MONASH University

Guan Ting Ho (Student ID: 35037687)
Friday, 24 October 2025

# FIT2102 Programming Paradigms – S2 2025

Report for Assignment 2: **BNF to Haskell Parser Generator**

## Design of the Code

The project is built around a single, comprehensive Abstract Data Type (ADT) that acts as the shared representation across all stages. Defined in *Assignment.hs* (Rule, Definition, Alternative, Term), it captures every grammatical construct from the specification, including modifiers and parameterised rules.

All core logic resides in *Assignment.hs*, which functions as a pure library. Its structure has four logical parts:

1. **ADT definitions** – shared data types.
2. **Parsing** – bottom-up combinator parsing that composes primitives (`pTerminal, pMacro`) into bnfParser.
3. **Haskell code generation** – traverses the ADT to render Haskell code (`renderDataDecl, renderParserDecl`).
4. **Validation** – pure analyses for correctness (`duplicateWarnings, leftRecursiveRuleNames`).

The backend (*Main.hs*) and frontend (*main.ts*) are orchestration layers handling I/O and the GHC API. This separation keeps the parsing and transformation logic pure, declarative, and easily testable.

Before code generation, grammars are filtered to ensure safety. `keepFirstRules` removes duplicates, and `filterInvalidRules` iteratively drops undefined or recursive rules so downstream stages never see malformed definitions. This consistent behaviour simplifies both testing and user interaction.

## Parsing Logic

The parsing strategy mirrors the BNF structure: "one or more rules, each with alternatives, each made of terms." The top-level bnfParser composes smaller parsers (`pRule, pDefinition, pAlternative, pTerm`), producing an AST that structurally matches the input grammar.

Everything is built using the provided Parser type, which supports Functor, Applicative, and Monad. This lets each parsing rule be a pure expression rather than manual recursion or state mutation.

**Choice and precedence – <|>**
We use <|> to express alternatives, directly reflecting the | symbol in BNF.

Example: pTerm = pTok <|> pBareTerm means "try tokenised term or bare term." Ordering matters: pParamUsage is placed before pNonTerminal to resolve ambiguity, since the parameterised form is more specific.

**Context-sensitive parsing – Monad**

Monad appears only where a parse depends on previous results. For example, parameter references [a] must exist in the rule's parameter list. We implemented a `ParamScope` that `pRule` constructs and passes down; `pParamRef` then checks membership and fails with "Unknown parameter" if not found. This enforces scope validation during parsing, as required.

Small helpers like `commaSeparated,` `pModifier`, and `usageArgs` capture recurring patterns, keeping the grammar modular and extensible. Adding new macros or modifiers becomes a localised change rather than a rewrite.

## Validation

Validation follows a two-stage model: **Report** and **Sanitise**.

1. **Report (validate)** runs on the raw ADT and produces warnings for:
   - *Duplicate rules* – by counting rule-name occurrences.
   - *Undefined non-terminals* – by comparing all references (`TNonTerminal,` `TParamUsage`) to the defined rule names.
   - *Left recursion* – detected through a dependency graph of left-most non-terminals and depth-first search for cycles.
2. **Sanitise (filterInvalidRules)** ensures generated code is safe. It first keeps only the first instance of each rule, then repeatedly removes rules that are undefined, recursive, or now indirectly invalid until the grammar stabilises.

This design means users see all their original errors, while the compiler only works on a guaranteed-safe subset. It balances feedback clarity with compile-time safety.

## Haskell and Functional Programming Techniques

The design relies heavily on Haskell's type system to maintain consistency. The grammar's recursive structure maps directly onto algebraic data types, and code generation simply walks that structure. Functions like `termType` and `termParser` translate grammar terms into Haskell types and parser expressions, ensuring the generated code stays faithful to the source grammar.

Functor, Applicative, and Monad make the parser composable: <*> and <|> handle sequencing and alternatives, while >>= supports the few context-dependent cases. This mix keeps the parser both readable and expressive.

We favoured small, single-purpose functions connected through higher-order tools like `map,` `foldl,` and `concatMap`, forming clear, pure data pipelines. For instance, `definitionReferences = nub . concatMap alternativeReferences` concisely expresses "collect all references and remove duplicates." Point-free style appears only where it clarifies the data flow, not for brevity's sake.

The advantage of this functional structure is predictability. Every stage is a pure transformation ([Rule] → [Rule], [Rule] → [String], etc.), so adding a new validation pass or generator involves no side effects and minimal risk. The immutable ADT makes the system robust and easy to reason about.

## Extensions

**Whitespace macro [space]**

The original specification had no clean way to parse a literal space, leading to hacks like misusing `[alpha]`. We introduced `MSpace` into `MacroType` and updated only three functions: `pMacro` (to recognise "space"), `termType` (to map it to Char), and `termParser` (to emit (is ' ')). This small, local change immediately improved grammar readability.

**Parameter-arity validation**

Parameterised rules were easy to misuse by passing the wrong number of arguments. We added a new validation stage that checks each call's argument count against the rule's declared arity. Implementation steps:

1. Map each rule name to its expected arity.
2. Collect all `TParamUsage` calls.
3. Compare actual versus expected counts.
4. Feed mismatches into both validate (for UI warnings) and `filterInvalidRules` (to block generation).

Because everything already shared a single ADT, this feature slotted in cleanly without changing any parsing or generation logic.

## Conclusion

The system's design is intentionally declarative: parse once into a rich AST, analyse and filter it through pure passes, then generate Haskell code that mirrors the grammar. Parser combinators let the BNF be described naturally and safely in Haskell. The result is a maintainable, testable, and extendable grammar-to-code pipeline that clearly demonstrates the strengths of the functional paradigm.