text{inline}} R_1 \mid \ldots \mid R_k$ from the set of derivations).

For example, consider the derivation

```
expr ⟶ expr binop expr
```

coupled with the derivation

```
binop ⟶ "AND" | "&&" | "||" | "<->" | "DIV" | "DIVRM" | "XOR" | "==" | "!="
      | ">" | ">=" | "-->" | "<" | "<=" | "+" | "-" | "MOD" | "*"
      | "OR" | "RDIV" | "<<" | ">>" | "^" | "++"
```

A grammar containing these two derivations results in shift-reduce conflicts. Resolving these conflicts is done by associating priority levels to each of the binary operators and creating a version of the first derivation for each binary operator:

```
expr ⟶ expr "AND" expr
     | expr "&&" expr
     | expr "||" expr
     ...
     | expr "++" expr
```

By defining the derivations of `binop` as inlined, we achieve the same effect more compactly:

```
binop -inline-> "AND" | "&&" | "||" | "<->" | "DIV" | "DIVRM" | "XOR" | "==" | "!="
      | ">" | ">=" | "-->" | "<" | "<=" | "+" | "-" | "MOD" | "*"
      | "OR" | "RDIV" | "<<" | ">>" | "^" | "++"
```

Barring mutually-recursive derivations involving inlined derivations, it is possible to expand all inlined derivations to obtain a context-free grammar without any inlined derivations.

6.2 Parametric Productions

A parametric production has the form $N(p_{1..m}) \longrightarrow R_1 \mid \ldots \mid R_k$ where $p_{1..m}$ are place holders for grammar symbols and may appear in any of the alternatives $R_{1..k}$. We refer to $N(p_{1..m})$ as a *parametric non-terminal*.

Given sentences $S_{1..m}$, we can expand $N(p_{1..m}) \longrightarrow R_1 \mid \ldots \mid R_k$ by creating a unique symbols for $N(p_{1..m})$, denoted as $\mathsf{unique}(N(S_{1..m}))$, defining the derivations

$$\mathsf{unique}(N(S_{1..m})) \longrightarrow R_1[S_1/p_1, \ldots, S_m/p_m] \mid \ldots \mid R_k[S_1/p_1, \ldots, S_m/p_m]$$

where for each $i = 1..k$, $R_i[S_1/p_1, \ldots, S_m/p_m]$ means replacing each instance of p_j with S_j, for each $j = 1..m$. Then, each instance of $S_{1..m}$ in the grammar is replaced by $\mathsf{unique}(N(S_{1..m}))$. If all instances of a parametric non-terminal are expanded this way, we can remove the derivations of the parametric non-terminal altogether.

We note that a parametric production can be either a normal derivation or an inlined derivation.

For example, the derivation for a list of ASL global declarations is as follows:

$\texttt{ast} \longrightarrow \mathsf{list}^*(\texttt{decl})$

It is defined via the parametric production for possibly-empty lists:

$\mathsf{list}^*(x) \longrightarrow \epsilon \mid x\ \mathsf{list}^*(x)$

Expanding $\mathsf{list}^*(\texttt{decl})$ produces the following derivations for a new unique symbol. That is, a symbol that does not appear anywhere else in the grammar. In this example we will choose $\mathsf{unique}(\mathsf{list}^*(\texttt{decl}))$ the be the symbol `decl_list`. The result of the expansion is then:

$\texttt{decl\_list} \longrightarrow \epsilon \mid \texttt{decl}\ \texttt{decl\_list}$

The new symbol is substituted anywhere $\mathsf{list}^*(\texttt{decl})$ appears in the original grammar, which results in the following derivation replacing the original derivation for `ast`:

$\texttt{ast} \longrightarrow \texttt{decl\_list}$

Expanding all instances of parametric productions results in a grammar without any parametric productions.

6.3 ASL Parametric Productions

We define the following parametric productions for various types of lists and optional productions.

Optional Symbol

$\mathsf{option}(x) \longrightarrow \epsilon \mid x$

Possibly-empty List

$\mathsf{list}^*(x) \longrightarrow \epsilon \mid x\ \mathsf{list}^*(x)$

Non-empty List

$\mathsf{list}^+(x) \longrightarrow x \mid x\ \mathsf{list}^+(x)$

Non-empty Comma-separated List

$\mathsf{clist}^+(x) \longrightarrow x \mid x\ \texttt{","}\ \mathsf{clist}^+(x)$

Possibly-empty Comma-separated List

$\mathsf{clist}^*(x) \xrightarrow{\mathsf{inline}} \epsilon \mid \mathsf{clist}^+(x)$

Comma-separated List With At Least Two Elements

$\mathsf{clist2}(x) \xrightarrow{\mathsf{inline}} x\ \texttt{","}\ \mathsf{clist}^+(x)$

Possibly-empty Parenthesized, Comma-separated List

$\mathsf{plist}^*(x) \xrightarrow{\mathsf{inline}} \texttt{"("}\ \mathsf{clist}^*(x)\ \texttt{")"}$

Parenthesized Comma-separated List With At Least Two Elements

$\mathsf{plist2}(x) \xrightarrow{\mathsf{inline}} \texttt{"("}\ x\ \texttt{","}\ \mathsf{clist}^+(x)\ \texttt{")"}$

Non-empty Comma-separated Trailing List

$$\begin{aligned}\mathsf{ntclist}(x) \longrightarrow\ & x\ \mathsf{option}(\texttt{","}) \\ \mid\ & x\ \texttt{","}\ \mathsf{ntclist}(x)\end{aligned}$$

Comma-separated Trailing List

$\mathsf{tclist}^*(x) \xrightarrow{\mathsf{inline}} \mathsf{option}(\mathsf{ntclist}(x))$

6.4 ASL Grammar

We now present the list of derivations for the ASL Grammar where the start non-terminal is `ast`. The derivations allow certain parse trees where lists may have invalid sizes. Those parse trees must be rejected in a later phase.

Notice that two of the derivations (for `expr_pattern` and for `expr`) end with precedence: **UNOPS**. This is a precedence annotation, which is not part of the right-hand-side sentence, and is explained in Section 6.6 and can be ignored upon first reading.

For brevity, tokens are presented via their label only, dropping their associated value. For example, instead of **ID**(`id`), we simply write **ID**.

```
ast ⟶ list*(decl)
```

```
decl ⟶[inline] "func" ID params_opt func_args return_type func_body
               | "func" ID params_opt func_args func_body
               | "getter" ID params_opt access_args return_type func_body
               | "getter" ID return_type func_body
               | "setter" ID params_opt access_args "=" typed_identifier
                ↪ func_body
               | "setter" ID "=" typed_identifier func_body
               | "type" ID "of" ty_decl subtype_opt ";"
               | "type" ID subtype ";"
               | storage_keyword ignored_or_identifier option(":" ty) "="
                ↪ expr ";"
               | "var" ignored_or_identifier ":" ty ";"
               | "pragma" ID clist*(expr) ";"
```

```
subtype ⟶[inline] "subtypes" ID "with" fields
                  | "subtypes" ID
```

```
subtype_opt ⟶[inline] option(subtype)
```

```
typed_identifier ⟶[inline] ID as_ty
```

opt_typed_identifier $\xrightarrow{\text{inline}}$ **ID** option(as_ty)

as_ty $\xrightarrow{\text{inline}}$ ":" ty

return_type $\xrightarrow{\text{inline}}$ "=>" ty

params_opt $\xrightarrow{\text{inline}}$ ϵ
| "{" clist*(opt_typed_identifier) "}"

access_args $\xrightarrow{\text{inline}}$ "[" clist*(typed_identifier) "]"

func_args $\xrightarrow{\text{inline}}$ "(" clist*(typed_identifier) ")"

maybe_empty_stmt_list $\xrightarrow{\text{inline}}$ ϵ | stmt_list

func_body $\xrightarrow{\text{inline}}$ "begin" maybe_empty_stmt_list "end"

ignored_or_identifier $\xrightarrow{\text{inline}}$ "-" | **ID**

Parsing note: "var" is not derived by local_decl_keyword to avoid an LR(1) conflict.

local_decl_keyword $\xrightarrow{\text{inline}}$ "let" | "constant"

storage_keyword $\xrightarrow{\text{inline}}$ "let" | "constant" | "var" | "config"

direction $\xrightarrow{\text{inline}}$ "to" | "downto"

```
alt -inline-> "when" pattern_list option("where" expr) "=>" stmt_list
       | "otherwise" stmt_list

otherwise_opt --> option("otherwise" "=>" stmt_list)

catcher -inline-> "when" ID ":" ty "=>" stmt_list
           | "when" ty "=>" stmt_list

stmt -inline-> "if" expr "then" stmt_list s_else "end"
         | "case" expr "of" list+(alt) "end"
         | "while" expr "do" stmt_list "end"
         | "@looplimit" "(" expr ")" "while" expr "do" stmt_list "end"
         | "for" ID "=" expr direction expr "do" stmt_list "end"
         | "try" stmt_list "catch" list+(catcher) otherwise_opt "end"
         | "pass" ";"
         | "return" option(expr) ";"
         | ID plist*(expr) ";"
         | "assert" expr ";"
         | local_decl_keyword decl_item "=" expr ";"
         | lexpr "=" expr ";"
         | "var" decl_item option("=" expr) ";"
         | "var" clist2(ID) ":" ty ";"
         | "print" plist*(expr) ";"
         | "repeat" stmt_list "until" expr ";"
         | "@looplimit" "(" expr ")" "repeat" stmt_list "until" expr ";"
         | "throw" expr ";"
         | "throw" ";"
         | "pragma" ID clist*(expr) ";"

stmt_list -inline-> list+(stmt)
```

```
s_else ⟶ "elseif" expr "then" stmt_list s_else
       | "pass"
       | "else" stmt_list
```

```
lexpr ⟶[inline] lexpr_atom
      | "_"
      | "(" clist+(lexpr) ")"
```

```
lexpr_atom ⟶ ID
           | lexpr_atom slices
           | lexpr_atom "." IDfield
           | lexpr_atom "." "[" clist*(ID) "]"
           | "[" clist+(lexpr_atom) "]"
```

A decl_item is another kind of left-hand-side expression, which appears only in declarations. It cannot have setter calls or set record fields, it must declare a new variable.

```
decl_item ⟶ untyped_decl_item as_ty
          | untyped_decl_item
```

```
untyped_decl_item ⟶[inline] ID
                  | "_"
                  | plist2(decl_item)
```

```
int_constraints ⟶[inline] "{" clist+(int_constraint) "}"
```

```
int_constraints_opt ⟶[inline] int_constraints | ϵ
```

```
int_constraint ⟶[inline] expr
               | expr ".." expr
```

Pattern expressions (expr_pattern), given by the following derivations, is similar to regular expressions (expr), except they do not derive tuples, which are the last derivation for expr.

```
expr_pattern ⟶ value
               | ID
               | expr_pattern binop expr
               | unop expr                                   precedence: UNOPS
               | "if" expr "then" expr e_else
               | ID plist*(expr)
               | expr_pattern slices
               | expr_pattern "." ID
               | expr_pattern "." "[" clist+(ID) "]"
               | "[" clist+(expr) "]"
               | expr_pattern "as" ty
               | expr_pattern "as" int_constraints
               | expr_pattern "IN" pattern_set
               | expr_pattern "IN" MASK_LIT
               | "UNKNOWN" ":" ty
               | ID "{" clist*(field_assign) "}"
               | "(" expr_pattern ")"

pattern_set ⟶[inline] "!" "{" pattern_list "}"
              | "{" pattern_list "}"

pattern_list ⟶[inline] clist+(pattern)

pattern ⟶ expr_pattern
          | expr_pattern ".." expr
          | "-"
          | "<=" expr
          | ">=" expr
          | MASK_LIT
          | plist2(pattern)
          | pattern_set
```

```
fields --inline--> "{" tclist*(typed_identifier) "}"

fields_opt --inline--> fields | ϵ

nslices --inline--> "[" clist+(slice) "]"

slices --inline--> "[" clist*(slice) "]"

slice --inline--> expr
                | expr ":" expr
                | expr "+:" expr
                | expr "*:" expr

bitfields --inline--> "{" tclist*(bitfield) "}"

bitfield --inline--> nslices ID
                   | nslices ID bitfields
                   | nslices ID ":" ty

ty --> "integer" option(int_constraints)
     | "real"
     | "boolean"
     | "string"
     | "bit"
     | "bits" "(" expr ")" list*(bitfields)
     | plist*(ty)
     | ID
     | "array" "[" expr "]" "of" ty
```

ty_decl ⟶ ty
| "enumeration" "{" ntclist(**ID**) "}"
| "record" fields_opt
| "exception" fields_opt

field_assign $\xrightarrow{\text{inline}}$ **ID** "=" expr

e_else ⟶ "else" expr
| "elseif" expr "then" expr e_else

expr ⟶ value
| **ID**
| expr binop expr
| unop expr precedence: **UNOPS**
| "if" expr "then" expr e_else
| **ID** plist*(expr)
| expr slices
| expr "." **ID**
| expr "." "[" clist^{+}(**ID**) "]"
| "[" clist^{+}(expr) "]"
| expr "as" ty
| expr "as" int_constraints
| expr "IN" pattern_set
| expr "IN" **MASK_LIT**
| "UNKNOWN" ":" ty
| **ID** "{" clist*(field_assign) "}"
| "(" expr ")"
| plist2(expr)

`value` $\xrightarrow{\text{inline}}$ **INT_LIT**
| **BOOL_LIT**
| **REAL_LIT**
| **BITVECTOR_LIT**
| **STRING_LIT**

`unop` $\xrightarrow{\text{inline}}$ `"!"` | `"-"` | `"NOT"`

`binop` $\xrightarrow{\text{inline}}$ `"AND"` | `"&&"` | `"||"` | `"<->"` | `"DIV"` | `"DIVRM"` | `"XOR"` | `"=="` | `"!="`
| `">"` | `">="` | `"-->"` | `"<"` | `"<="` | `"+"` | `"-"` | `"MOD"` | `"*"`
| `"OR"` | `"RDIV"` | `"<<"` | `">>"` | `"^"` | `"++"`

6.5 Parse Trees

We now define *parse trees* for the ASL expanded grammar. Those are later used for build Abstract Syntax Trees.

Definition 26 (Parse Trees) *A* parse tree *has one of the following forms:*

- *A* token node*, given by the token itself, for example,* **LEXEME**(`"=>"`) *and* **ID**(*id*)*;*
- *epsilon_node, which represents the empty sentence —* ϵ*.*
- *A* non-terminal node *of the form* $N(n_{1..k})$ *where* N *is a non-terminal symbol, which is said to label the node, and* $n_{1..k}$ *are its children parse nodes, for example,* *decl*(*"func"*, **ID**(*id*), *params_opt*, *func_args*, *func_body*) *is labeled by* *decl* *and has five children nodes.*

(In the literature, parse trees are also referred to as *derivation trees*.)

Definition 27 (Well-formed Parse Trees) *A parse tree is* well-formed *if its root is labelled by the start non-terminal (*`ast`* for ASL) and each non-terminal node* $N(n_{1..k})$ *corresponds to a grammar derivation* $N \longrightarrow l_{1..k}$ *where* l_i *is the label of node* n_i *if it is a non-terminal node and* n_i *itself when it is a token. A non-terminal node* N(*epsilon_node*) *is well-formed if the grammar includes a derivation* $N \longrightarrow \epsilon$*.*

Definition 28 (Parse Tree Yield) *The* yield *of a parse tree is the list of tokens given by an in-order walk of the tree:*

$$\mathit{yield}(n) \triangleq \begin{cases} [t] & n \text{ is a token } t \\ [\,] & n = \mathit{epsilon\_node} \\ \mathit{yield}(n_1) + \ldots + \mathit{yield}(n_k) & n = N(n_{1..k}) \end{cases}$$

We denote the set of well-formed parse trees for a non-terminal symbol S by $\mathbb{PARSE}[S]$.

A parser is a function

$$\mathit{asl\_parse} : (\mathbb{TOKEN}^* \setminus \{\mathbf{T\_ERR}\}) \longrightarrow \mathbb{PARSE}[\mathtt{ast}] \cup \{\#\mathsf{PE}\}$$

where $\#\mathsf{PE}$ stands for a *parse error*. If $\mathit{asl\_parse}(ts) = n$ then $\mathsf{yield}(n) = ts$ and if $\mathit{asl\_parse}(ts) = \#\mathsf{PE}$ then there is no well-formed tree n such that $\mathsf{yield}(n) = ts$. (Notice that we do not define a parser if ts is lexically illegal.)

The *language of a grammar* G is defined as follows:

$$\mathsf{Lang}(G) = \{\mathsf{yield}(n) \mid n \text{ is a well-formed parse tree for } G\} \ .$$

6.6 Priority and Associativity

A context-free grammar G is *ambiguous* if there can be more than one parse tree for a given list of tokens $ts \in \mathsf{Lang}(G)$. Indeed the expanded ASL grammar is ambiguous, for example, due to its definition of binary operation expressions. To allow assigning a unique parse tree to each sequence of tokens in the language of the ASL grammar, we utilize the standard technique of associating priority levels to productions and using them to resolve any shift-reduce conflicts in the LR(1) parser associated with our grammar (our grammar does not have any reduce-reduce conflicts).

The priority of a grammar derivation is defined as the priority of its rightmost token. Derivations that do not contain tokens do not require a priority as they do not induce shift-reduce conflicts.

The table below assigns priorities to tokens in increasing order, starting from the lowest priority (for `"else"`) to the highest priority (for `"."`). When a shift-reduce conflict arises during the LR(1) grammar construction it resolve in favor of the action (shift or reduce) associated with the derivation that has the higher priority. If two derivations have the same priority due to them both having the same rightmost token, the conflict is resolved based on the associativity associated with the token below: reduce for left, shift for right, and a parsing error for nonassoc.

The two rules involving a unary minus operation are not assigned the priority level of `"-"`, but rather then the priority level **UNOPS**, as denoted by the notation precedence: **UNOPS** appearing to their right. This is a standard way of dealing with a unary minus operation in many programming languages, which involves defining an artificial token **UNOPS**, which is never returned by the scanner.

Terminals	**Associativity**
`"else"`	nonassoc
`"||"`, `"&&"`, `"-->"`, `"<->"`, `"as"`	left
`"=="`, `"!="`	left
`">"`, `">="`, `"<"`, `"<="`	nonassoc
`"+"`, `"-"`, `"OR"`, `"XOR"`, `"AND"`	left
`"*"`, `"DIV"`, `"DIVRM"`, `"RDIV"`, `"MOD"`, `"<<"`, `">>"`	left
`"^"`, `"++"`	left
UNOPS	nonassoc
`"IN"`	nonassoc
`"."`, `"["`	left

Chapter 7

ASL Abstract Syntax

An abstract syntax is a form of context-free grammar over structured trees. Compilers and interpreters typically start by parsing the text of a program and producing an abstract syntax tree (AST, for short), and then continue to operate over that tree. The reason for this is that abstract syntax trees abstract away details that are irrelevant to the semantics of the program, such as punctuation and scoping syntax, which are useful for readability and parsing.

Technically, there are two abstract syntaxes: an *untyped abstract syntax* and a *typed abstract syntax*. The first syntax results from parsing the text of an ASL specification. The type checker checks whether the untyped AST is valid and if so produces a typed AST where some nodes in the untyped AST have been transformed to more explicit representation. For example, the untyped AST may contain what looks like a slicing expression, which turns out to be a call to a getter. The typed AST represents that call directly, making it simpler for an interpreter to evaluate that expression.

Outline The outline of this chapter is as follows, We first define the type of Abstract Syntax Trees used by ASL (Section 7.1). We then define the notations for defining the AST grammar used by ASL (Section 7.2) Finally, we define the AST grammar rules for the different ASL constructs along with examples:

- Identifiers (Section 7.3.1)
- Literal values (Section 7.3.2)
- Basic Operations (Section 7.3.3)
- Expressions (Section 7.3.4)
- Patterns (Section 7.3.5)
- Slices (Section 7.3.6)
- Types (Section 7.3.7)

- Constraints (Section 7.3.8)
- Bit Fields (Section 7.3.9)
- Array Indices (Section 7.3.10)
- Fields and Typed Identifiers (Section 7.3.11)
- Left-hand Side Expressions (Section 7.3.12)
- Local Declarations (Section 7.3.13)
- Statements (Section 7.3.14)
- Case Alternatives (Section 7.3.15)
- Exception Catchers (Section 7.3.16)
- Subprograms (Section 7.3.17)
- Global Declarations (Section 7.3.18)
- Specifications (Section 7.3.19)

7.1 ASL Abstract Syntax Trees

In an ASL abstract syntax tree, a node is one the following data types:

Token Node. A lexical token, denoted as in the lexical description of ASL;

Label Node. A label

Unlabelled Tuple Node. A tuple of children nodes, denoted as $(n_1, \ldots, n_k)$;

Labelled Tuple Node. A tuple labelled L, denoted as $L(n_1, \ldots, n_k)$;

List Node. A list of 0 or more children nodes, denoted as [ ] when the list is empty and $[n_1, \ldots, n_k]$ for non-empty lists;

Optional. An optional node stands for a list of 0 or 1 occurrences of a sub-node n. We denote an empty optional by $\langle\rangle$ and the non-empty optional by $\langle n \rangle$;

Record Node. A record node, denoted as $\{\text{name}_1 : n_1, \ldots, \text{name}_k : n_k\}$, where $\text{name}_1 \ldots \text{name}_k$ are names, which associates names with corresponding nodes.

7.2 ASL Abstract Syntax Grammar

An abstract syntax is defined in terms of derivation rules containing variables (also referred to as non-terminals). A *derivation rule* has the form $v \longrightarrow rhs$ where v is a non-terminal variable and *rhs* is a *node type*. We write n, $n_1, \ldots, n_k$ to denote node types. Node types are defined recursively as follows:

Non-terminal. A non-terminal variable;

Terminal. A lexical token t or a label L;

Unlabelled Tuple. A tuple of node types, denoted as $(n_1, \ldots, n_k)$;

Labelled Tuple. A tuple labelled L, denoted as $L(n_1, \ldots, n_k)$;

List. A list node type, denoted as n^*;

Optional. An optional node type, denoted as $n?$;

Record. A record, denoted as $\{\text{name}_1 : n_1, \ldots, \text{name}_k : n_k\}$ where name_i, which associates names with corresponding node types.

An abstract syntax consists of a set of derivation rules and a start non-terminal.

7.3 ASL Untyped Abstract Syntax

The abstract syntax of ASL is given in terms of the derivation rules below and the start non-terminal `specification`. Some extra details are given by using the notation $\overbrace{symbol}^{\text{detail}}$.

7.3.1 Identifiers

Identifiers in the AST, denoted identifier are simply strings representing ASL identifiers. Those are obtained directly from the values of identifier tokens, **ID**(s).

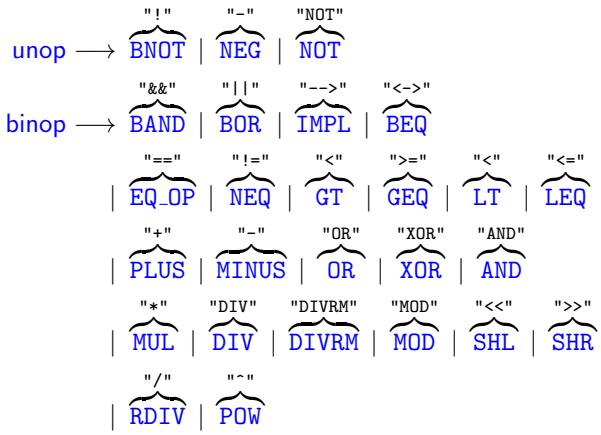
7.3.2 Literal Values

The following rules correspond to literal values of the following ASL data types: integers, Booleans, real numbers, bitvectors, and strings.

$$
\begin{aligned}
\text{literal} \longrightarrow\ & \text{L\_Int}(\overbrace{n}^{\mathbb{Z}}) \\
\mid\ & \text{L\_Bool}(\overbrace{b}^{\{\texttt{TRUE},\texttt{FALSE}\}}) \\
\mid\ & \text{L\_Real}(\overbrace{q}^{\mathbb{Q}}) \\
\mid\ & \text{L\_Bitvector}(\overbrace{B}^{B \in \{0,1\}^*}) \\
\mid\ & \text{L\_String}(\overbrace{S}^{S \in \{C \mid \text{"}C\text{"} \in \mathbb{S}\}})
\end{aligned}
$$

7.3.3 Basic Operations

The following rules correspond to unary operations and binary operations that can be applied to expressions.



7.3.4 Expressions

The following rules correspond to various types of expressions: literal expressions, variable expressions, typing assertions, binary operation expressions, unary operation expressions, call expressions, slicing expressions, conditional expressions, single-field access expressions, multiple-field access expressions, record and exception construction expressions, concatenation expressions, tuple expressions, unknown-value expressions, and pattern

matching expressions.

expr ⟶ E_Literal(literal)
| E_Var(variable name: identifier)
| E_ATC(Type assertion: expr, asserted type: ty)
| E_Binop(binop, expr, expr)
| E_Unop(unop, expr)
| E_Call(subprogram name: identifier, actual arguments: expr*)
| E_Slice(expr, slice*)
| E_Cond(condition: expr, then: expr, else: expr)
| E_GetField(record: expr, field name: identifier)
| E_GetFields(record: expr, field names: identifier*)
| E_Record(record type: ty, field initializers: (identifier, expr)*)
| E_Concat(expr+)
| E_Tuple(expr+)
| E_Unknown(ty)
| E_Pattern(expr, pattern)

Figure. 7.1 and Figure. 7.2 exemplify the different kinds of expressions, as indicated by respective comments.

- E_Var(`x`) represents variables (`E_Var 1`) as well as getters declared without a list of arguments (`E_Var 2`).
- E_ATC(`e`, `t`) represents typing assertions. For example: `x as integer`. Here `e` corresponds to `x` and `t` corresponds to `integer`.
- E_Slice(`e`, `slices`) represents slices of bitvectors (`E_Slice 1`), slices of integers (`E_Slice 2`), calls to getters (`E_Slice 3` and `E_Slice 4`), and access to array elements (`E_Slice 5`).
- E_GetField(`e`, `id`) represents an access to a record (`E_GetField 1`) or exception field as well as an access to a tuple component (`E_GetField 2`).
- E_GetFields(`e`, `ids`) represents an access to multiple record fields (`E_GetFields 1`).

```
getter g_no_args => integer
begin
  return 0;
end

getter g0_bits[] => bits(4)
begin
  return '1000';
end

getter g1_bits[p: integer] => bits(4)
begin
  return '1000'[p, 2:0];
end

type point of record{x: bits(4), y: bits(4)};
type except of exception;

func main() => integer
begin
  var v: integer = 4;
  // E_Var 1: v is a variable expression.
  // E_Var 2: g_no_args is a call to a getter with no arguments.
  var - = v + g_no_args;

  var b0 = '1111 1000'[3:1, 0]; // E_Slice 1: a bitvector slice.
  var b1 = 0xF8[3:1, 0]; // E_Slice 2: an integer slice.
  // E_Slice 3: g0_bits[] is a call to a getter.
  assert b0 == g0_bits[];
  // E_Slice 4: g1_bits[3] is a call to a getter.
  assert b0 == g1_bits[3];
  var bits_arr : array [1] of bits(4);
  // E_Binop: 1: b0 == b1 is a binary expression for ==.
  // E_Cond 1: the right-hand side of the assignment is
  //           a conditional expression.
  bits_arr[0] = if (b0 == b1) then '1000' else '0000';
  // E_Slice 5: bits_arr[0] stands for an array access
  assert b0 == bits_arr[0];
  // E_Unop 1   : (NOT b8) negate the bits of b8.
  // E_Binop 2  : the right-hand side of the assignment is
  //              a binay AND expression.
  // E_Concat 1 : [b0, b1] concatenates two bitvectors.
  // E_Unknown 1: UNKNOWN: bits(8) represents an arbitrary
  //              8-bits bitvector
  var b8 = [b0, b1];
  b8 = (NOT b8) AND UNKNOWN: bits(8);
  return 0;
end
```

Figure 7.1: Examples of expressions

```
getter g_no_args => integer
begin
  return 0;
end

getter g0_bits[] => bits(4)
begin
  return '1000';
end

getter g1_bits[p: integer] => bits(4)
begin
  return '1000'[p, 2:0];
end

type point of record{x: bits(4), y: bits(4)};
type except of exception;

func main() => integer
begin
  // E_Record 1: a record construction expression.
  var p = point{x = '1111', y = '0000'};
  // E_GetField 1: reading a single field.
  var b0 = p.x;
  // E_GetFields 1: reading multiple fields.
  var b8: bits(8)  = p.[x, y];
  // E_Concat 1: [b0, b1] concatenates two bitvectors.
  b8 = [b0, b0];
  // E_Tuple 1: constructing a pair of two 4-bit bitvectors.
  var t2 = (b0, b0);
  // E_GetField 2: reading the first tuple item.
  // E_Pattern 1: the condition in side the if is a pattern.
  if (t2.item0 IN {'1110'}) then
    // E_Record 2: an exception construction.
    throw except{};
  end

  return 0;
end
```

Figure 7.2: Examples of expressions

7.3.5 Patterns

pattern $\longrightarrow$ Pattern_All
| Pattern_Any(pattern*)
| Pattern_Geq(expr)
| Pattern_Leq(expr)
| Pattern_Mask($\overbrace{\{0,1,x\}^*}^{\text{mask constant}}$)
| Pattern_Not(pattern)
| Pattern_Range($\overbrace{\text{expr}}^{\text{lower}}$, $\overbrace{\text{expr}}^{\text{upper}}$)
| Pattern_Single(expr)
| Pattern_Tuple(pattern*)

7.3.6 Slices

slice $\longrightarrow$ Slice_Single($\overbrace{\text{expr}}^{\texttt{i}}$)
| Slice_Range($\overbrace{\text{expr}}^{\texttt{j}}$, $\overbrace{\text{expr}}^{\texttt{i}}$)
| Slice_Length($\overbrace{\text{expr}}^{\texttt{i}}$, $\overbrace{\text{expr}}^{\texttt{n}}$)
| Slice_Star($\overbrace{\text{expr}}^{\texttt{i}}$, $\overbrace{\text{expr}}^{\texttt{n}}$)

7.3.7 Types

ty ⟶ T_Int(int_constraints)
| T_Real
| T_String
| T_Bool
| T_Bits(width: expr, bitfield*)
| T_Enum(labels: identifier*)
| T_Tuple(ty*)
| T_Array(array_index, ty)
| T_Record(field*)
| T_Exception(field*)
| T_Named(type name: identifier)

7.3.8 Constraints

int_constraints ⟶ Unconstrained
| WellConstrained(int_constraint+)
| Parameterized(parameter: identifier)

int_constraint ⟶ Constraint_Exact(expr)
| Constraint_Range(expr, expr)

7.3.9 Bit Fields

bitfield ⟶ BitField_Simple(identifier, slice*)
| BitField_Nested(identifier, slice*, bitfield*)
| BitField_Type(identifier, slice*, ty)

7.3.10 Array Indices

The type of array indices is given by the following AST type:

array_index ⟶ ArrayLength_Expr(array length: expr)

7.3.11 Fields and Typed Identifiers

The following rule corresponds to a field of a record-like structure:

field ⟶ (identifier, ty)

The following rule corresponds to an identifier with its associated type:

typed_identifier ⟶ (identifier, ty)

7.3.12 Left-hand Side Expressions

The following rules define the types of left-hand side of assignments:

lexpr ⟶ "-" LE_Discard
| LE_Var(identifier)
| LE_Slice(lexpr, slice*)
| LE_SetArray(lexpr, expr)
| LE_SetField(lexpr, identifier)
| LE_SetFields(lexpr, identifier*)
| LE_Destructuring(lexpr*)
| LE_Concat(lexpr+)

LE_Concat(`les`) represents the left-hand-side list of expressions for simultaneous assignments.

7.3.13 Local Declarations

local_decl_keyword ⟶ LDK_Var | LDK_Constant | LDK_Let

A local declaration item is the left-hand side of a declaration statements. In the following example of a declaration statement:

```
let (x, -, z): (integer, integer, integer {0..32}) = (2, 3, 4);
```

the local declaration item is `(x, -, z): (integer, integer, integer {0..32})`.

local_decl_item ⟶ LDI_Discard
| LDI_Var(identifier)
| LDI_Tuple(local_decl_item*)
| LDI_Typed(local_decl_item, ty)

7.3.14 Statements

for_direction $\longrightarrow$ `Up` | `Down`

stmt $\longrightarrow$ S_Pass
| S_Seq(stmt, stmt)
| S_Decl(local_decl_keyword, local_decl_item, expr?)
| S_Assign(lexpr, expr)
| S_Call($\overbrace{\text{identifier}}^{\text{subprogram name}}$, $\overbrace{\text{expr}^*}^{\text{actual arguments}}$)
| S_Return(expr?)
| S_Cond(expr, stmt, stmt)
| S_Case(expr, case_alt*)
| S_Assert(expr)
| S_For $\left\{ \begin{array}{rcl} \text{index\_name} & : & \text{identifier}, \\ \text{start\_e} & : & \text{expr}, \\ \text{dir} & : & \text{for\_direction}, \\ \text{end\_e} & : & \text{expr}, \\ \text{body} & : & \text{stmt}, \\ \text{limit} & : & \text{expr}? \end{array} \right\}$
| S_While($\overbrace{\text{expr}}^{\text{condition}}$, $\overbrace{\text{expr}?}^{\text{loop limit}}$, $\overbrace{\text{stmt}}^{\text{loop body}}$)
| S_Repeat($\overbrace{\text{stmt}}^{\text{loop body}}$, $\overbrace{\text{expr}}^{\text{condition}}$, $\overbrace{\text{expr}?}^{\text{loop limit}}$)
// *The option represents an implicit throw:* `throw;`.
| S_Throw(expr?)

| S_Try(stmt, catcher*, $\overbrace{\text{stmt}?}^{\text{otherwise}}$)
| S_Print($\overbrace{\text{expr}^*}^{\text{args}}$, $\overbrace{\mathbb{B}}^{\text{debug}}$)

7.3.15 Case Alternatives

case_alt $\longrightarrow$ {pattern : pattern, where : expr?, stmt : stmt}

7.3.16 Exception Catchers

$$\text{catcher} \longrightarrow (\ \overbrace{\text{identifier?}}^{\text{exception to match}}\ ,\ \overbrace{\text{ty}}^{\text{guard type}}\ ,\ \overbrace{\text{stmt}}^{\text{statement to execute on match}}\ )$$

7.3.17 Subprograms

$$\begin{aligned}\text{sub\_program\_type} \longrightarrow\ & \texttt{ST\_Procedure} \mid \texttt{ST\_Function} \\ \mid\ & \texttt{ST\_Getter} \mid \texttt{ST\_EmptyGetter} \\ \mid\ & \texttt{ST\_Setter} \mid \texttt{ST\_EmptySetter}\end{aligned}$$

$$\text{sub\_program\_body} \longrightarrow \texttt{SB\_ASL}(\text{stmt}) \mid \texttt{SB\_Primitive}$$

$$\text{func} \longrightarrow \left\{ \begin{array}{rcl} \text{name} & : & \mathbb{S}, \\ \text{parameters} & : & (\text{identifier}, \text{ty?})^*, \\ \text{args} & : & \text{typed\_identifier}^*, \\ \text{body} & : & \text{sub\_program\_body}, \\ \text{return\_type} & : & \text{ty?}, \\ \text{subprogram\_type} & : & \text{sub\_program\_type} \end{array} \right\}$$

7.3.18 Global Declarations

Declaration keyword for global storage elements:

$$\text{global\_decl\_keyword} \longrightarrow \texttt{GDK\_Constant} \mid \texttt{GDK\_Config} \mid \texttt{GDK\_Let} \mid \texttt{GDK\_Var}$$

$$\text{global\_decl} \longrightarrow \left\{ \begin{array}{rcl} \text{keyword} & : & \text{global\_decl\_keyword}, \\ \text{name} & : & \text{identifier}, \\ \text{ty} & : & \text{ty?}, \\ \text{initial\_value} & : & \text{expr?} \end{array} \right\}$$

$$\begin{aligned}\text{decl} \longrightarrow\ & \texttt{D\_Func}(\text{func}) \\ \mid\ & \texttt{D\_GlobalStorage}(\text{global\_decl}) \\ \mid\ & \texttt{D\_TypeDecl}(\text{identifier}, \text{ty}, (\text{identifier}, \overbrace{\text{field}^*}^{\text{with fields}})?)\end{aligned}$$

7.3.19 Specifications

$$\texttt{specification} \longrightarrow \text{decl}^*$$

7.4 ASL Typed Abstract Syntax

The derivation rules for the typed abstract syntax are the same as the rules for the untyped abstract syntax, except for the following differences.

The rules for expressions have the extra derivation rule:

$$\mathsf{expr} \longrightarrow \mathsf{E\_GetArray}(\overbrace{\mathsf{expr}}^{\text{base}}, \overbrace{\mathsf{expr}}^{\text{index}})$$

The AST node for call expressions includes an extra component that explicitly associates expressions with parameters:

$$\mathsf{expr} \longrightarrow \mathsf{E\_Call}(\ \overbrace{\mathsf{identifier}}^{\text{subprogram name}}\ ,\ \overbrace{\mathsf{expr}^{*}}^{\text{actual arguments}}\ ,\ \overbrace{(\mathsf{identifier}, \mathsf{expr})^{*}}^{\text{parameters with initializers}}\ )$$

The AST node for a left-hand-side tuple of expressions contains a second component, which is a type annotation:

$$\mathsf{lexpr} \longrightarrow \mathsf{LE\_Concat}(\mathsf{lexpr}^{+}, \mathbf{INT\_LIT}^{+}?)$$

The rules for statements refine the throw statement by annotating it with the type of the thrown exception.

$$\mathsf{stmt} \longrightarrow \mathsf{S\_Throw}((\mathsf{expr},\ \overbrace{\mathsf{ty}}^{\text{exception type}}\ )?)$$

Similar to expressions, the AST node for call statements includes an extra component that explicitly associates expressions with parameters:

$$\mathsf{stmt} \longrightarrow \mathsf{S\_Call}(\ \overbrace{\mathsf{identifier}}^{\text{subprogram name}}\ ,\ \overbrace{\mathsf{expr}^{*}}^{\text{actual arguments}}\ ,\ \overbrace{(\mathsf{identifier}, \mathsf{expr})^{*}}^{\text{parameters with initializers}}\ )$$

The rules for slices is replaced by the following:

$$\mathsf{slice} \longrightarrow \mathsf{Slice\_Length}(\mathsf{expr}, \mathsf{expr})$$

This reflects the fact that all other slicing constructs are syntactic sugar for Slice_Length.

The following extra rule enables expressing array indices based on enumeration:

$$\mathsf{array\_index} \longrightarrow \mathsf{ArrayLength\_Enum}(\ \overbrace{\mathsf{identifier}}^{\text{name of enumeration}}\ ,\ \overbrace{\mathbb{Z}}^{\text{length}}\ )$$

Chapter 8

Building Abstract Syntax Trees

This chapter define how to transform a parse tree into the corresponding AST via recursively traversing the parse tree and applying a *builder* function for each non-terminal node.

(Some of the builders are relations due to non-determinism induced by naming global variables for assignments whose left-hand-side variable is discarded (`"-"`).)

For each non-terminal $N \longrightarrow R_1 \mid \ldots R_k$, we define a builder function $\mathsf{build\_}N$ which takes a parse tree $\mathbb{PARSE}[N]$ and returns the corresponding AST. The builder function is defined in terms of one inference rule per alternative R_i. The input for the builder for an alternative $R = S_{1..m}$ is a parse node $N(S_{1..m})$. To allow the builder to refer to the children nodes of N, we use the notation $n_i : S_i$, which names the child node S_i as n_i.

8.1 Example

Consider the derivation for while loops:

$$\texttt{stmt} \longrightarrow \texttt{"while"}\ \texttt{expr}\ \texttt{"do"}\ \texttt{stmt\_list}\ \texttt{"end"}$$

The parse node for a while statement has the form

$$\texttt{stmt}(\texttt{"while"}, \texttt{e} : \texttt{expr}, \texttt{"do"}, \texttt{stmt\_list} : \texttt{stmt\_list}, \texttt{"end"})$$

where `e` names the node representing the condition of the loop and `stmt_list` names the list of statements that form the body of the loop.

To build the corresponding AST, we employ the builder function *build_stmt*, since the non-terminal labelling the parse node is `stmt`.

We also employ the following rule:

$$\frac{\mathit{build\_expr}(\texttt{e}) \xrightarrow{\text{ast}} \texttt{e\_ast} \qquad \mathit{build\_stmt\_list}(\texttt{stmt\_list}) \xrightarrow{\text{ast}} \texttt{stmt\_list\_ast}}{\mathit{build\_stmt}(\texttt{stmt}(\texttt{"while"}, \texttt{e} : \texttt{expr}, \texttt{"do"}, \texttt{stmt\_list} : \texttt{stmt\_list}, \texttt{"end"})) \xrightarrow{\text{ast}} \text{S\_While}(\texttt{e\_ast}, \texttt{None}, \texttt{stmt\_list\_ast})}$$

That is, we apply the *build_expr* to transform the condition parse node `e` into the corresponding AST node, we apply *build_stmt_list* to transform the parse node `stmt_list` for the body of the list into the corresponding AST node, and finally return the AST node for `while` loops — S_While — with the two nodes as its children.

8.2 Abbreviated Rule Notation

Notice that there is only one instance of `expr` and one instance of `stmt_list` in this production. This is very common and we therefore use the following shorthand notation for such cases, as explained next.

In a non-terminal N appears only once in the right-hand-side of a derivation, we use the name `N` to name the corresponding child parse node. For example, `expr` : `expr` and `stmt_list` : `stmt_list`. In such cases, we always have the premise $\text{build\_}N(\texttt{N}) \xrightarrow{\text{ast}} N\texttt{\_ast}$ to obtain the corresponding AST node. We therefore make this premise implicit, by dropping it entirely and using $\overline{N}$ to mean that the parse node N is named `N`, the premise $\text{build\_}N(\texttt{N}) \xrightarrow{\text{ast}} N\texttt{\_ast}$ is considered part of the rule and N`_ast` itself stands for N`_ast`.

In our example, this results in the abbreviated rule notation

$$\mathit{build\_stmt}(\texttt{stmt}(\texttt{"while"}, \texttt{expr}, \texttt{"do"}, \texttt{stmt\_list}, \texttt{"end"})) \xrightarrow{\text{ast}} \text{S\_While}(\overline{\texttt{expr}}, \texttt{None}, \overline{\texttt{stmt\_list}})$$

8.3 AST Builder Functions and Relations

We define the following rules for transforming the various non-terminal parse nodes into the corresponding AST nodes:

- SyntaxRule.AST (see Section 8.4)
- SyntaxRule.GlobalDecl (see Section 8.5)
- SyntaxRule.Subtype (see Section 8.6)
- SyntaxRule.Subtypeopt (see Section 8.7)
- SyntaxRule.TypedIdentifier (see Section 8.8)
- SyntaxRule.OptTypedIdentifier (see Section 8.9)
- SyntaxRule.ReturnType (see Section 8.10)
- SyntaxRule.ParamsOpt (see Section 8.11)

- SyntaxRule.AccessArgs (see Section 8.12)
- SyntaxRule.FuncArgs (see Section 8.13)
- SyntaxRule.MaybeEmptyStmtList (see Section 8.14)
- SyntaxRule.FuncBody (see Section 8.15)
- SyntaxRule.IgnoredOrIdentifier (see Section 8.16)
- SyntaxRule.LocalDeclKeyword (see Section 8.17)
- SyntaxRule.StorageKeyword (see Section 8.18)
- SyntaxRule.Direction (see Section 8.19)
- SyntaxRule.Alt (see Section 8.20)
- SyntaxRule.OtherwiseOpt (see Section 8.21)
- SyntaxRule.Catcher (see Section 8.22)
- SyntaxRule.Stmt (see Section 8.23)
- SyntaxRule.StmtList (see Section 8.24)
- SyntaxRule.SElse (see Section 8.25)
- SyntaxRule.LExpr (see Section 8.26)
- SyntaxRule.LExprAtom (see Section 8.27)
- SyntaxRule.DeclItem (see Section 8.28)
- SyntaxRule.UntypedDeclItem (see Section 8.29)
- SyntaxRule.IntConstraints (see Section 8.30)
- SyntaxRule.IntConstraintsopt (see Section 8.31)
- SyntaxRule.IntConstraint (see Section 8.32)
- SyntaxRule.ExprPattern (see Section 8.33)
- SyntaxRule.PatternSet (see Section 8.34)
- SyntaxRule.PatternList (see Section 8.35)
- SyntaxRule.Pattern (see Section 8.36)
- SyntaxRule.Fields (see Section 8.37)
- SyntaxRule.FieldsOpt (see Section 8.38)

- SyntaxRule.NSlices (see Section 8.39)
- SyntaxRule.Slices (see Section 8.40)
- SyntaxRule.Slice (see Section 8.41)
- SyntaxRule.Bitfields (see Section 8.42)
- SyntaxRule.Bitfield (see Section 8.43)
- SyntaxRule.Ty (see Section 8.44)
- SyntaxRule.TyDecl (see Section 8.45)
- SyntaxRule.FieldAssign (see Section 8.46)
- SyntaxRule.EElse (see Section 8.47)
- SyntaxRule.Expr (see Section 8.48)
- SyntaxRule.Value (see Section 8.49)
- SyntaxRule.Unop (see Section 8.50)
- SyntaxRule.Binop (see Section 8.51)

We also define the following helper functions:

- SyntaxRule.StmtFromList (see Section 8.52)
- SyntaxRule.SequenceStmts (see Section 8.53)

8.4 SyntaxRule.AST

The relation

$$\mathit{build\_ast} : \overbrace{\mathbb{PARSE}[\texttt{ast}]}^{\texttt{parsed\_node}} \times \overbrace{\texttt{specification}}^{\texttt{ast\_node}}$$

transforms an `ast` node `parsed_node` into an AST specification node `ast_node`.

Notice that we define some builders as relations rather than functions. This is due to the non-determinism in creating identifiers for auxiliary variables that stand in for instances of `-` on the left-hand-side of assignments and declarations. For example, `- = 5;` will effectively create some auxiliary variable, which will result in an AST node such as S_Assign(E_Var(`aux-1`), E_Literal(L_Int(5))). Recall that hyphens are not legal characters in ASL identifiers, which avoids potential clashes with user-supplied identifiers. An implementation is free however to choose other naming schemes that avoid name clashes, for example, by employing counters.

$$\frac{\text{AST}\quad \mathit{build\_list}[\mathit{build\_decl}](\texttt{decls}) \xrightarrow{\text{ast}} \texttt{adecls}}{\mathit{build\_ast}(\overbrace{\texttt{ast}(\texttt{decls} : \mathsf{list}^*(\texttt{decl}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{adecls}}^{\texttt{ast\_node}}}$$

8.5 SyntaxRule.GlobalDecl

The relation

$$\mathit{build\_decl} : \overbrace{\mathbb{PARSE}[\texttt{decl}]}^{\texttt{parsed\_node}} \times \overbrace{\texttt{decl}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

FUNC_DECL

$$\mathit{build\_decl}(\overbrace{\texttt{decl}(\texttt{"func"}, \mathbf{ID}(\texttt{name}), \texttt{params\_opt}, \texttt{func\_args}, \texttt{return\_type}, \texttt{func\_body})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}}$$
$$\overbrace{\texttt{D\_Func}\left(\left\{\begin{array}{rcl} \text{name} & : & \texttt{name}, \\ \text{parameters} & : & \overline{\texttt{params\_opt}}, \\ \text{args} & : & \overline{\texttt{func\_args}}, \\ \text{body} & : & \texttt{SB\_ASL}(\overline{\texttt{func\_body}}), \\ \text{return\_type} & : & \langle\overline{\texttt{return\_type}}\rangle, \\ \text{subprogram\_type} & : & \texttt{ST\_Function} \end{array}\right\}\right)}^{\texttt{ast\_node}}$$

PROCEDURE_DECL

$$\mathit{build\_decl}(\overbrace{\texttt{decl}(\texttt{"func"}, \mathbf{ID}(\texttt{name}), \texttt{params\_opt}, \texttt{func\_args}, \texttt{func\_body})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}}$$
$$\overbrace{\texttt{D\_Func}\left(\left\{\begin{array}{rcl} \text{name} & : & \texttt{name}, \\ \text{parameters} & : & \overline{\texttt{params\_opt}}, \\ \text{args} & : & \overline{\texttt{func\_args}}, \\ \text{body} & : & \texttt{SB\_ASL}(\overline{\texttt{func\_body}}), \\ \text{return\_type} & : & \texttt{None}, \\ \text{subprogram\_type} & : & \texttt{ST\_Procedure} \end{array}\right\}\right)}^{\texttt{ast\_node}}$$

GETTER

$$\mathit{build\_decl}\left(\overbrace{\texttt{decl}\left(\begin{array}{l}\texttt{"getter"}, \mathbf{ID}(\texttt{name}), \texttt{params\_opt}, \texttt{access\_args}, \\ \hookrightarrow \texttt{return\_type}, \texttt{func\_body}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}}$$
$$\overbrace{\texttt{D\_Func}\left(\left\{\begin{array}{rcl} \text{name} & : & \texttt{name}, \\ \text{parameters} & : & \overline{\texttt{params\_opt}}, \\ \text{args} & : & \overline{\texttt{access\_args}}, \\ \text{body} & : & \texttt{SB\_ASL}(\overline{\texttt{func\_body}}), \\ \text{return\_type} & : & \langle\overline{\texttt{return\_type}}\rangle, \\ \text{subprogram\_type} & : & \texttt{ST\_Getter} \end{array}\right\}\right)}^{\texttt{ast\_node}}$$

NO_ARG_GETTER

$$\mathit{build\_decl}(\overbrace{\texttt{decl}(\texttt{"getter"}, \textbf{ID}(\texttt{name}), \texttt{return\_type}, \texttt{func\_body})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{D\_Func}\left(\left\{\begin{array}{rcl} \text{name} & : & \texttt{name}, \\ \text{parameters} & : & [\,], \\ \text{args} & : & [\,], \\ \text{body} & : & \text{SB\_ASL}(\overline{\texttt{func\_body}}), \\ \text{return\_type} & : & \langle\overline{\texttt{return\_type}}\rangle, \\ \text{subprogram\_type} & : & \texttt{ST\_EmptyGetter} \end{array}\right\}\right)}^{\texttt{ast\_node}}$$

SETTER

$$\mathit{build\_decl}\left(\overbrace{\texttt{decl}\left(\begin{array}{l}\texttt{"setter"}, \textbf{ID}(\texttt{name}), \texttt{params\_opt}, \texttt{access\_args}, \texttt{"="}, \\ \hookrightarrow \texttt{v} : \texttt{typed\_identifier}, \texttt{func\_body}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\texttt{D\_Func}\left(\left\{\begin{array}{rcl} \text{name} & : & \texttt{name}, \\ \text{parameters} & : & \overline{\texttt{params\_opt}}, \\ \text{args} & : & [\texttt{v}] + \overline{\texttt{access\_args}}, \\ \text{body} & : & \text{SB\_ASL}(\overline{\texttt{func\_body}}), \\ \text{return\_type} & : & \texttt{None}, \\ \text{subprogram\_type} & : & \texttt{ST\_Setter} \end{array}\right\}\right)}^{\texttt{ast\_node}}$$

NO_ARG_SETTER

$$\mathit{build\_decl}(\overbrace{\texttt{decl}(\texttt{"setter"}, \textbf{ID}(\texttt{name}), \texttt{"="}, \texttt{v} : \texttt{typed\_identifier}, \texttt{func\_body})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{D\_Func}\left(\left\{\begin{array}{rcl} \text{name} & : & \texttt{name}, \\ \text{parameters} & : & [\,], \\ \text{args} & : & [\texttt{v}], \\ \text{body} & : & \text{SB\_ASL}(\overline{\texttt{func\_body}}), \\ \text{return\_type} & : & \texttt{None}, \\ \text{subprogram\_type} & : & \texttt{ST\_EmptySetter} \end{array}\right\}\right)}^{\texttt{ast\_node}}$$

TYPE_DECL

$$\mathit{build\_decl}(\overbrace{\texttt{decl}(\texttt{"type"}, \textbf{ID}(\texttt{x}), \texttt{"of"}, \texttt{ty\_decl}, \texttt{subtype\_opt}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{D\_TypeDecl}(\texttt{x}, \overline{\texttt{t}}, \overline{\texttt{subtype\_opt}})}^{\texttt{ast\_node}}$$

SUBTYPE_DECL

$$\frac{\mathit{build\_subtype}(\texttt{subtype}) \xrightarrow{\text{ast}} \texttt{s} \qquad \texttt{s} \overset{\text{is}}{=} (\texttt{name}, \texttt{fields})}{\mathit{build\_decl}(\overbrace{\texttt{decl}(\texttt{"type"}, \mathbf{ID}(\texttt{x}), \texttt{"of"}, \texttt{subtype}, \texttt{";"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{D\_TypeDecl}(\texttt{x}, \texttt{T\_Named}(\texttt{name}), \langle(\texttt{name}, \texttt{fields})\rangle)}^{\text{ast\_node}}}$$

GLOBAL_STORAGE

$$\frac{\begin{array}{c}\mathit{build\_storage\_keyword}(\texttt{keyword}) \xrightarrow{\text{ast}} \overline{\texttt{keyword}} \qquad \mathit{build\_option}[\mathit{build\_as\_ty}](\texttt{ty}) \xrightarrow{\text{ast}} \texttt{ty'} \\ \mathit{build\_expr}(\texttt{initial\_value}) \xrightarrow{\text{type}} \overline{\texttt{initial\_value}}\end{array}}{\mathit{build\_decl}\left(\overbrace{\texttt{decl}\left(\begin{array}{l}\texttt{keyword} : \texttt{storage\_keyword}, \texttt{name} : \texttt{ignored\_or\_identifier}, \\ \hookrightarrow\ \texttt{ty} : \texttt{option}(\texttt{as\_ty}), \texttt{"="}, \texttt{initial\_value} : \texttt{expr}, \texttt{";"}\end{array}\right)}^{\text{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\texttt{D\_GlobalStorage}\left(\left\{\begin{array}{rcl}\text{keyword} & : & \overline{\texttt{keyword}}, \\ \text{name} & : & \overline{\texttt{name}}, \\ \text{ty} & : & \texttt{ty'}, \\ \text{initial\_value} & : & \overline{\texttt{initial\_value}}\end{array}\right\}\right)}^{\text{ast\_node}}}$$

GLOBAL_UNINIT_VAR

$$\frac{\mathit{build\_ignored\_or\_identifier}(\texttt{cname}) \xrightarrow{\text{ast}} \texttt{name}}{\mathit{build\_decl}(\overbrace{\texttt{decl}(\texttt{"var"}, \texttt{cname} : \texttt{ignored\_or\_identifier}, \texttt{as\_ty}, \texttt{";"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{D\_GlobalStorage}(\{\text{keyword} : \texttt{GDK\_Var}, \text{name} : \texttt{name}, \text{ty} : \langle\overline{\texttt{as\_ty}}\rangle, \text{initial\_value} : \texttt{None}\})}^{\text{ast\_node}}}$$

GLOBAL_PRAGMA

$$\mathit{build\_decl}(\overbrace{\texttt{decl}(\texttt{"pragma"}, \mathbf{ID}(\texttt{x}), \texttt{clist}^*(\texttt{expr}), \texttt{";"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{pragma\_node}}^{\text{ast\_node}}$$

8.6 SyntaxRule.Subtype

The function

$$\mathit{build\_subtype}(\overbrace{\mathbb{PARSE}[\texttt{subtype}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{(\texttt{identifier} \times (\texttt{identifier} \times \texttt{ty})^*)}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

WITH_FIELDS

$$\mathit{build\_subtype}(\overbrace{\texttt{subtype}(\texttt{"subtypes"}, \mathbf{ID}(\texttt{id}), \texttt{"with"}, \texttt{fields})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{(\texttt{id}, \overline{\texttt{fields}})}^{\texttt{ast\_node}}$$

NO_FIELDS

$$\mathit{build\_subtype}(\overbrace{\texttt{subtype}(\texttt{"subtypes"}, \mathbf{ID}(\texttt{id}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{(\texttt{id}, [\,])}^{\texttt{ast\_node}}$$

8.7 SyntaxRule.Subtypeopt

The function

$$\mathit{build\_subtype\_opt}(\overbrace{\mathbb{PARSE}[\texttt{subtype\_opt}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\langle(\mathsf{identifier} \times \langle(\mathsf{identifier} \times \mathsf{ty})^*\rangle)\rangle}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

SUBTYPE_OPT

$$\frac{\mathit{build\_option}[\texttt{subtype}](\texttt{subtype\_opt}) \xrightarrow{\text{ast}} \texttt{ast\_node}}{\mathit{build\_subtype\_opt}(\overbrace{\texttt{subtype\_opt}(\texttt{subtype\_opt} : \mathsf{option}(\texttt{subtype}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

8.8 SyntaxRule.TypedIdentifier

The function

$$\mathit{build\_typed\_identifier}(\overbrace{\mathbb{PARSE}[\texttt{typed\_identifier}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{(\mathsf{identifier} \times \mathsf{ty})}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\mathit{build\_typed\_identifier}(\overbrace{\texttt{typed\_identifier}(\mathbf{ID}(\texttt{id}), \texttt{as\_ty})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{(\texttt{id}, \overline{\texttt{as\_ty}})}^{\texttt{ast\_node}}$$

8.9 SyntaxRule.OptTypedIdentifier

The function

$$\mathit{build\_opt\_typed\_identifier}(\overbrace{\mathbb{PARSE}[\texttt{opt\_typed\_identifier}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{(\mathsf{identifier} \times \langle\mathsf{ty}\rangle)}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\mathit{build\_option}[\texttt{as\_ty}](\texttt{as\_ty\_opt}) \xrightarrow{\text{ast}} \texttt{as\_ty\_opt\_ast}}{\mathit{build\_opt\_typed\_identifier}(\overbrace{\texttt{typed\_identifier}(\textbf{ID}(\texttt{id}), \texttt{as\_ty\_opt} : \mathsf{option}(\texttt{as\_ty}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{(\texttt{id}, \texttt{as\_ty\_opt\_ast})}^{\texttt{ast\_node}}}$$

8.10 SyntaxRule.ReturnType

The function

$$\mathit{build\_return\_type}(\overbrace{\mathbb{PARSE}[\texttt{return\_type}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{ty}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\mathit{build\_return\_type}(\overbrace{\texttt{return\_type}(\texttt{"=>"}, \mathsf{ty})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\overline{\mathsf{ty}}}^{\texttt{ast\_node}}$$

8.11 SyntaxRule.ParamsOpt

The function

$$\mathit{build\_params\_opt}(\overbrace{\mathbb{PARSE}[\texttt{params\_opt}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{(\mathsf{identifier} \times \langle \mathsf{ty} \rangle)^*}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

EMPTY

$$\mathit{build\_params\_opt}(\overbrace{\texttt{params\_opt}(\mathsf{epsilon\_node})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{[\,]}^{\texttt{ast\_node}}$$

NON_EMPTY

$$\frac{\mathit{build\_clist}[\texttt{opt\_typed\_identifier}](\texttt{ids}) \xrightarrow{\text{ast}} \texttt{ids\_ast}}{\mathit{build\_params\_opt}(\overbrace{\texttt{params\_opt}(\texttt{"\{"}, \texttt{ids} : \mathsf{clist}^*(\texttt{opt\_typed\_identifier}), \texttt{"\}"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{ids\_ast}}^{\texttt{ast\_node}}}$$

8.12 SyntaxRule.AccessArgs

The function

$$\mathit{build\_access\_args}(\overbrace{\mathbb{PARSE}[\texttt{access\_args}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{(\mathsf{identifier} \times \mathsf{ty})^*}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\mathit{build\_clist}[\texttt{typed\_identifier}](\texttt{ids}) \xrightarrow{\text{ast}} \texttt{ids\_ast}}{\mathit{build\_access\_args}(\overbrace{\texttt{access\_args}(\texttt{"["}, \texttt{ids} : \mathsf{clist}^*(\texttt{typed\_identifier}), \texttt{"]"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{ids\_ast}}^{\text{ast\_node}}}$$

8.13 SyntaxRule.FuncArgs

The function

$$\mathit{build\_func\_args}(\overbrace{\mathbb{PARSE}[\texttt{func\_args}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{(\mathsf{identifier} \times \mathsf{ty})^*}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\mathit{build\_clist}[\texttt{typed\_identifier}](\texttt{ids}) \xrightarrow{\text{ast}} \texttt{ids\_ast}}{\mathit{build\_func\_args}(\overbrace{\texttt{func\_args}(\texttt{"("}, \texttt{ids} : \mathsf{clist}^*(\texttt{typed\_identifier}), \texttt{")"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{ids\_ast}}^{\text{ast\_node}}}$$

8.14 SyntaxRule.MaybeEmptyStmtList

The function

$$\mathit{build\_maybe\_empty\_stmt\_list}(\overbrace{\mathbb{PARSE}[\texttt{maybe\_empty\_stmt\_list}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

EMPTY

$$\mathit{build\_maybe\_empty\_stmt\_list}(\overbrace{\texttt{maybe\_empty\_stmt\_list}(\mathsf{epsilon\_node})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Pass}}^{\text{ast\_node}}$$

NON_EMPTY

$$\mathit{build\_maybe\_empty\_stmt\_list}(\overbrace{\texttt{maybe\_empty\_stmt\_list}(\texttt{stmt\_list})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{stmt\_list}}^{\text{ast\_node}}$$

8.15 SyntaxRule.FuncBody

The function

$$\mathit{build\_func\_args}(\overbrace{\mathbb{PARSE}[\texttt{func\_body}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\mathit{build\_func\_body}(\overbrace{\texttt{func\_body}(\texttt{"begin"}, \texttt{stmts} : \texttt{maybe\_empty\_stmt\_list}, \texttt{"end"})}^{\texttt{parsed\_node}}) \xrightarrow{\texttt{ast}} \overbrace{\texttt{maybe\_empty\_stmt\_list}}^{\texttt{ast\_node}}$$

8.16 SyntaxRule.IgnoredOrIdentifier

The relation

$$\mathit{build\_func\_args}(\overbrace{\mathbb{PARSE}[\texttt{ignored\_or\_identifier}]}^{\texttt{parsed\_node}}) \bigtimes \overbrace{\mathsf{identifier}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

DISCARD

$$\frac{\texttt{id} \in \mathsf{identifier} \text{ is fresh}}{\mathit{build\_ignored\_or\_identifier}(\overbrace{\texttt{ignored\_or\_identifier}(\texttt{"-"})}^{\texttt{parsed\_node}}) \xrightarrow{\texttt{ast}} \overbrace{\texttt{id}}^{\texttt{ast\_node}}}$$

ID

$$\mathit{build\_ignored\_or\_identifier}(\overbrace{\texttt{ignored\_or\_identifier}(\mathbf{ID}(\texttt{id}))}^{\texttt{parsed\_node}}) \xrightarrow{\texttt{ast}} \overbrace{\texttt{id}}^{\texttt{ast\_node}}$$

8.17 SyntaxRule.LocalDeclKeyword

The function

$$\mathit{build\_local\_decl\_keyword}(\overbrace{\mathbb{PARSE}[\texttt{local\_decl\_keyword}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{local\_decl\_keyword}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

LET

$$\mathit{build\_local\_decl\_keyword}(\overbrace{\texttt{local\_decl\_keyword}(\texttt{"let"})}^{\texttt{parsed\_node}}) \xrightarrow{\texttt{ast}} \overbrace{\mathsf{LDK\_Let}}^{\texttt{ast\_node}}$$

CONSTANT

$$\mathit{build\_local\_decl\_keyword}(\overbrace{\texttt{local\_decl\_keyword}(\texttt{"constant"})}^{\texttt{parsed\_node}}) \xrightarrow{\texttt{ast}} \overbrace{\mathsf{LDK\_Constant}}^{\texttt{ast\_node}}$$

8.18 SyntaxRule.StorageKeyword

The function

$$\mathit{build\_storage\_keyword}(\overbrace{\mathbb{PARSE}[\texttt{storage\_keyword}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{global\_decl\_keyword}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

LET

$$\mathit{build\_storage\_keyword}(\overbrace{\texttt{storage\_keyword}(\texttt{"let"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{GDK\_Let}}^{\texttt{ast\_node}}$$

CONSTANT

$$\mathit{build\_storage\_keyword}(\overbrace{\texttt{storage\_keyword}(\texttt{"constant"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{GDK\_Constant}}^{\texttt{ast\_node}}$$

VAR

$$\mathit{build\_storage\_keyword}(\overbrace{\texttt{storage\_keyword}(\texttt{"var"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{GDK\_Var}}^{\texttt{ast\_node}}$$

CONFIG

$$\mathit{build\_storage\_keyword}(\overbrace{\texttt{storage\_keyword}(\texttt{"config"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{GDK\_Config}}^{\texttt{ast\_node}}$$

8.19 SyntaxRule.Direction

The function

$$\mathit{build\_direction}(\overbrace{\mathbb{PARSE}[\texttt{direction}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{for\_direction}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

TO

$$\mathit{build\_direction}(\overbrace{\texttt{direction}(\texttt{"to"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{Up}}^{\texttt{ast\_node}}$$

DOWNTO

$$\mathit{build\_direction}(\overbrace{\texttt{direction}(\texttt{"downto"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{Down}}^{\texttt{ast\_node}}$$

8.20 SyntaxRule.Alt

The function

$$\mathit{build\_alt}(\overbrace{\mathbb{PARSE}[\texttt{alt}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{case\_alt}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

WHEN

$$\frac{\mathit{build\_option}[\mathit{build\_expr}](\texttt{where\_opt}) \xrightarrow{\text{ast}} \texttt{where\_ast}}{\mathit{build\_alt}\left(\overbrace{\texttt{alt}\left(\begin{array}{l}\texttt{"when"}, \texttt{pattern\_list},\\ \hookrightarrow\ \texttt{where\_opt} : \mathsf{option}(\texttt{"where"}, \texttt{expr}), \texttt{"=>"},\\ \hookrightarrow\ \texttt{stmts} : \texttt{stmt\_list}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\mathsf{case\_alt}(\text{pattern} : \overline{\texttt{pattern\_list}}, \text{where} : \texttt{where\_ast}, \text{stmt} : \overline{\texttt{stmt\_list}})}^{\texttt{ast\_node}}}$$

OTHERWISE

$$\frac{}{\mathit{build\_alt}(\overbrace{\mathsf{alt}(\texttt{"otherwise"}, \texttt{stmts} : \mathsf{stmt\_list})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{case\_alt}(\text{pattern} : \mathsf{Pattern\_All}, \text{where} : \mathsf{None}, \text{stmt} : \overline{\texttt{stmt\_list}})}^{\texttt{ast\_node}}}$$

8.21 SyntaxRule.OtherwiseOpt

The function

$$\mathit{build\_otherwise\_opt}(\overbrace{\mathbb{PARSE}[\texttt{otherwise\_opt}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{stmt}?}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\mathit{build\_option}[\mathit{build\_stmt\_list}](\texttt{v}) \xrightarrow{\text{ast}} \texttt{ast\_node}}{\mathit{build\_otherwise\_opt}(\overbrace{\texttt{otherwise\_opt}(\texttt{v} : \mathsf{option}(\texttt{"otherwise"}, \texttt{"=>"}, \mathsf{stmt\_list}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

8.22 SyntaxRule.Catcher

The function

$$\mathit{build\_catcher}(\overbrace{\mathbb{PARSE}[\texttt{catcher}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{catcher}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

NAMED

$$\mathit{build\_catcher}(\overbrace{\texttt{catcher}(\texttt{"when"}, \mathbf{ID}(\texttt{id}), \texttt{":"}, \texttt{ty}, \texttt{"=>"}, \texttt{stmt\_list})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{(\langle \texttt{id} \rangle, \overline{\texttt{ty}}, \overline{\texttt{stmt\_list}})}^{\texttt{ast\_node}}$$

UNNAMED

$$\mathit{build\_catcher}(\overbrace{\texttt{catcher}(\texttt{"when"}, \texttt{ty}, \texttt{"=>"}, \texttt{stmt\_list})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{(\texttt{None}, \overline{\texttt{ty}}, \overline{\texttt{stmt\_list}})}^{\texttt{ast\_node}}$$

8.23 SyntaxRule.Stmt

The function

$$\mathit{build\_stmt}(\overbrace{\mathbb{PARSE}[\texttt{stmt}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{stmt}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

IF

$$\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"if"}, \texttt{expr}, \texttt{"then"}, \texttt{stmt\_list}, \texttt{s\_else}, \texttt{"end"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{S\_Cond}(\overline{\texttt{expr}}, \overline{\texttt{stmt\_list}}, \overline{\texttt{else}})}^{\texttt{ast\_node}}$$

CASE

$$\frac{\mathit{build\_list}[\texttt{alt}](\texttt{alt\_list}) \xrightarrow{\text{ast}} \texttt{alt\_list\_ast}}{\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"case"}, \texttt{expr}, \texttt{"of"}, \texttt{alt\_list} : \mathsf{list}^{+}(\texttt{alt}), \texttt{"end"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{S\_Case}(\overline{\texttt{expr}}, \texttt{alt\_list\_ast})}^{\texttt{ast\_node}}}$$

WHILE

$$\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"while"}, \texttt{expr}, \texttt{"do"}, \texttt{stmt\_list}, \texttt{"end"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{S\_While}(\overline{\texttt{expr}}, \texttt{None}, \overline{\texttt{stmt\_list}})}^{\texttt{ast\_node}}$$

LOOPLIMIT_WHILE

$$\frac{\mathit{build\_expr}(\texttt{limit\_expr}) \xrightarrow{\text{ast}} \texttt{limit\_expr\_ast}}{\mathit{build\_stmt}\left(\overbrace{\texttt{stmt}\left(\begin{array}{r}\texttt{"@looplimit"},\texttt{"("},\texttt{limit\_expr} : \texttt{expr},\texttt{")"},\texttt{"while"},\\ \hookrightarrow\ \texttt{expr},\texttt{"do"},\texttt{stmt\_list},\texttt{"end"}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_While}(\overline{\texttt{expr}},\langle\texttt{limit\_expr\_ast}\rangle,\overline{\texttt{stmt\_list}})}^{\texttt{ast\_node}}}$$

FOR

$$\frac{\mathit{build\_expr}(\texttt{start\_e}) \xrightarrow{\text{ast}} \texttt{start\_e\_ast} \qquad \mathit{build\_expr}(\texttt{end\_e}) \xrightarrow{\text{ast}} \texttt{end\_e\_ast}}{\mathit{build\_stmt}\left(\overbrace{\texttt{stmt}\left(\begin{array}{l}\texttt{"for"},\mathbf{ID}(\texttt{index\_name}),\texttt{"="},\texttt{start\_e} : \texttt{expr},\texttt{direction},\\ \hookrightarrow\ \texttt{end\_e} : \texttt{expr},\texttt{"do"},\texttt{stmt\_list},\texttt{"end"}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_For}\left(\left\{\begin{array}{rcl}\text{index\_name} & : & \texttt{index\_name}\\ \text{start\_e} & : & \texttt{start\_e\_ast}\\ \text{end\_e} & : & \texttt{end\_e\_ast}\\ \text{body} & : & \overline{\texttt{stmt\_list}}\\ \text{limit} & : & \mathsf{None}\end{array}\right\}\right)}^{\texttt{ast\_node}}}$$

TRY

$$\frac{\mathit{build\_list}[\texttt{catcher}] \xrightarrow{\text{ast}} \texttt{catcher\_list\_ast}}{\mathit{build\_stmt}\left(\overbrace{\texttt{stmt}\left(\begin{array}{r}\texttt{"try"},\texttt{stmt\_list},\texttt{"catch"},\\ \hookrightarrow\ \texttt{catcher\_list} : \mathsf{list}^{+}(\texttt{catcher}),\\ \hookrightarrow\ \texttt{otherwise\_opt},\texttt{"end"}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Try}(\overline{\texttt{stmt\_list}},\texttt{catcher\_list\_ast},\overline{\texttt{otherwise\_opt}})}^{\texttt{ast\_node}}}$$

PASS

$$\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"pass"},\texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Pass}}^{\texttt{ast\_node}}$$

RETURN

$$\frac{\mathit{build\_option}[\texttt{expr}](\texttt{expr}) \xrightarrow{\text{ast}} \texttt{expr\_ast}}{\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"return"},\texttt{expr} : \mathsf{option}(\texttt{expr}),\texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Return}(\texttt{expr\_ast})}^{\texttt{ast\_node}}}$$

CALL

$$\frac{\mathsf{build\_plist}[\mathsf{expr}](\mathtt{args}) \xrightarrow{\mathsf{ast}} \mathtt{args\_ast}}{\mathit{build\_stmt}(\overbrace{\mathsf{stmt}(\mathbf{ID}(\mathtt{x}), \mathtt{args} : \mathsf{plist}^*(\mathsf{expr}), \texttt{";"})}^{\mathsf{parsed\_node}}) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{S\_Call}(\mathtt{x}, \mathtt{args\_ast})}^{\mathsf{ast\_node}}}$$

ASSERT

$$\mathit{build\_stmt}(\overbrace{\mathsf{stmt}(\texttt{"assert"}, \mathsf{expr}, \texttt{";"})}^{\mathsf{parsed\_node}}) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{S\_Assert}(\overline{\mathtt{expr}})}^{\mathsf{ast\_node}}$$

DECL

$$\mathit{build\_stmt}(\overbrace{\mathsf{stmt}(\mathtt{local\_decl\_keyword}, \mathtt{decl\_item}, \texttt{"="}, \mathsf{expr}, \texttt{";"})}^{\mathsf{parsed\_node}}) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{S\_Decl}(\overline{\mathtt{local\_decl\_keyword}}, \overline{\mathtt{decl\_item}}, \langle\overline{\mathtt{expr}}\rangle)}^{\mathsf{ast\_node}}$$

ASSIGNMENT

$$\mathit{build\_stmt}(\overbrace{\mathsf{stmt}(\mathtt{lexpr}, \texttt{"="}, \mathsf{expr}, \texttt{";"})}^{\mathsf{parsed\_node}}) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{S\_Assign}(\overline{\mathtt{lexpr}}, \overline{\mathtt{expr}})}^{\mathsf{ast\_node}}$$

VAR_DECL

$$\frac{\mathit{build\_option}[\mathit{build\_expr}](\mathtt{e}) \xrightarrow{\mathsf{ast}} \mathtt{e\_ast}}{\mathit{build\_stmt}(\overbrace{\mathsf{stmt}(\texttt{"var"}, \mathtt{decl\_item}, \mathtt{e} : \mathsf{option}(\texttt{"="}, \mathsf{expr}), \texttt{";"})}^{\mathsf{parsed\_node}}) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{S\_Decl}(\mathsf{LDK\_Var}, \overline{\mathtt{decl\_item}}, \mathtt{e\_ast})}^{\mathsf{ast\_node}}}$$

MULTI_VAR_DECL

$$\frac{\mathit{build\_clist}[\mathit{build\_identity}](\mathtt{ids}) \xrightarrow{\mathsf{ast}} \mathtt{ids\_ast} \qquad \mathtt{stmts} := [\mathtt{x} \in \mathtt{ids\_ast} : \mathsf{S\_Decl}(\mathsf{LDK\_Var}, \mathtt{x}, \overline{\mathtt{ty}})] \qquad \mathit{stmt\_from\_list}(\mathtt{stmts}) \xrightarrow{\mathsf{ast}} \mathtt{ast\_node}}{\mathit{build\_stmt}(\overbrace{\mathsf{stmt}(\texttt{"var"}, \mathtt{ids} : \mathsf{clist2}(\mathbf{ID}), \texttt{":"}, \mathtt{ty}, \texttt{";"})}^{\mathsf{parsed\_node}}) \xrightarrow{\mathsf{ast}} \mathtt{ast\_node}}$$

PRINT

$$\frac{\mathsf{build\_plist}[\mathsf{expr}](\mathtt{args}) \xrightarrow{\mathsf{ast}} \mathtt{args\_ast}}{\mathit{build\_stmt}(\overbrace{\mathsf{stmt}(\texttt{"print"}, \mathtt{args} : \mathsf{plist}^*(\mathsf{expr}), \texttt{";"})}^{\mathsf{parsed\_node}}) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{S\_Print}(\mathtt{args\_ast})}^{\mathsf{ast\_node}}}$$

REPEAT

$$\mathit{build\_stmt}(\overbrace{\mathsf{stmt}(\texttt{"repeat"}, \mathtt{stmt\_list}, \texttt{"until"}, \mathsf{expr}, \texttt{";"})}^{\mathsf{parsed\_node}}) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{S\_Repeat}(\overline{\mathtt{stmt\_list}}, \overline{\mathtt{expr}}, \mathsf{None})}^{\mathsf{ast\_node}}$$

LOOPLIMIT_REPEAT

$$\frac{\mathit{build\_expr}(\texttt{limit\_expr}) \xrightarrow{\text{ast}} \texttt{limit\_expr\_ast}}{\mathit{build\_stmt}\left(\overbrace{\texttt{stmt}\left(\begin{array}{l}\texttt{"@looplimit"},\texttt{"("},\texttt{limit\_expr} : \texttt{expr},\texttt{")"},\texttt{"repeat"},\\ \hookrightarrow\ \texttt{stmt\_list},\texttt{"until"},\texttt{expr},\texttt{";"}\end{array}\right)}^{\text{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Repeat}(\texttt{stmt\_list}, \overline{\texttt{expr}}, \langle\texttt{limit\_expr\_ast}\rangle)}^{\text{ast\_node}}}$$

THROW_SOME

$$\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"throw"}, \texttt{expr}, \texttt{";"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Throw}(\langle\overline{\texttt{expr}}\rangle)}^{\text{ast\_node}}$$

THROW_NONE

$$\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"throw"}, \texttt{";"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Throw}(\texttt{None})}^{\text{ast\_node}}$$

PRAGMA

$$\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"pragma"}, \mathbf{ID}, \mathsf{clist}^*(\texttt{expr}), \texttt{";"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{pragma\_node}}^{\text{ast\_node}}$$

8.24 SyntaxRule.StmtList

The function

$$\mathit{build\_stmt\_list}(\overbrace{\mathbb{PARSE}[\texttt{stmt\_list}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\texttt{stmt}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\mathit{build\_list}[\texttt{stmt}](\texttt{stmts}) \xrightarrow{\text{ast}} \texttt{stmt\_list} \qquad \mathit{stmt\_from\_list}(\texttt{stmt\_list}) \xrightarrow{\text{ast}} \texttt{ast\_node}}{\mathit{build\_stmt\_list}(\texttt{stmt\_list}(\texttt{stmts} : \mathsf{list}^+(\texttt{stmt}))) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

8.25 SyntaxRule.SElse

The function

$$\mathit{build\_s\_else}(\overbrace{\mathbb{PARSE}[\texttt{s\_else}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\texttt{stmt}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

ELSEIF

$$\mathit{build\_s\_else}(\texttt{s\_else}(\texttt{"elseif"}, \texttt{expr}, \texttt{"when"}, \texttt{stmt\_list}, \texttt{s\_else})) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Cond}(\overline{\texttt{expr}}, \overline{\texttt{stmt\_list}}, \overline{\texttt{s\_else}})}^{\text{ast\_node}}$$

PASS

$$\mathit{build\_s\_else}(\texttt{s\_else}(\texttt{"pass"})) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Pass}}^{\text{ast\_node}}$$

ELSE

$$\mathit{build\_s\_else}(\texttt{s\_else}(\texttt{"else"}, \texttt{stmt\_list})) \xrightarrow{\text{ast}} \overbrace{\overline{\texttt{stmt\_list}}}^{\text{ast\_node}}$$

8.26 SyntaxRule.LExpr

The function

$$\mathit{build\_lexpr}(\overbrace{\mathbb{PARSE}[\texttt{lexpr}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\mathsf{lexpr}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

LEXPR_ATOM

$$\mathit{build\_lexpr}(\texttt{lexpr}(\texttt{lexpr\_atom})) \xrightarrow{\text{ast}} \overbrace{\overline{\texttt{lexpr\_atom}}}^{\text{ast\_node}}$$

DISCARD

$$\mathit{build\_lexpr}(\texttt{lexpr}(\texttt{"-"})) \xrightarrow{\text{ast}} \overbrace{\mathsf{LE\_Discard}}^{\text{ast\_node}}$$

MULTI_LEXPR

$$\frac{\mathit{build\_clist}[\texttt{lexpr}](\texttt{lexprs}) \xrightarrow{\text{ast}} \texttt{lexpr\_asts}}{\mathit{build\_lexpr}(\texttt{lexpr}(\texttt{"("}, \texttt{lexprs} : \mathsf{clist}^{+}(\texttt{lexpr}), \texttt{")"})) \xrightarrow{\text{ast}} \overbrace{\mathsf{LE\_Destructuring}(\texttt{lexpr\_asts})}^{\text{ast\_node}}}$$

8.27 SyntaxRule.LExprAtom

The function

$$\mathit{build\_lexpr\_atom}(\overbrace{\mathbb{PARSE}[\texttt{lexpr\_atom}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\mathsf{lexpr}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

VAR

$$\mathit{build\_lexpr\_atom}(\texttt{lexpr}(\textbf{ID}(\texttt{id}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{LE\_Var}(\texttt{id})}^{\texttt{ast\_node}}$$

SLICE

$$\mathit{build\_lexpr\_atom}(\texttt{lexpr}(\texttt{lexpr\_atom}, \texttt{slices})) \xrightarrow{\text{ast}} \overbrace{\mathsf{LE\_Slice}(\overline{\texttt{lexpr\_atom}}, \overline{\texttt{slices}})}^{\texttt{ast\_node}}$$

SET_FIELD

$$\mathit{build\_lexpr\_atom}(\texttt{lexpr}(\texttt{lexpr\_atom}, \texttt{"."}, \textbf{ID}(\texttt{id}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{LE\_SetField}(\overline{\texttt{lexpr\_atom}}, \texttt{id})}^{\texttt{ast\_node}}$$

SET_FIELDS

$$\frac{\mathit{build\_clist}[\mathit{build\_identity}](\texttt{fields}) \xrightarrow{\text{ast}} \texttt{field\_asts}}{\mathit{build\_lexpr\_atom}(\texttt{lexpr}(\texttt{lexpr\_atom}, \texttt{"."}, \texttt{"["}, \texttt{fields} : \mathsf{clist}^*(\textbf{ID}), \texttt{"]"})) \xrightarrow{\text{ast}} \overbrace{\mathsf{LE\_SetFields}(\overline{\texttt{lexpr\_atom}}, \texttt{field\_asts})}^{\texttt{ast\_node}}}$$

CONCAT

$$\frac{\mathit{build\_clist}[\mathit{build\_lexpr\_atom}](\texttt{lexprs}) \xrightarrow{\text{ast}} \texttt{lexpr\_asts}}{\mathit{build\_lexpr\_atom}(\texttt{lexpr}(\texttt{"["}, \texttt{lexprs} : \mathsf{clist}^+(\texttt{lexpr\_atom}), \texttt{"]"})) \xrightarrow{\text{ast}} \overbrace{\mathsf{LE\_Concat}(\texttt{lexpr\_asts})}^{\texttt{ast\_node}}}$$

8.28 SyntaxRule.DeclItem

The function

$$\mathit{build\_decl\_item}(\overbrace{\mathbb{PARSE}[\texttt{decl\_item}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{local\_decl\_item}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

TYPED

$$\mathit{build\_decl\_item}(\texttt{decl\_item}(\texttt{untyped\_decl\_item}, \texttt{as\_ty})) \xrightarrow{\text{ast}} \overbrace{\mathsf{LDI\_Typed}(\overline{\texttt{untyped\_local\_decl\_item}}, \overline{\texttt{as\_ty}})}^{\texttt{ast\_node}}$$

UNTYPED

$$\mathit{build\_decl\_item}(\texttt{decl\_item}(\texttt{untyped\_decl\_item})) \xrightarrow{\text{ast}} \overbrace{\texttt{untyped\_local\_decl\_item}}^{\texttt{ast\_node}}$$

8.29 SyntaxRule.UntypedDeclItem

The function

$$\mathit{build\_untyped\_decl\_item}(\overbrace{\mathbb{PARSE}[\texttt{untyped\_decl\_item}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{local\_decl\_item}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

VAR

$$\mathit{build\_untyped\_decl\_item}(\texttt{untyped\_decl\_item}(\textbf{ID}(\texttt{id}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{LDI\_Var}(\texttt{id})}^{\texttt{ast\_node}}$$

DISCARD

$$\mathit{build\_untyped\_decl\_item}(\texttt{untyped\_decl\_item}(\texttt{"-"})) \xrightarrow{\text{ast}} \overbrace{\mathsf{LDI\_Discard}}^{\texttt{ast\_node}}$$

TUPLE

$$\frac{\mathit{build\_clist}[\mathit{build\_decl\_item}](\texttt{decls\_items}) \xrightarrow{\text{ast}} \texttt{decls\_item\_asts}}{\mathit{build\_untyped\_decl\_item}(\texttt{untyped\_decl\_item}(\texttt{decls\_items} : \mathsf{plist2}(\texttt{decl\_item}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{LDI\_Tuple}(\texttt{decls\_item\_asts})}^{\texttt{ast\_node}}}$$

8.30 SyntaxRule.IntConstraints

The function

$$\mathit{build\_int\_constraints}(\overbrace{\mathbb{PARSE}[\texttt{int\_constraints}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{int\_constraints}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\mathit{build\_clist}[\mathit{build\_int\_constraint}](\texttt{constraint\_asts}) \xrightarrow{\text{ast}} \texttt{constraint\_asts}}{\mathit{build\_int\_constraints}(\texttt{int\_constraints}(\texttt{"\{"}, \texttt{constraints} : \mathsf{clist}^{+}(\texttt{int\_constraint}), \texttt{"\}"})) \xrightarrow{\text{ast}} \overbrace{\mathsf{WellConstrained}(\texttt{constraint\_asts})}^{\texttt{ast\_node}}}$$

8.31 SyntaxRule.IntConstraintsopt

The function

$$\mathit{build\_int\_constraints\_opt}(\overbrace{\mathbb{PARSE}[\texttt{int\_constraints\_opt}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{int\_constraints}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

CONSTRAINED

$$\mathit{build\_int\_constraints\_opt}(\texttt{int\_constraints\_opt}(\texttt{int\_constraints})) \xrightarrow{\text{ast}} \overbrace{\texttt{int\_constraints}}^{\texttt{ast\_node}}$$

UNCONSTRAINED

$$\mathit{build\_int\_constraints\_opt}(\texttt{int\_constraints\_opt}(\epsilon)) \xrightarrow{\text{ast}} \overbrace{\mathsf{Unconstrained}}^{\texttt{ast\_node}}$$

8.32 SyntaxRule.IntConstraint

The function

$$\mathit{build\_int\_constraint}(\overbrace{\mathbb{PARSE}[\texttt{int\_constraint}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{int\_constraint}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

EXACT

$$\mathit{build\_int\_constraint}(\texttt{int\_constraint}(\texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\mathsf{Constraint\_Exact}(\overline{\texttt{expr}})}^{\texttt{ast\_node}}$$

RANGE

$$\frac{\mathit{build\_expr}(\texttt{from\_expr}) \xrightarrow{\text{ast}} \texttt{from\_expr\_ast} \qquad \mathit{build\_expr}(\texttt{to\_expr}) \xrightarrow{\text{ast}} \texttt{to\_expr\_ast}}{\mathit{build\_int\_constraint}(\texttt{int\_constraint}(\texttt{from\_expr} : \texttt{expr}, \texttt{".."}, \texttt{to\_expr} : \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\mathsf{Constraint\_Range}(\texttt{from\_expr\_ast}, \texttt{to\_expr\_ast})}^{\texttt{ast\_node}}}$$

8.33 SyntaxRule.ExprPattern

The function

$$\mathit{build\_expr\_pattern}(\overbrace{\mathbb{PARSE}[\texttt{expr\_pattern}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{expr}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

LITERAL

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{value})) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Literal}(\overline{\texttt{value}})}^{\texttt{ast\_node}}$$

VAR

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\textbf{ID}(\texttt{id}))) \xrightarrow{\text{ast}} \overbrace{\text{E\_Var}(\texttt{id})}^{\texttt{ast\_node}}$$

BINOP

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{binop}, \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\text{E\_Binop}(\overline{\texttt{expr\_pattern}}, \overline{\texttt{binop}}, \overline{\texttt{expr}})}^{\texttt{ast\_node}}$$

UNOP

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{unop}, \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\text{E\_Unop}(\overline{\texttt{unop}}, \overline{\texttt{expr}})}^{\texttt{ast\_node}}$$

COND

$$\frac{\begin{array}{c}\mathit{build\_expr}(\texttt{cond\_expr}) \xrightarrow{\text{ast}} \texttt{cond\_expr\_ast} \\ \mathit{build\_expr}(\texttt{then\_expr}) \xrightarrow{\text{ast}} \texttt{then\_expr\_ast}\end{array}}{\mathit{build\_expr\_pattern}\left(\texttt{expr\_pattern}\left(\begin{array}{l}\texttt{"if"}, \texttt{cond\_expr} : \texttt{expr}, \texttt{"then"}, \\ \hookrightarrow\ \texttt{then\_expr} : \texttt{expr}, \texttt{e\_else}\end{array}\right)\right) \xrightarrow{\text{ast}} \overbrace{\text{E\_Cond}(\texttt{cond\_expr\_ast}, \texttt{then\_expr\_ast}, \overline{\texttt{e\_else}})}^{\texttt{ast\_node}}}$$

CALL

$$\frac{\text{build\_plist}[\mathit{build\_expr}](\texttt{args}) \xrightarrow{\text{ast}} \texttt{expr\_asts}}{\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\textbf{ID}(\texttt{id}), \texttt{args} : \text{plist}^*(\texttt{expr}))) \xrightarrow{\text{ast}} \overbrace{\text{E\_Call}(\texttt{id}, \texttt{expr\_asts})}^{\texttt{ast\_node}}}$$

SLICE

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{slice})) \xrightarrow{\text{ast}} \overbrace{\text{E\_Slice}(\overline{\texttt{expr\_pattern}}, \overline{\texttt{slice}})}^{\texttt{ast\_node}}$$

GET_FIELD

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{"."}, \textbf{ID}(\texttt{id}))) \xrightarrow{\text{ast}} \overbrace{\text{E\_GetField}(\overline{\texttt{expr}}, \texttt{id})}^{\texttt{ast\_node}}$$

GET_FIELDS

$$\frac{\mathit{build\_clist}[\mathit{build\_identity}](\texttt{ids}) \xrightarrow{\text{ast}} \texttt{id\_asts}}{\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{"."}, \texttt{"["}, \texttt{ids} : \text{clist}^+(\textbf{ID}), \texttt{"]"})) \xrightarrow{\text{ast}} \overbrace{\text{E\_GetFields}(\overline{\texttt{expr\_pattern}}, \texttt{id\_asts})}^{\texttt{ast\_node}}}$$

CONCAT

$$\frac{\mathit{build\_clist}[\mathit{build\_expr}](\texttt{exprs}) \xrightarrow{\text{ast}} \texttt{expr\_asts}}{\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{"["}, \texttt{exprs} : \mathsf{clist}^{+}(\texttt{expr}), \texttt{"]"})) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Concat}(\texttt{expr\_asts})}^{\texttt{ast\_node}}}$$

ATC

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{"as"}, \texttt{ty})) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_ATC}(\overline{\texttt{expr\_pattern}}, \overline{\texttt{ty}})}^{\texttt{ast\_node}}$$

ATC_INT_CONSTRAINTS

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{"as"}, \texttt{int\_constraints})) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_ATC}(\overline{\texttt{expr\_pattern}}, \mathsf{T\_Int}(\overline{\texttt{int\_constraints}}))}^{\texttt{ast\_node}}$$

PATTERN_SET

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{"IN"}, \texttt{pattern\_set})) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Pattern}(\overline{\texttt{expr\_pattern}}, \overline{\texttt{pattern\_set}})}^{\texttt{ast\_node}}$$

PATTERN_MASK

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{"IN"}, \mathbf{MASK\_LIT}(\texttt{m}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Pattern}(\overline{\texttt{expr\_pattern}}, \mathsf{Pattern\_Mask}(\texttt{m}))}^{\texttt{ast\_node}}$$

UNKNOWN

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{"UNKNOWN"}, \texttt{":"}, \texttt{ty})) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Unknown}(\overline{\texttt{ty}})}^{\texttt{ast\_node}}$$

RECORD

$$\frac{\mathit{build\_clist}[\mathit{build\_field\_assign}](\texttt{field\_assigns}) \xrightarrow{\text{ast}} \texttt{field\_assign\_asts}}{\mathit{build\_expr\_pattern}\left(\texttt{expr\_pattern}\left(\begin{array}{l}\mathbf{ID}(\texttt{t}), \texttt{"\{"},\\ \hookrightarrow\ \texttt{field\_assigns} : \mathsf{clist}^{*}(\texttt{field\_assign}),\\ \hookrightarrow\ \texttt{"\}"}\end{array}\right)\right) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Record}(\mathsf{T\_Named}(\texttt{t}), \texttt{field\_assign\_asts})}^{\texttt{ast\_node}}}$$

SUB_EXPR

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{"("}, \texttt{expr\_pattern}, \texttt{")"})) \xrightarrow{\text{ast}} \overbrace{\overline{\texttt{expr\_pattern}}}^{\texttt{ast\_node}}$$

8.34 SyntaxRule.PatternSet

The function

$$\mathit{build\_pattern\_set}(\overbrace{\mathbb{PARSE}[\texttt{pattern\_set}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{pattern}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

NOT

$$\mathit{build\_pattern\_set}(\texttt{pattern\_set}(\texttt{"!"}, \texttt{"\{"}, \texttt{pattern\_list}, \texttt{"\}"})) \xrightarrow{\text{ast}} \overbrace{\mathsf{Pattern\_Not}(\texttt{pattern\_list})}^{\texttt{ast\_node}}$$

LIST

$$\mathit{build\_pattern\_set}(\texttt{pattern\_set}(\texttt{"\{"}, \texttt{pattern\_list}, \texttt{"\}"})) \xrightarrow{\text{ast}} \overbrace{\texttt{pattern\_list}}^{\texttt{ast\_node}}$$

8.35 SyntaxRule.PatternList

The function

$$\mathit{build\_pattern\_list}(\overbrace{\mathbb{PARSE}[\texttt{pattern\_list}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{pattern}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\mathit{build\_clist}[\mathit{build\_pattern}](\texttt{patterns}) \xrightarrow{\text{ast}} \texttt{pattern\_asts}}{\mathit{build\_pattern\_list}(\texttt{pattern\_list}(\texttt{patterns} : \mathsf{clist}^{+}(\mathsf{pattern}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{Pattern\_Any}(\texttt{pattern\_asts})}^{\texttt{ast\_node}}}$$

8.36 SyntaxRule.Pattern

The function

$$\mathit{build\_pattern}(\overbrace{\mathbb{PARSE}[\texttt{pattern}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{pattern}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

SINGLE

$$\mathit{build\_pattern}(\texttt{pattern}(\texttt{expr\_pattern})) \xrightarrow{\text{ast}} \overbrace{\mathsf{Pattern\_Single}(\texttt{expr\_pattern})}^{\texttt{ast\_node}}$$

RANGE

$$\mathit{build\_pattern}(\texttt{pattern}(\texttt{expr\_pattern}, \texttt{".."}, \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\text{Pattern\_Range}(\overline{\texttt{expr\_pattern}}, \overline{\texttt{expr}})}^{\text{ast\_node}}$$

ALL

$$\mathit{build\_pattern}(\texttt{pattern}(\texttt{"-"})) \xrightarrow{\text{ast}} \overbrace{\text{Pattern\_All}}^{\text{ast\_node}}$$

LEQ

$$\mathit{build\_pattern}(\texttt{pattern}(\texttt{"<="}, \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\text{Pattern\_Leq}(\overline{\texttt{expr}})}^{\text{ast\_node}}$$

GEQ

$$\mathit{build\_pattern}(\texttt{pattern}(\texttt{">="}, \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\text{Pattern\_Geq}(\overline{\texttt{expr}})}^{\text{ast\_node}}$$

MASK

$$\mathit{build\_pattern}(\texttt{pattern}(\textbf{MASK\_LIT}(\texttt{m}))) \xrightarrow{\text{ast}} \overbrace{\text{Pattern\_Mask}(\texttt{m})}^{\text{ast\_node}}$$

TUPLE

$$\frac{\text{build\_plist}[\mathit{build\_pattern}](\texttt{patterns}) \xrightarrow{\text{ast}} \texttt{pattern\_asts}}{\mathit{build\_pattern}(\texttt{pattern}(\texttt{patterns} : \text{plist2}(\texttt{pattern}))) \xrightarrow{\text{ast}} \overbrace{\text{Pattern\_Tuple}(\texttt{pattern\_asts})}^{\text{ast\_node}}}$$

SET

$$\mathit{build\_pattern}(\texttt{pattern}(\texttt{pattern\_set})) \xrightarrow{\text{ast}} \overbrace{\overline{\texttt{pattern\_set}}}^{\text{ast\_node}}$$

8.37 SyntaxRule.Fields

The function

$$\mathit{build\_fields}(\overbrace{\mathbb{PARSE}[\texttt{fields}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{field}^*}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\mathit{build\_tclist}[\mathit{build\_typed\_identifier}](\texttt{fields}) \xrightarrow{\text{ast}} \texttt{field\_asts}}{\mathit{build\_fields}(\texttt{fields}(\texttt{"\{"}, \texttt{fields} : \text{tclist}^*(\texttt{typed\_identifier}), \texttt{"\}"})) \xrightarrow{\text{ast}} \overbrace{\texttt{field\_asts}}^{\text{ast\_node}}}$$

8.38 SyntaxRule.FieldsOpt

The function

$$\mathit{build\_fields\_opt}(\overbrace{\mathbb{PARSE}[\texttt{fields\_opt}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{field}^*}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

FIELDS

$$\mathit{build\_fields\_opt}(\texttt{fields\_opt}(\texttt{fields})) \xrightarrow{\text{ast}} \overbrace{\texttt{fields}}^{\texttt{ast\_node}}$$

EMPTY

$$\mathit{build\_fields\_opt}(\texttt{fields\_opt}(\epsilon)) \xrightarrow{\text{ast}} \overbrace{[\,]}^{\texttt{ast\_node}}$$

8.39 SyntaxRule.NSlices

The function

$$\mathit{build\_nslices}(\overbrace{\mathbb{PARSE}[\texttt{nslices}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{slice}^+}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\mathit{build\_clist}[\mathit{build\_slice}](\texttt{slices}) \xrightarrow{\text{ast}} \texttt{slice\_asts}}{\mathit{build\_nslices}(\texttt{nslices}(\texttt{"["}, \texttt{slices} : \mathsf{clist}^+(\texttt{slice}), \texttt{"]"})) \xrightarrow{\text{ast}} \overbrace{\texttt{slice\_asts}}^{\texttt{ast\_node}}}$$

8.40 SyntaxRule.Slices

The function

$$\mathit{build\_slices}(\overbrace{\mathbb{PARSE}[\texttt{slices}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{slice}^*}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\mathit{build\_clist}[\mathit{build\_slice}](\texttt{slices}) \xrightarrow{\text{ast}} \texttt{slice\_asts}}{\mathit{build\_slices}(\texttt{slices}(\texttt{"["}, \texttt{slices} : \mathsf{clist}^*(\texttt{slice}), \texttt{"]"})) \xrightarrow{\text{ast}} \overbrace{\texttt{slice\_asts}}^{\texttt{ast\_node}}}$$

8.41 SyntaxRule.Slice

The function

$$\mathit{build\_slice}(\overbrace{\mathbb{PARSE}[\texttt{slice}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{slice}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

SINGLE

$$\mathit{build\_slices}(\texttt{slice}(\texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\texttt{Slice\_Single}(\overline{\texttt{expr}})}^{\texttt{ast\_node}}$$

RANGE

$$\frac{\mathit{build\_expr}(\texttt{e1}) \xrightarrow{\text{ast}} \texttt{e1\_ast} \qquad \mathit{build\_expr}(\texttt{e2}) \xrightarrow{\text{ast}} \texttt{e2\_ast}}{\mathit{build\_slices}(\texttt{slice}(\texttt{e1} : \texttt{expr}, \texttt{":"}, \texttt{e2} : \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\texttt{Slice\_Range}(\texttt{e1\_ast}, \texttt{e2\_ast})}^{\texttt{ast\_node}}}$$

LENGTH

$$\frac{\mathit{build\_expr}(\texttt{e1}) \xrightarrow{\text{ast}} \texttt{e1\_ast} \qquad \mathit{build\_expr}(\texttt{e2}) \xrightarrow{\text{ast}} \texttt{e2\_ast}}{\mathit{build\_slices}(\texttt{slice}(\texttt{e1} : \texttt{expr}, \texttt{"+:"}, \texttt{e2} : \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\texttt{Slice\_Length}(\texttt{e1\_ast}, \texttt{e2\_ast})}^{\texttt{ast\_node}}}$$

STAR

$$\frac{\mathit{build\_expr}(\texttt{e1}) \xrightarrow{\text{ast}} \texttt{e1\_ast} \qquad \mathit{build\_expr}(\texttt{e2}) \xrightarrow{\text{ast}} \texttt{e2\_ast}}{\mathit{build\_slices}(\texttt{slice}(\texttt{e1} : \texttt{expr}, \texttt{"*:"}, \texttt{e2} : \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\texttt{Slice\_Star}(\texttt{e1\_ast}, \texttt{e2\_ast})}^{\texttt{ast\_node}}}$$

8.42 SyntaxRule.Bitfields

The function

$$\mathit{build\_bitfields}(\overbrace{\mathbb{PARSE}[\texttt{bitfields}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{bitfield}^*}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\mathit{build\_tclist}[\mathit{build\_bitfield}](\texttt{bitfields}) \xrightarrow{\text{ast}} \texttt{bitfield\_asts}}{\mathit{build\_bitfields}(\texttt{bitfields}(\texttt{"\{"}, \texttt{bitfields} : \texttt{tclist}^*(\texttt{bitfield}), \texttt{"\}"})) \xrightarrow{\text{ast}} \overbrace{\texttt{bitfield\_asts}}^{\texttt{ast\_node}}}$$

8.43 SyntaxRule.Bitfield

The function

$$\mathit{build\_bitfield}(\overbrace{\mathbb{PARSE}[\texttt{bitfield}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{bitfield}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

SIMPLE

$$\mathit{build\_bitfields}(\texttt{bitfield}(\texttt{nslices}, \mathbf{ID}(\texttt{x}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{BitField\_Simple}(\texttt{x}, \overline{\texttt{slices}})}^{\texttt{ast\_node}}$$

NESTED

$$\mathit{build\_bitfields}(\texttt{bitfield}(\texttt{nslices}, \mathbf{ID}(\texttt{x}), \texttt{bitfields})) \xrightarrow{\text{ast}} \overbrace{\mathsf{BitField\_Nested}(\texttt{x}, \overline{\texttt{slices}}, \overline{\texttt{bitfields}})}^{\texttt{ast\_node}}$$

TYPE

$$\mathit{build\_bitfields}(\texttt{bitfield}(\texttt{nslices}, \mathbf{ID}(\texttt{x}), \texttt{":"}, \texttt{ty})) \xrightarrow{\text{ast}} \overbrace{\mathsf{BitField\_Type}(\texttt{x}, \overline{\texttt{slices}}, \overline{\texttt{ty}})}^{\texttt{ast\_node}}$$

8.44 SyntaxRule.Ty

The function

$$\mathit{build\_ty}(\overbrace{\mathbb{PARSE}[\texttt{ty}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{ty}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

INTEGER

$$\mathit{build\_ty}(\texttt{ty}(\texttt{"integer"}, \texttt{int\_constraints\_opt})) \xrightarrow{\text{ast}} \overbrace{\mathsf{T\_Int}(\overline{\texttt{intconstraints\_opt}})}^{\texttt{ast\_node}}$$

REAL

$$\mathit{build\_ty}(\texttt{ty}(\texttt{"real"})) \xrightarrow{\text{ast}} \overbrace{\mathsf{T\_Real}}^{\texttt{ast\_node}}$$

BOOLEAN

$$\mathit{build\_ty}(\texttt{ty}(\texttt{"boolean"})) \xrightarrow{\text{ast}} \overbrace{\mathsf{T\_Bool}}^{\texttt{ast\_node}}$$

STRING

$$\mathit{build\_ty}(\texttt{ty}(\texttt{"string"})) \xrightarrow{\text{ast}} \overbrace{\mathsf{T\_String}}^{\texttt{ast\_node}}$$

BIT

$$\mathit{build\_ty}(\texttt{ty}(\texttt{"bit"})) \xrightarrow{\text{ast}} \overbrace{\mathsf{T\_Bits}(\mathsf{E\_Literal}(\mathsf{L\_Int}(1)), [\,])}^{\texttt{ast\_node}}$$

BITS

$$\frac{\mathit{build\_list}[\mathit{build\_bitfield}](\texttt{bitfields}) \xrightarrow{\text{ast}} \texttt{bitfield\_asts}}{\mathit{build\_ty}(\texttt{ty}(\texttt{"bits"}, \texttt{"("}, \texttt{expr}, \texttt{")"}, \texttt{bitfields} : \mathsf{list}^*(\texttt{bitfields}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{T\_Bits}(\overline{\texttt{expr}}, \texttt{bitfield\_asts})}^{\texttt{ast\_node}}}$$

TUPLE

$$\frac{\mathsf{build\_plist}[\mathit{build\_ty}](\texttt{types}) \xrightarrow{\text{ast}} \texttt{type\_asts}}{\mathit{build\_ty}(\texttt{ty}(\texttt{types} : \mathsf{plist}^*(\texttt{ty}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{T\_Tuple}(\texttt{type\_asts})}^{\texttt{ast\_node}}}$$

NAMED

$$\mathit{build\_ty}(\texttt{ty}(\mathbf{ID}(\texttt{id}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{T\_Named}(\texttt{id})}^{\texttt{ast\_node}}$$

ARRAY

$$\mathit{build\_ty}(\texttt{ty}(\texttt{"array"}, \texttt{"["}, \texttt{expr}, \texttt{"]"}, \texttt{"of"}, \texttt{ty})) \xrightarrow{\text{ast}} \overbrace{\mathsf{T\_Array}(\mathsf{ArrayLength\_Expr}(\overline{\texttt{expr}}), \overline{\texttt{ty}})}^{\texttt{ast\_node}}$$

8.45 SyntaxRule.TyDecl

The function

$$\mathit{build\_ty\_decl}(\overbrace{\mathbb{PARSE}[\texttt{ty\_decl}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{ty}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

TY

$$\mathit{build\_ty\_decl}(\texttt{ty\_decl}(\texttt{ty})) \xrightarrow{\text{ast}} \overbrace{\overline{\texttt{ty}}}^{\texttt{ast\_node}}$$

ENUMERATION

$$\frac{\mathit{build\_tclist}[\mathit{build\_identity}](\texttt{ids}) \xrightarrow{\text{ast}} \texttt{id\_asts}}{\mathit{build\_ty\_decl}(\texttt{ty\_decl}(\texttt{"enumeration"}, \texttt{"\{"}, \texttt{ids} : \mathsf{ntclist}(\mathbf{ID}), \texttt{"\}"})) \xrightarrow{\text{ast}} \overbrace{\mathsf{T\_Enum}(\texttt{id\_asts})}^{\texttt{ast\_node}}}$$

RECORD

$$\mathit{build\_ty\_decl}(\texttt{ty\_decl}(\texttt{"record"}, \texttt{fields\_opt})) \xrightarrow{\text{ast}} \overbrace{\mathsf{T\_Record}(\overline{\texttt{fields\_opt}})}^{\texttt{ast\_node}}$$

EXCEPTION

$$\mathit{build\_ty\_decl}(\texttt{ty\_decl}(\texttt{"exception"}, \texttt{fields\_opt})) \xrightarrow{\text{ast}} \overbrace{\mathsf{T\_Exception}(\overline{\texttt{fields\_opt}})}^{\texttt{ast\_node}}$$

8.46 SyntaxRule.FieldAssign

The function

$$\mathit{build\_field\_assign}(\overbrace{\mathbb{PARSE}[\texttt{field\_assign}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{(\mathsf{identifier} \times \mathsf{expr})}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\mathit{build\_field\_assign}(\texttt{field\_assign}(\mathbf{ID}(\texttt{id}), \texttt{"="}, \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{(\texttt{id}, \overline{\texttt{expr}})}^{\texttt{ast\_node}}$$

8.47 SyntaxRule.EElse

The function

$$\mathit{build\_e\_else}(\overbrace{\mathbb{PARSE}[\texttt{field\_assign}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{expr}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

ELSE

$$\mathit{build\_e\_else}(\texttt{e\_else}(\texttt{"else"}, \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\overline{\texttt{expr}}}^{\texttt{ast\_node}}$$

ELSE_IF

$$\frac{\begin{array}{c}\mathit{build\_expr}(\texttt{cond\_expr}) \xrightarrow{\text{ast}} \texttt{cond\_expr\_ast} \\ \mathit{build\_expr}(\texttt{then\_expr}) \xrightarrow{\text{ast}} \texttt{then\_expr\_ast}\end{array}}{\mathit{build\_e\_else}\left(\texttt{e\_else}\left(\begin{array}{l}\texttt{"elseif"}, \texttt{cond\_expr} : \mathsf{expr}, \\ \hookrightarrow\ \texttt{"then"}, \texttt{then\_expr} : \mathsf{expr}, \texttt{e\_else}\end{array}\right)\right) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Cond}(\texttt{cond\_expr\_ast}, \texttt{then\_expr\_ast}, \overline{\texttt{e\_else}})}^{\texttt{ast\_node}}}$$

8.48 SyntaxRule.Expr

The function

$$\mathit{build\_expr}(\overbrace{\mathbb{PARSE}[\texttt{expr}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{expr}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

LITERAL

$$\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{value})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_Literal}(\overline{\texttt{value}})}^{\texttt{ast\_node}}$$

VAR

$$\mathit{build\_expr}(\overbrace{\texttt{expr}(\textbf{ID}(\texttt{id}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_Var}(\texttt{id})}^{\texttt{ast\_node}}$$

BINOP

$$\frac{\mathit{build\_expr}(\texttt{e1}) \xrightarrow{\text{ast}} \texttt{e1\_ast} \qquad \mathit{build\_expr}(\texttt{e2}) \xrightarrow{\text{ast}} \texttt{e2\_ast}}{\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{e1} : \texttt{expr}, \texttt{binop}, \texttt{e2} : \texttt{expr})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_Binop}(\texttt{e1\_ast}, \overline{\texttt{binop}}, \texttt{e2\_ast})}^{\texttt{ast\_node}}}$$

UNOP

$$\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{unop}, \texttt{expr})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_Unop}(\overline{\texttt{unop}}, \overline{\texttt{expr}})}^{\texttt{ast\_node}}$$

COND

$$\frac{\begin{array}{c}\mathit{build\_expr}(\texttt{cond\_expr}) \xrightarrow{\text{ast}} \texttt{cond\_expr\_ast} \\ \mathit{build\_expr}(\texttt{then\_expr}) \xrightarrow{\text{ast}} \texttt{then\_expr\_ast}\end{array}}{\mathit{build\_expr}\left(\overbrace{\texttt{expr}\left(\begin{array}{l}\texttt{"if"}, \texttt{cond\_expr} : \texttt{expr}, \texttt{"then"}, \\ \hookrightarrow\ \texttt{then\_expr} : \texttt{expr}, \texttt{e\_else}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_Cond}(\texttt{cond\_expr\_ast}, \texttt{then\_expr\_ast}, \overline{\texttt{e\_else}})}^{\texttt{ast\_node}}}$$

CALL

$$\frac{\texttt{build\_plist}[\mathit{build\_expr}](\texttt{args}) \xrightarrow{\text{ast}} \texttt{expr\_asts}}{\mathit{build\_expr}(\overbrace{\texttt{expr}(\textbf{ID}(\texttt{id}), \texttt{args} : \texttt{plist}^*(\texttt{expr}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_Call}(\texttt{id}, \texttt{expr\_asts})}^{\texttt{ast\_node}}}$$

SLICE

$$\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{expr}, \texttt{slice})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Slice}(\overline{\texttt{expr}}, \overline{\texttt{slice}})}^{\texttt{ast\_node}}$$

GET_FIELD

$$\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{expr}, \texttt{"."}, \mathbf{ID}(\texttt{id}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_GetField}(\overline{\texttt{expr}}, \texttt{id})}^{\texttt{ast\_node}}$$

GET_FIELDS

$$\frac{\mathit{build\_clist}[\mathit{build\_identity}](\texttt{ids}) \xrightarrow{\text{ast}} \texttt{id\_asts}}{\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{expr}, \texttt{"."}, \texttt{"["}, \texttt{ids} : \mathsf{clist}^{+}(\mathbf{ID}), \texttt{"]"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_GetFields}(\overline{\texttt{expr}}, \texttt{id\_asts})}^{\texttt{ast\_node}}}$$

CONCAT

$$\frac{\mathit{build\_clist}[\mathit{build\_expr}](\texttt{exprs}) \xrightarrow{\text{ast}} \texttt{expr\_asts}}{\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{"["}, \texttt{exprs} : \mathsf{clist}^{+}(\texttt{expr}), \texttt{"]"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Concat}(\texttt{expr\_asts})}^{\texttt{ast\_node}}}$$

ATC

$$\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{expr}, \texttt{"as"}, \texttt{ty})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_ATC}(\overline{\texttt{expr}}, \overline{\texttt{ty}})}^{\texttt{ast\_node}}$$

ATC_INT_CONSTRAINTS

$$\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{expr}, \texttt{"as"}, \texttt{int\_constraints})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_ATC}(\overline{\texttt{expr}}, \mathsf{T\_Int}(\overline{\texttt{int\_constraints}}))}^{\texttt{ast\_node}}$$

PATTERN_SET

$$\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{expr}, \texttt{"IN"}, \texttt{pattern\_set})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Pattern}(\overline{\texttt{expr}}, \overline{\texttt{pattern\_set}})}^{\texttt{ast\_node}}$$

PATTERN_MASK

$$\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{expr}, \texttt{"IN"}, \textbf{MASK\_LIT}(\texttt{m}))}^{\texttt{parsed\_node}}) \xrightarrow{\texttt{ast}} \overbrace{\mathsf{E\_Pattern}(\overline{\texttt{expr}}, \mathsf{Pattern\_Mask}(\texttt{m}))}^{\texttt{ast\_node}}$$

UNKNOWN

$$\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{"UNKNOWN"}, \texttt{":"}, \texttt{ty})}^{\texttt{parsed\_node}}) \xrightarrow{\texttt{ast}} \overbrace{\mathsf{E\_Unknown}(\overline{\texttt{ty}})}^{\texttt{ast\_node}}$$

RECORD

$$\frac{\mathit{build\_clist}[\mathit{build\_field\_assign}](\texttt{field\_assigns}) \xrightarrow{\texttt{ast}} \texttt{field\_assign\_asts}}{\mathit{build\_expr}\left(\overbrace{\texttt{expr}\left(\begin{array}{l}\textbf{ID}(\texttt{t}), \texttt{"\{"},\\ \hookrightarrow\ \texttt{field\_assigns} : \mathsf{clist}^*(\texttt{field\_assign}),\\ \hookrightarrow\ \texttt{"\}"}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\texttt{ast}} \overbrace{\mathsf{E\_Record}(\mathsf{T\_Named}(\texttt{t}), \texttt{field\_assign\_asts})}^{\texttt{ast\_node}}}$$

SUB_EXPR

$$\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{"("}, \texttt{expr}, \texttt{")"})}^{\texttt{parsed\_node}}) \xrightarrow{\texttt{ast}} \overbrace{\overline{\texttt{expr}}}^{\texttt{ast\_node}}$$

TUPLE

$$\frac{\mathsf{build\_plist}[\mathit{build\_expr}](\texttt{exprs}) \xrightarrow{\texttt{ast}} \texttt{expr\_asts}}{\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{exprs} : \mathsf{plist2}(\texttt{expr}))}^{\texttt{parsed\_node}}) \xrightarrow{\texttt{ast}} \overbrace{\mathsf{E\_Tuple}(\texttt{expr\_asts})}^{\texttt{ast\_node}}}$$

8.49 SyntaxRule.Value

The function

$$\mathit{build\_value}(\overbrace{\mathbb{PARSE}[\texttt{value}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{literal}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

INTEGER

$$\mathit{build\_value}(\texttt{value}(\textbf{INT\_LIT}(\texttt{i}))) \xrightarrow{\texttt{ast}} \overbrace{\mathsf{L\_Int}(\texttt{i})}^{\texttt{ast\_node}}$$

BOOLEAN

$$\mathit{build\_value}(\texttt{value}(\textbf{BOOL\_LIT}(\texttt{b}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{L\_Bool}(\texttt{b})}^{\texttt{ast\_node}}$$

REAL

$$\mathit{build\_value}(\texttt{value}(\textbf{REAL\_LIT}(\texttt{r}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{L\_Real}(\texttt{r})}^{\texttt{ast\_node}}$$

BITVECTOR

$$\mathit{build\_value}(\texttt{value}(\textbf{BITVECTOR\_LIT}(\texttt{b}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{L\_Bitvector}(\texttt{b})}^{\texttt{ast\_node}}$$

STRING

$$\mathit{build\_value}(\texttt{value}(\textbf{STRING\_LIT}(\texttt{s}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{L\_String}(\texttt{s})}^{\texttt{ast\_node}}$$

8.50 SyntaxRule.Unop

The function

$$\mathit{build\_unop}(\overbrace{\mathbb{PARSE}[\texttt{unop}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{unop}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

BNOT

$$\mathit{build\_unop}(\texttt{unop}(\texttt{"!"})) \xrightarrow{\text{ast}} \overbrace{\texttt{BNOT}}^{\texttt{ast\_node}}$$

NEG

$$\mathit{build\_unop}(\texttt{unop}(\texttt{"-"})) \xrightarrow{\text{ast}} \overbrace{\texttt{NEG}}^{\texttt{ast\_node}}$$

NOT

$$\mathit{build\_unop}(\texttt{unop}(\texttt{"NOT"})) \xrightarrow{\text{ast}} \overbrace{\texttt{NOT}}^{\texttt{ast\_node}}$$

8.51 SyntaxRule.Binop

The function

$$\mathit{build\_binop}(\overbrace{\mathbb{PARSE}[\texttt{binop}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{binop}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\textit{build\_binop}(\texttt{binop}(\texttt{"AND"})) \xrightarrow{\text{ast}} \overbrace{\texttt{AND}}^{\texttt{ast\_node}}$$

$$\textit{build\_binop}(\texttt{binop}(\texttt{"\&\&"})) \xrightarrow{\text{ast}} \overbrace{\texttt{BAND}}^{\texttt{ast\_node}}$$

$$\textit{build\_binop}(\texttt{binop}(\texttt{"||"})) \xrightarrow{\text{ast}} \overbrace{\texttt{BOR}}^{\texttt{ast\_node}}$$

$$\textit{build\_binop}(\texttt{binop}(\texttt{"<->"})) \xrightarrow{\text{ast}} \overbrace{\texttt{EQ\_OP}}^{\texttt{ast\_node}}$$

$$\textit{build\_binop}(\texttt{binop}(\texttt{"DIV"})) \xrightarrow{\text{ast}} \overbrace{\texttt{DIV}}^{\texttt{ast\_node}}$$

$$\textit{build\_binop}(\texttt{binop}(\texttt{"DIVRM"})) \xrightarrow{\text{ast}} \overbrace{\texttt{DIVRM}}^{\texttt{ast\_node}}$$

$$\textit{build\_binop}(\texttt{binop}(\texttt{"XOR"})) \xrightarrow{\text{ast}} \overbrace{\texttt{XOR}}^{\texttt{ast\_node}}$$

$$\textit{build\_binop}(\texttt{binop}(\texttt{"=="})) \xrightarrow{\text{ast}} \overbrace{\texttt{EQ\_OP}}^{\texttt{ast\_node}}$$

$$\textit{build\_binop}(\texttt{binop}(\texttt{"!="})) \xrightarrow{\text{ast}} \overbrace{\texttt{NEQ}}^{\texttt{ast\_node}}$$

$$\textit{build\_binop}(\texttt{binop}(\texttt{">"})) \xrightarrow{\text{ast}} \overbrace{\texttt{GT}}^{\texttt{ast\_node}}$$

$$\textit{build\_binop}(\texttt{binop}(\texttt{">="})) \xrightarrow{\text{ast}} \overbrace{\texttt{GEQ}}^{\texttt{ast\_node}}$$

$$\textit{build\_binop}(\texttt{binop}(\texttt{"-->"})) \xrightarrow{\text{ast}} \overbrace{\texttt{IMPL}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop("<")}) \xrightarrow{\text{ast}} \overbrace{\texttt{LT}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop("<=")}) \xrightarrow{\text{ast}} \overbrace{\texttt{LEQ}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop("+")}) \xrightarrow{\text{ast}} \overbrace{\texttt{PLUS}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop("-")}) \xrightarrow{\text{ast}} \overbrace{\texttt{MINUS}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop("MOD")}) \xrightarrow{\text{ast}} \overbrace{\texttt{MOD}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop("*")}) \xrightarrow{\text{ast}} \overbrace{\texttt{MUL}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop("OR")}) \xrightarrow{\text{ast}} \overbrace{\texttt{OR}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop("RDIV")}) \xrightarrow{\text{ast}} \overbrace{\texttt{RDIV}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop("<<")}) \xrightarrow{\text{ast}} \overbrace{\texttt{SHL}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop(">>")}) \xrightarrow{\text{ast}} \overbrace{\texttt{SHR}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop("^")}) \xrightarrow{\text{ast}} \overbrace{\texttt{POW}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop("++")}) \xrightarrow{\text{ast}} \overbrace{\texttt{CONCAT}}^{\texttt{ast\_node}}$$

8.52 SyntaxRule.StmtFromList

The function

$$\mathit{stmt\_from\_list}(\overbrace{\mathsf{stmt}^*}^{\mathtt{stmts}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\mathtt{new\_s}}$$

builds a statement `new_s` from a possibly-empty list of statements `stmts`.

EMPTY

$$\frac{}{\mathit{stmt\_from\_list}(\overbrace{[\,]}^{\mathtt{stmts}}) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{S\_Pass}}^{\mathtt{new\_s}}}$$

NON_EMPTY

$$\frac{\mathit{stmt\_from\_list}(\mathtt{stmts1}) \xrightarrow{\mathsf{ast}} \mathtt{s1} \qquad \mathit{sequence\_stmts}(\mathtt{s}, \mathtt{s1}) \xrightarrow{\mathsf{ast}} \mathtt{new\_s}}{\mathit{stmt\_from\_list}(\overbrace{[\mathtt{s}] + \mathtt{stmts1}}^{\mathtt{stmts}}) \xrightarrow{\mathsf{ast}} \mathtt{new\_s}}$$

8.53 SyntaxRule.SequenceStmts

The function

$$\mathit{sequence\_stmts}(\overbrace{\mathsf{stmt}}^{\mathtt{s1}}, \overbrace{\mathsf{stmt}}^{\mathtt{s2}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\mathtt{new\_s}}$$

Combines the statement `s1` with `s2` into the statement `new_s`, while filtering away instances of S_Pass.

S1_SPASS

$$\mathit{sequence\_stmts}(\overbrace{\mathsf{S\_Pass}}^{\mathtt{s1}}, \mathtt{s2}) \xrightarrow{\mathsf{ast}} \overbrace{\mathtt{s2}}^{\mathtt{new\_s}}$$

S2_SPASS

$$\frac{\mathtt{s1} \neq \mathsf{S\_Pass}}{\mathit{sequence\_stmts}(\mathtt{s1}, \overbrace{\mathsf{S\_Pass}}^{\mathtt{s2}}) \xrightarrow{\mathsf{ast}} \overbrace{\mathtt{s1}}^{\mathtt{new\_s}}}$$

NO_SPASS

$$\frac{\mathtt{s1} \neq \mathsf{S\_Pass} \qquad \mathtt{s2} \neq \mathsf{S\_Pass}}{\mathit{sequence\_stmts}(\mathtt{s1}, \mathtt{s2}) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{S\_Seq}(\mathtt{s1}, \mathtt{s2})}^{\mathtt{new\_s}}}$$

Chapter 9

Building Macro Productions

This chapter defines builder relations for the subset of macro productions in Section 6.2 that are not inlined:

- SyntaxRule.List (see Section 9.1)
- SyntaxRule.CList (see Section 9.2)
- SyntaxRule.NTCList (see Section 9.3)
- SyntaxRule.Option (see Section 9.4)

We also define SyntaxRule.Identity (see Section 9.5), which can be used in conjunction with the rules above in application to terminals.

9.1 SyntaxRule.List

The meta relation

$$\mathit{build\_list}[b](\overbrace{N}^{\texttt{syms}}) \bigtimes \overbrace{A}^{\texttt{sym\_asts}}$$

which is parameterized by an AST building relation $b : E \bigtimes A$, takes a parse node that represents a possibly-empty list of E values — `syms` — and returns the result of applying b to each of them — `sym_asts`.

EMPTY

$$\mathit{build\_list}[b](\overbrace{\epsilon}^{\texttt{syms}}) \xrightarrow{\text{ast}} \overbrace{[\,]}^{\texttt{sym\_asts}}$$

NON_EMPTY

$$\frac{b(\texttt{v}) \xrightarrow{\text{ast}} \texttt{v\_ast} \qquad \mathit{build\_list}[b](\texttt{syms1}) \xrightarrow{\text{ast}} \texttt{sym\_asts1}}{\mathit{build\_list}[b](\overbrace{\mathsf{list}^*(N)(\texttt{v} : E, \texttt{syms1} : \mathsf{list}^*(N))}^{\texttt{syms}}) \xrightarrow{\text{ast}} \overbrace{[\texttt{v\_ast}] + \texttt{sym\_asts1}}^{\texttt{sym\_asts}}}$$

9.2 SyntaxRule.CList

The meta relation

$$\mathit{build\_clist}[b](\overbrace{N}^{\texttt{syms}}) \bigtimes \overbrace{A}^{\texttt{sym\_asts}}$$

which is parameterized by an AST building relation $b : E \bigtimes A$, takes a parse node that represents a possibly-empty comma-separated list of E values — `syms` — and returns the result of applying b to each of them — `sym_asts`.

EMPTY

$$\mathit{build\_clist}[b](\overbrace{\epsilon}^{\texttt{syms}}) \xrightarrow{\text{ast}} \overbrace{[\,]}^{\texttt{sym\_asts}}$$

NON_EMPTY

$$\frac{b(\texttt{v}) \xrightarrow{\text{ast}} \texttt{v\_ast} \qquad \mathit{build\_clist}[b](\texttt{syms1}) \xrightarrow{\text{ast}} \texttt{sym\_asts1}}{\mathit{build\_clist}[b](\overbrace{\mathsf{clist}^*(N)(\texttt{v} : E, \texttt{","}, \texttt{syms1} : \mathsf{clist}^+(N))}^{\texttt{syms}}) \xrightarrow{\text{ast}} \overbrace{[\texttt{v\_ast}] + \texttt{sym\_asts1}}^{\texttt{sym\_asts}}}$$

9.3 SyntaxRule.NTCList

The meta relation

$$\mathit{build\_ntclist}[b](\overbrace{N}^{\texttt{syms}}) \bigtimes \overbrace{A}^{\texttt{sym\_asts}}$$

which is parameterized by an AST building relation $b : E \bigtimes A$, takes a parse node that represents a non-empty comma-separated trailing list of E values — `syms` — and returns the result of applying b to each of them — `sym_asts`.

EMPTY

$$\frac{b(\texttt{v}) \xrightarrow{\text{ast}} \texttt{v\_ast}}{\mathit{build\_ntclist}[b](\overbrace{\texttt{v}\ \mathsf{option}(\texttt{","})}^{\texttt{syms}}) \xrightarrow{\text{ast}} \overbrace{[\texttt{v\_ast}]}^{\texttt{sym\_asts}}}$$

NON_EMPTY

$$\frac{b(\texttt{v}) \xrightarrow{\text{ast}} \texttt{v\_ast} \qquad \mathit{build\_ntclist}[b](\texttt{syms1}) \xrightarrow{\text{ast}} \texttt{sym\_asts1}}{\mathit{build\_ntclist}[b](\overbrace{\texttt{v} : E, \texttt{","}, \texttt{syms1} : \mathsf{ntclist}(N)}^{\texttt{syms}}) \xrightarrow{\text{ast}} \overbrace{[\texttt{v\_ast}] + \texttt{sym\_asts1}}^{\texttt{sym\_asts}}}$$

9.4 SyntaxRule.Option

The meta relation

$$\mathit{build\_option}[b](\overbrace{N}^{\texttt{sym}}) \bigtimes \overbrace{\langle A \rangle}^{\texttt{sym\_ast}}$$

which is parameterized by an AST building relation $b : E \bigtimes A$, takes a parse node that represents an optional E value — `sym` — and returns the result of applying b to the value if it exists — `sym_asts`.

NONE

$$\mathit{build\_option}[b](\overbrace{\epsilon}^{\texttt{sym}}) \xrightarrow{\text{ast}} \overbrace{\texttt{None}}^{\texttt{sym\_ast}}$$

SOME

$$\frac{b(\texttt{v}) \xrightarrow{\text{ast}} \texttt{v\_ast}}{\mathit{build\_option}[b](\overbrace{\texttt{v} : E}^{\texttt{sym}}) \xrightarrow{\text{ast}} \overbrace{\langle \texttt{v\_ast} \rangle}^{\texttt{sym\_ast}}}$$

When this relation is applied to a sentence consisting of a prefix of terminals $t_{1..k}$, ending with a non-terminal `v`, it ignore the terminals and returns the result for the non-terminal.

LAST

$$\frac{\mathit{build\_option}[b](\texttt{v}) \xrightarrow{\text{ast}} \texttt{sym\_ast}}{\mathit{build\_option}[b](t_{1..k}, \texttt{v} : E) \xrightarrow{\text{ast}} \texttt{sym\_ast}}$$

9.5 SyntaxRule.Identity

The meta function

$$\mathit{build\_identity}(\overbrace{T}^{x}) \longrightarrow \overbrace{T}^{x}$$

is the identity function, which can be used as an argument to meta functions such as *build_list* when they are applied to terminals.

$$\mathit{build\_identity}(x) \xrightarrow{\text{ast}} x$$

Chapter 10

Correspondence Between Left-hand-side Expressions and Right-hand-side Expressions

The recursive function rexpr : lexpr → expr transforms left-hand-side expressions to corresponding right-hand-side expressions, which is utilized both for the type system and semantics:

Left hand side expression		**Right hand side expression**
rexpr(LE_Var(x))	=	E_Var(x)
rexpr(LE_Slice(le, args))	=	E_Slice(rexpr(le), args)
rexpr(LE_SetArray(le, e))	=	E_GetArray(rexpr(le), e)
rexpr(LE_SetField(le, x))	=	E_GetField(rexpr(le), x)
rexpr(LE_SetFields(le, x))	=	E_GetFields(rexpr(le), x)
rexpr(LE_Discard)	=	E_Var(-)
rexpr(LE_Destructuring([$\texttt{le}_{1..k}$]))	=	E_Tuple([$i = 1..k$: rexpr($\texttt{le}_i$)])
rexpr(LE_Concat([$\texttt{le}_{1..k}$], _))	=	E_Concat([$i = 1..k$: rexpr($\texttt{le}_i$)])