One Parser to Rule Them All

Ali Afroozeh Anastasia Izmaylova
Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
{ali.afroozeh, anastasia.izmaylova}@cwi.nl

Abstract

Despite the long history of research in parsing, constructing parsers for real programming languages remains a difficult and painful task. In the last decades, different parser generators emerged to allow the construction of parsers from a BNF-like specification. However, still today, many parsers are handwritten, or are only partly generated, and include various hacks to deal with different peculiarities in programming languages. The main problem is that current declarative syntax definition techniques are based on pure context-free grammars, while many constructs found in programming languages require context information.

In this paper we propose a parsing framework that embraces context information in its core. Our framework is based on data-dependent grammars, which extend context-free grammars with arbitrary computation, variable binding and constraints. We present an implementation of our framework on top of the Generalized LL (GLL) parsing algorithm, and show how common idioms in syntax of programming languages such as (1) lexical disambiguation filters, (2) operator precedence, (3) indentation-sensitive rules, and (4) conditional preprocessor directives can be mapped to data-dependent grammars. We demonstrate the initial experience with our framework, by parsing more than 20 000 Java, C#, Haskell, and OCaml source files.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Syntax; D.3.4 [*Programming Languages*]: Processors—Parsing

Keywords Parsing, data-dependent grammars, GLL, disambiguation, operator precedence, offside rule, preprocessor directives, scannerless parsing, context-aware scanning

1. Introduction

Parsing is a well-researched topic in computer science, and it is common to hear from fellow researchers in the field of programming languages that parsing is a *solved* problem. This statement mostly originates from the success of Yacc [18] and its underlying theory that has been developed in the 70s. Since Knuth's seminal paper on LR parsing [25], and DeRemer's work on practical LR parsing (LALR) [6], there is a linear parsing technique that covers most syntactic constructs in programming languages. Yacc, and its various ports to other languages, enabled the generation of efficient parsers from a BNF specification. Still, research papers and tools on parsing in the last four decades show an ongoing effort to develop new parsing techniques.

A central goal in research in parsing has been to enable language engineers (i.e., language designers and tool builders) to *declaratively* build a parser for a real programming language from an (E)BNF-like specification. Nevertheless, still today, many parsers are hand-written or are only partially generated and include many hacks to deal with peculiarities in programming languages. The reason is that grammars of programming languages in their simple and readable form are often not deterministic and also often ambiguous. Moreover, many constructs found in programming languages are not context-free, e.g., indentation rules in Haskell. Parser generators based on pure context-free grammars cannot natively deal with such constructs, and require ad-hoc extensions or hacks in the lexer. Therefore, additional means are necessary outside of the power of context-free grammars to address these issues.

General parsing algorithms [7, 33, 35] support all context-free grammars, therefore the language engineer is not limited by a specific deterministic class, and there are known declarative disambiguation constructs to address the problem of ambiguity in general parsing [12, 36, 38]. However, implementing disambiguation constructs is notoriously hard and requires thorough knowledge of the underlying parsing technology. This means that it is costly to declaratively build a parser for a given programming language in the wild if the required disambiguation constructs are not already supported. Perhaps surprisingly, examples of such languages are not only the legacy languages, but also modern languages such as Haskell, Python, OCaml and C#.

In this paper we propose a parsing framework that is able to deal with many challenges in parsing existing and new programming languages. We embrace the need for context information at runtime, and base our framework on data-dependent grammars [16]. Data-dependent grammars are an extension of context-free grammars that allow arbitrary computation, variable binding and constraints. These features allow us to simulate hand-written parsers and to implement disambiguation constructs.

To demonstrate the concept of data-dependent grammars we use the IMAP protocol [29]. In network protocol messages it is common to send the length of data before the actual data. In IMAP, these messages are called literals, and are described by the following (simplified) context-free rule:

```
L8 ::= '~{' Number '}' Octets
```

Here `Octets` recognizes a list of octet (any 8-bit) values. An example of `L8` is `~{6}aaaaaa`. As can be seen, there is no data dependency in this context-free grammar, but the IMAP specification says that the number of `Octets` is determined by the value parsed by `Number`. Using data-dependent grammars, we can specify such a data-dependency as:

```
L8 ::= '~{' nm:Number {n=toInt(nm.yield)} '}' Octets(n)
Octets(n) ::= [n > 0] Octets(n - 1) Octet
            | [n == 0] ϵ
```

In the data-dependent version, `nm` provides access to the value parsed by `Number`. We retrieve the substring of the input parsed by `Number` via `nm.yield` which is converted to integer using `toInt`. This integer value is bound to variable `n`, and is passed to `Octets`. `Octets` takes an argument that specifies the number of iterations. Conditions `[n > 0]` and `[n == 0]` specify which alternative is selected at each iteration.

It is possible to parse IMAP using a general parser, and then remove the derivations that violate data dependencies post parse. However, such an approach would be slow. Without enforcing the dependency on the length of `Octets` during parsing, given the nondeterministic nature of general parsing, all possible lengths of `Octets` will be tried.

There are many common grammar and disambiguation idioms that can be desugared into data-dependent grammars. Examples of these idioms are operator precedence, longest match, and the offside rule. Expecting the language engineer to write low-level data-dependent grammars for such cases would be wasteful. Instead, we describe a number of such idioms, provide high-level notation for them and their desugaring to data-dependent grammars. For example, using our high-level notation the indentation rules in Haskell can be expressed as follows:

```
Decls ::= align (offside Decl)*
        | ignore('{' Decl (';' Decl)* '}')
```

This definition clearly and concisely specifies that either all declarations in the list are aligned, and each `Decl` is offsided with regard to its first token (first alternative), or indentation is ignored inside curly braces (second alternative).

Our vision is a parsing framework that provides the right level of abstraction for both the language engineer, who designs new languages, and the tool builder, who needs a parsing technology as part of her toolset. From the language engineer's perspective, our parsing framework provides an out of the box set of high-level constructs for most common idioms in syntax of programming languages. The language engineer can also always express her needs directly using data-dependent grammars. From the tool builder's perspective, our framework provides open, extensible means to define higher-level syntactic notation, without requiring knowledge of the internal workings of a parsing technology.

The contributions of this paper are:

- We provide a unified perspective on many important challenges in parsing real programming languages.
- We present several high-level syntactical constructs, and their mappings to data-dependent grammars.
- We provide an implementation of data-dependent grammars on top of Generalized LL (GLL) parsing [33] that runs over ATN grammars [40]. The implementation is part of the Iguana parsing framework[1] [2].
- We demonstrate the initial results of our parsing framework, by parsing 20363 real source files of Java, C#, Haskell (91% success rate), and excerpts from OCaml.

The rest of this paper is organized as follows. In Section 2 we describe the landscape of parsing programming languages. In Section 3 we present data-dependent grammars, our high-level syntactic notation, and the mapping to data-dependent grammars. Section 4 discusses the extension of GLL parsing with data-dependency. In Section 5 we demonstrate the initial results of our parsing framework using grammars of real programming languages. We discuss related work in Section 6. A conclusion and discussion of future work is given in Section 7.

2. The Landscape of Parsing Programming Languages

In this section we discuss well-known features of programming languages that make them hard to parse. These features motivate our design decisions.

2.1 General Parsing for Programming Languages

Grammars of programming languages in their natural form are not deterministic, and are often ambiguous. A well-known example is the `if-then-else` construct found in many programming languages. This construct, when written in its natural form, is ambiguous (the dangling-else ambiguity), and therefore cannot be deterministic. Some nondeterministic (and ambiguous) syntactic constructs, such as `if-then-else`, can be rewritten to be deterministic and un-

[1] `https://github.com/iguana-parser`

ambiguous. However, such grammar rewriting in general is not trivial, the resulting grammar is hard to read, maintain and evolve, and the resulting parse trees are different from the original ones the grammar writer had in mind.

Instead of rewriting a grammar, it is common to use an ambiguous grammar, and rely on some implicit behavior of a parsing technology for disambiguation. For example, the dangling-else ambiguity is often resolved using a longest match scheme provided by the underlying parsing technology. Relying on implicit behavior of a parsing technology to achieve determinism can make it quite difficult to reason about the accepted language. Seemingly correct sentences may be rejected by the parser because at a nondeterministic point, a wrong path was chosen. For example, Yacc is an LALR parser generator, but can accept any context-free grammar by automatically resolving *all* shift/reduce and reduce/reduce conflicts. Using Yacc, the language engineer should manually check the resolved conflicts in case of unexpected behavior.

A common theme in research in parsing has been to increase the recognition power of deterministic parsing techniques such as LL(k) or LR(k). One of the widely used general parsing techniques for programming languages is the Generalized LR (GLR) algorithm [35]. GLR parsers support all context-free grammars and can produce a parse forest containing all derivation trees in form of a Shared Packed Parse Forest (SPPF) in cubic time and space [34]. Note that the cubic bound is for the worst-case, highly ambiguous grammars. As GLR is a generalization of LR, a GLR parser runs linearly on LR parts of the grammar, and as the grammars of real programming languages are in most parts near deterministic, one can expect near-linear performance using GLR for parsing programming languages. GLR parsing has successfully been used in source code analysis and developing domain-specific languages [12, 22].

General parsing enables the language engineer to use the most natural version of a grammar, but leaves open the problem of ambiguity. In declarative syntax definition [12, 23], it is common to use declarative disambiguation constructs, e.g., for operator precedence or the longest match. As a general parser is able to return all ambiguities in form of a parse forest, it is possible to apply the disambiguation rules post-parse, removing the undesired derivations from the parse forest. However, such post-parse disambiguation is not practical in cases where the grammar is highly ambiguous. For example, parsing expression grammars without applying operator precedence during parsing is only limited to small inputs. Therefore, it is required to resolve ambiguity while parsing to achieve near-linear performance.

Implementing disambiguation mechanisms that are executed during parsing is difficult. This is because the implementation of such disambiguation mechanisms requires knowledge of the internal workings of a parsing technology. Therefore, the choice of the general parsing technology becomes very important when considering parsing programming languages. For example, GLR parsers operate on LR automata, and have a rather complicated execution model, as a parsing state corresponds to multiple grammar positions.

The Generalized LL (GLL) parsing algorithm [33] is a new generalization of recursive-descent parsing that supports all context-free grammars, including left recursive ones. GLL parsers produce a parse forest in cubic time and space in the worst case, and are linear on LL parts of the grammar. GLL parsers are attractive because they have the close relationship with the grammar that recursive-descent parsers have. From the end user's perspective, GLL parsers can produce better error messages, and can be debugged in a programming language IDE.

To deal with left recursive rules and to keep the cubic bound, a GLL parser uses a GSS to handle multiple call stacks. While the execution model of a GLL parser is close to recursive-descent parsing, the underlying machinery is much more complicated, and still an in-depth knowledge of GLL is required to implement disambiguation constructs. In this paper, we propose a parser-independent framework for parsing programming languages based on data-dependent grammars. We use GLL parsing as the basis for our data-dependent parsing framework, as it allows an intuitive way to implement components of data-dependent grammars, such as environment threading, and enables an implementation that is very close to the stack-evaluation based semantics of data-dependent grammars [16].

2.2 On the Interaction between Lexer and Parser

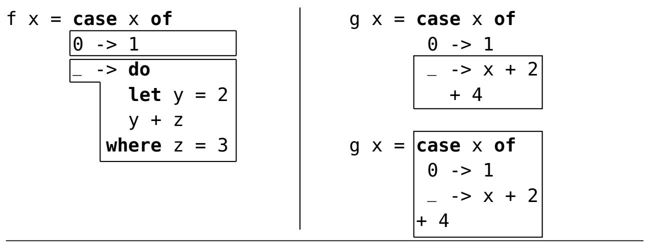
Conventional parsing techniques use a separate lexing phase before parsing to transform a stream of characters to a stream of tokens. In particular, whitespace and comments are discarded by the lexer to reduce the number of lookahead in the parsing phase, and enable deterministic parsing.

The main problem with a separate lexing phase is that without having access to the parsing context, i.e., the applicable grammar rules, the lexer cannot unambiguously determine the type of some tokens. An example is `>>` that can either be parsed as a right shift operator, or two closing angle brackets of a generic type, e.g., `List<List<String>>` in Java. Some handwritten parsers deal with this issue by rewriting the token stream. For example, when the `javac` parser reads a `>>` token and is in a parsing state that expects only one `>`, e.g., when matching the closing angle bracket of a generic type, it only consumes the first `>` and puts the second one back to prevent a parse error when matching the next angle bracket.

To resolve the problems of a separate lexing phase, we need to expose the parsing context to the lexer. To achieve this, the separate lexing phase is abandoned, and the lexing phase is effectively integrated into the parsing phase. We call this model *single-phase* parsing. There are two options to achieve *single-phase parsing*. The first option is called *scannerless* parsing [32, 38] where lexical definitions are treated as context-free rules. In scannerless parsing, gram-

```
E ::= '-' E        | E ::= E '+' T   | E  ::= T E1
    | E '*' E      |     | T         | E1 ::= '+' T E1 | ϵ
    | E '+' E      | T ::= T '*' F   | T  ::= F T1
    | 'a'          |     | F         | T1 ::= '*' F | ϵ
                   | F ::= '-' F     | F  ::= '-' F
                   |     | 'a'       |      | 'a'
```

Figure 1. Three grammars that accept the same language: the natural, ambiguous grammar (left), the grammar with precedence encoding (middle), and the grammar after left-recursion removal (right).



```
f x = case x of
        0 -> 1
        _ -> do
          let y = 2
          y + z
        where z = 3
```

```
g x = case x of
        0 -> 1
        _ -> x + 2
          + 4

g x = case x of
        0 -> 1
        _ -> x + 2
      + 4
```

Figure 2. Examples of indentation rules in Haskell.

mars are defined to the level of characters. The second option is *context-aware* scanning [37], where the parser calls the lexer on demand. At each parsing state, the lexer is called with the expected set of terminals at that state.

In almost all modern programming languages longest match (maximal munch) is applied, and keywords are excluded from being recognized as identifiers. These disambiguation rules are conventionally embedded in the lexer. In *single-phase parsing*—scannerless or context-aware—longest match and keyword exclusion have to be applied during parsing, by using lexical disambiguation filters such as follow restrictions [32, 36]. These disambiguation filters have parser-specific implementations [2, 38]. In Section 3 we show how these filters can be mapped to data-dependent grammars. Also note that although a context-aware scanner employs longest match, for example by implementing the Kleene star ($*$) as a greedy operator, in some cases we still need to use explicit disambiguation filters, see Section 3.2.

2.3 Operator Precedence

Expressions are an integral part of virtually every programming language. In reference manuals of programming languages it is common to specify the semantics of expressions using the priority and associativity of operators. However, the implementation of expression grammars can considerably deviate from such precedence specification.

It is possible to encode operator precedence by rewriting the grammar: a new nonterminal is created for each precedence level. The rewriting is not trivial for real programming languages, and the resulting grammar becomes large. This rewriting is particularly problematic in parsing techniques that do not support left recursion. The left-recursion removal transformation disfigures the grammar and adds extra complexity in transforming the trees to the intended ones. Figure 1 shows three versions of the same expression grammar.

In the 70s, Aho *et al.* [4] presented a technique in which a parser is constructed from an ambiguous expression grammar accompanied with a set of precedence rules. This work can be seen as the starting point for declarative disambiguation using operator precedence rules. Aho *et al.*'s approach is implemented by modifying LALR parse tables to resolve shift/reduce conflicts based on the operator precedence. However, the semantics of operator precedence in this approach is bound to the internal workings of LR parsing. There have been other solutions to build parsers from declarative operator precedence which we discuss in Section 6. In Section 3.4 we provide a mapping from operator precedence rules to data-dependent grammars.

2.4 Offside Rule

In most programming languages, indentation of code blocks does not play a role in the syntactic structure. Rather, explicit delimiters, such as `begin` and `end` or `{` and `}` are used to specify blocks of statements. Landin introduced the *offside* rule [26], which serves as a basis for indentation-sensitive languages. The offside rule says that all the tokens of an expression should be indented to the right of the first token. Haskell and Python are two examples of popular programming languages that use a variation of the offside rule.

Figure 2 shows two examples of the offside rule in Haskell. The keywords `do`, `let`, `of`, and `where` signal the start of a block where the starting tokens of the statements should be aligned, and each statement should be offsided with regard to its first token. In Figure 2 (left), `case` has two alternatives which are aligned, and the second alternative that spans several lines is offsided with regard to its first token, i.e., `_`. Figure 2 (right) shows two examples that look the same, but the indentation of the last part, `+ 4`, is different. In the top declaration `+ 4` belongs to the last alternative, but in the bottom declaration, `+ 4` belongs to the expression on the right hand side of `=`.

Indentation sensitivity in programming languages cannot be expressed by pure context-free grammars, and has often been implemented by hacks in the lexer. For example, in Haskell and Python, indentation is dealt with in the lexing phase, and the context-free part is written as if no indentation-sensitivity exists. Both GHC and CPython, the popular implementations of Haskell and Python, use LALR parser generators. In Python, the lexer maintains a stack and emits `INDENT` and `DEDENT` tokens when indentation changes. In Haskell, the lexer translates indentation information into curly braces and semicolons based on the rules specified by the L function [28].

In Section 3.5 we show how data-dependent grammars can be used for single-phase parsing of indentation-sensitive programming languages in a declarative way. As data-dependent grammars are rather low-level for such solutions, we introduce three high-level constructs: **`align`**, **`offside`**, and **`ignore`** which are desugared to data-dependent grammars.

```
void test()
{
#if Debug
  System.Console.WriteLine("Debug")
}
#else
}
#endif
```

```
#if X
/*
#else
/* */ class Q { }
#endif
```

Figure 3. Problematic cases of using C# directives [30].

2.5 Conditional Directives

Many programming languages allow compiler *pragmas* that specify how the compiler (or the interpreter) processes parts of the input. The C family of programming languages, i.e., C, C++ and C#, allow preprocessor directives such as `#if` and `#define`. GHC also allows various compiler pragmas [28, §12.3]. For example, it is possible to enable C preprocessor directives in Haskell using `{-# LANGUAGE CPP #-}`.

Preprocessor directives pose considerable difficulty in parsing programming languages. The main reason is that they are not part of the grammar of a language, but can appear anywhere in the source code. In this regard, preprocessor directives are similar to whitespace and comment. However, conditional directives may affect the syntactic structure of a program, and cannot be simply ignored as a special kind of whitespace. This is especially important if we consider single-phase parsing where no lexing/preprocessing is available. We need a mechanism to allow the parser to switch between the preprocessor mode and main grammar, and to evaluate conditional directives to select the right branch.

Figure 3 (left) shows a C# example where ignoring the conditional directive will lead to a parse error, as the closing bracket of the `test` method is in the conditional directive, and one of the branches should be included in the input. The example in Figure 3 (right) shows another aspect of directives in C#. If `X` is true, `"/* #else /* */ class Q {}"` will be considered as part of the source code. If `X` is false, only the else-part will be considered: `"/* */ class Q {}"`. Note that when `X` is false, the if-part does not have to be syntactically correct, in this case an unclosed multi-line comment.

Among the family of C languages we selected C#, as parsing C# is more manageable. The problematic part of parsing C with directives is textual macros. Without a preprocessor to expand macros before parsing, we need to deal with macros at runtime. Parsing C without a preprocessor is future work. In C#, `#define` does not define a macro, rather it only sets a boolean variable. It should also be noted that C# supports multi-line strings, where directives should not be processed. Figure 4 shows a C# example that uses conditional directives in a multi-line string. In single-phase parsing, however, multi-line strings are not a problem, as we effectively parse each terminal in the context where it appears.

```
   static void Main() {
      System.Console.WriteLine(@"hello,
#if Debug
      world
#else
      Nebraska
#endif
      ");
   }
```

Figure 4. C# multi-line string containing directives [30].

2.6 Miscellaneous Features

There are many other peculiarities in programming languages and data formats that cannot be expressed by context-free grammars. There has been considerable effort to build declarative parsers for data formats, e.g. PADS [9], and one of the main motivations for data-dependent grammars [16] is indeed to enable parsing data formats.

Examples of languages that require data-dependent parsing are data protocols, such as IMAP and HTTP, and tag-based languages such as XML. In programming languages, data-dependent grammars can be used to implement some language-specific disambiguation mechanisms. For example, to maintain a table of type definitions in C to allow resolving the infamous typedef ambiguity, e.g., in `x * y` which can be either interpreted as a variable of pointer type `x` or as a multiplication, depending on the type of `x`. We give an example of parsing XML and resolving the typedef ambiguity in C in Section 3.7.

3. Parsing Programming Languages with Data-dependent Grammars

In this section we describe data-dependent grammars [16], discuss our single-phase parsing strategy, and demonstrate how various high-level, declarative syntax definition constructs can be desugared into data-dependent grammars.

3.1 Data-dependent Grammars

Data-dependent grammars are defined as an extension of *context-free grammars* (CFGs), where a CFG is, as usual, a tuple (N, T, P, S) where

- N is a finite set of nonterminals;
- T is a finite set of terminals;
- P is a finite set of rules. A rule (production) is written as $A ::= \alpha$, where A (head) is a nonterminal, and α (body) is a string in $(N \cup T)^*$;
- $S \in N$ is the start symbol of the grammar.

We use A, B, C to range over nonterminals, and a, b, c to range over terminals. We use α, β, γ for a possibly empty sequence of terminals and nonterminals, and ϵ represents the empty sequence. It is common to group rules with the same head and write them as $A ::= \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$. In this representation, each α_i is an alternative of A.

Data-dependent grammars introduce parametrized nonterminals, arbitrary computation via an expression language, constraints, and variable binding. Here, we assume that the expression language e is a simple functional programming language with immutable values and no side-effects. In a data-dependent grammar a rule is of the from $A(p) ::= \alpha$, where p is a formal parameter of A. Here, for simplicity of presentation and without loss of generality, we assume that a nonterminal can have at most one parameter. The body of a rule, α, can now contain the following additional symbols:

- $x = l : A(e)$ is a labeled call to A with argument e, label l, and variable x bound to the value returned by $A(e)$;
- $l : a$ is a labeled terminal a with label l;
- $[e]$ is a constraint;
- $\{x = e\}$ is a variable binding;
- $\{e\}$ is a return expression (only as the last symbol in α);
- $e \,?\, \alpha : \beta$ is a conditional selection.

The symbols above are presented in their general forms. For example, labels, variables to hold return values, and return expressions are optional.

Our data-dependent grammars are very similar to the ones introduced in [16] with four additions. First, terminals and nonterminals can be labeled, and labels refer to properties associated with the result of parsing a terminal or nonterminal. These properties are the start input index (or left extent), the end input index (or right extent), and the parsed substring. Properties can be accessed using dot-notation, e.g., for labeled nonterminal $b : B$, $b.l$ gives the left extent, $b.r$ the right extent, and $b.yield$ the substring.

Second, nonterminals can return arbitrary values (return expressions) which can be bound to variables. In several cases, we found this feature very practical as we could express data dependency without changing the shape of the original specification grammar. Specifically, cases where a global table needs to be maintained along a parse (C# conditional directives discussed in Section 3.6 and C typedef declarations in Section 3.7), or where semantic information needs to be propagated upwards from a complicated syntactic structure (`Declarator` of the C grammar in Section 3.7). In some cases a data-dependent grammar that uses return values can be rewritten to one without return values. However, in general, whether return values enlarge the class of languages expressible with the original data-dependent grammars is an open question for future work.

Third, we support regular expression operators (EBNF constructs): $*$, $+$, and $?$, by desugaring them to data dependent rules as follows: $A* ::= A+ \mid \epsilon$; $A+ ::= A+A \mid A$; and $A? ::= A \mid \epsilon$. In the data-dependent setting, this translation must also account for variable binding. For example, if symbol $([e]\ A)*$ appears in a rule, and x is a free variable in e, captured from the scope of the rule, our translation lifts this variable, introducing a parameter x to the new nonterminal. In addition, EBNF constructs introduce new scopes: variables declared inside an EBNF construct, e.g., $(l : A\ [e])*$, are not visible outside, e.g., in the rule that uses it.

Finally, we also introduce a conditional selection symbol $e \,?\, \alpha : \beta$, which selects α if e is evaluated to true, otherwise β, i.e., introduces deterministic choice. Similar to EBNF constructs, we implement conditional selection by desugaring it into a data-dependent grammar. For example, $A ::= \alpha\ e \,?\, X : Y \beta$ is translated to $A ::= \alpha\ C(e)\ \beta$, where $C(b) ::= [b]\ X \mid [!b]\ Y$. We illustrate use of the conditional selection when discussing C# directives in Section 3.6.

3.2 Single-phase Parsing Strategy

We implement our data-dependent grammars on top of the generalized LL (GLL) parsing algorithm [33]. As general parsers can deal with any context-free grammar, lexical definitions can be specified to the level of characters. For example, comment in the C# specification [30] is defined as:

```
Comment ::= SingleLineComment | DelimitedComment
SingleLineComment ::= "//" InputCharacter*
InputCharacter ::= ![\r \n]
DelimitedComment ::= "/*" DelimitedCommentSect* [*]+ "/"
DelimitedCommentSect ::= "/" [*]*  NotSlashOrAsterisk
NotSlashOrAsterisk = ![/ *]
```

Such character-level grammars, however, lead to very large parse forests. These parse forests reflect the full structure of lexical definitions, which are not needed in most cases. We provide the option to use an on-demand context-aware scanner, where terminals are defined using regular expression. For example, `Comment` in C# can be compiled to a regular expression. In cases where the structure is needed, or it is not possible to use a regular expression, e.g., recursive definitions of nested comments, the user can use character-level grammars.

Our support for context-aware scanning borrows many ideas from the original work by Van Wyk and Schwerdfeger [37], but because of the top-down nature of GLL parsing, there are some differences. The original context-aware scanning approach [37] is based on LR parsing, and as each LR state corresponds to multiple grammar rules, there may be several terminals that are valid at a state. The set of valid terminals in a parsing state is called *valid lookahead* set [37]. In GLL parsing, in contrast, the parser is at a single grammar position at each time, i.e., either before a nonterminal or before a terminal in a single grammar rule. Therefore, in GLL parsing, the valid lookahead set of a terminal grammar position contains only one element, which allows us to directly call the matcher of the regular expression of that terminal.

We use our simple context-aware scanning model for better performance, see Section 5.1. The implementation of the context-aware scanner in [37] is more sophisticated. The scanner is composed of all terminal definitions, as a composite DFA. This enables a longest match scheme across terminals in the same context, for example in programming languages where one terminal is a prefix of another, e.g.,

`A ::=` α `B !>> c` β	`A ::=` α `b:B [input.at(b.r) != c]` β
`A ::=` α `c !<< B` β	`A ::=` α `b:B [input.at(b.l-1) != c]` β
`A ::=` α `B \ s` β	`A ::=` α `b:B [input.sub(b.l,b.r)!= s]` β

Figure 5. Mapping of lexical disambiguation filters.

`'fun'` and `'function'` in OCaml. To enforce longest match across terminals we use follow/precede restrictions, in this case a follow restriction on `'fun'` or a precede restriction on identifiers. Moreover, keyword reservation in [37] is done by giving priority to keywords at matching states of the composite DFA. In our model, keyword exclusion should be explicitly applied in the grammar rules using an exclude disambiguation filter. We explain follow/precede restrictions and keyword reservation in Section 3.3.

In single-phase parsing layout (whitespace and comments) are treated the same way as other lexical definitions. Because layout is almost always needed in parsing programming languages, we support automatic layout insertion into the rules. There are two approaches to deal with layout insertion: a layout nonterminal can be inserted exactly before or after each terminal node [20, 37]. Another way is to insert layout between the symbols in a rule, like in SDF [12]. We use SDF-style layout insertion: if $X ::= x_1 x_2 \ldots x_n$ is a rule, and L is a nonterminal defining layout, after the layout insertion, the rule becomes $X ::= x_1 L\, x_2\, L \ldots L\, x_n$. A benefit of SDF-style layout insertion is that no symbol definition accidentally ends or starts with layout, provided that the layout is defined greedily (see Section 3.3). This is helpful when defining the offside rule (see Section 3.5).

3.3 Lexical Disambiguation Filters

Common lexical disambiguation filters [36], such as follow restrictions, precede restrictions and keyword exclusion, can be mapped to data-dependent grammars without further extensions to the parser generator or parsing algorithm. These disambiguation filters are common in scannerless parsing [32] and have been implemented for various generalized parsers [2, 38].

A follow restriction (`!>>`) specifies which characters cannot immediately appear after a symbol in a rule. This restriction is used to locally define longest match (as opposed to a global longest match in the lexer). For example, to enforce longest match on identifiers we write `Id ::= [A-Za-z]+ !>> [A-Za-z]`. A precede restriction (`!<<`) is similar to a follow restriction, but specifies the characters that cannot immediately precede a symbol in a rule. Precede restrictions can be used to implement longest match on keywords. For example, `[A-Za-z] !<< Id` disallows an identifier to start immediately after a character. This disallows, for example, to recognize `intx` as the keyword `'int'` followed by the identifier `x`. Finally, exclusion (`\`) is usually used to implement keyword reservation. For example, `Id \'int'` excludes the keyword `int` from beging recognized as `Id`.

Figure 5 shows the mapping from character-level disambiguation filters to data-dependent grammars. The mapping is straightforward: each restriction is translated into a condition that operates on the input. A note should be made regarding the condition implementing precede restrictions. This condition only depends on the left extent, `b.l`, that permits its application before parsing `B`. We consider this optimization in the implementation of our parsing framework, permitting application of such conditions before parsing labeled nonterminals or terminals.

The restrictions of Figure 5 are just examples and can be extended in many ways. For example, instead of defining the restriction using a single character, we can use regular expressions or character classes. One can also define similar restrictions for related disambiguation purposes. For example, consider the cast expression in C#:

```
cast-exp ::= '(' type ')' unary-exp
```

An expression such as `(x)-y` is ambiguous, and can be interpreted as either a type cast of `-y` to the type `x`, or a subtraction of `y` from `(x)`. In the C# language specification, it is stated that this ambiguity is resolved during parsing based on the character that comes after the closing parentheses: if the character following the closing parentheses is ~, `'!'`, `'('`, an identifier, a literal or keywords, the expression should be interpreted as a cast. We can implement this rule as follows:

```
cast-exp ::= '(' type ')' >>> [~!(A-Za-z0-9] unary-exp
```

The `>>>` notation specifies that the next character after the closing parentheses should be an element of the specified character class. The implementation of `>>>` is similar to that of `>>` with an additional aspect: it adds the condition on the automatically inserted layout nonterminal after `')'` instead.

These examples show how more syntactic sugar can be added to the existing framework for various common lexical disambiguation tasks in programming languages without changes to the underlying parsing technology.

3.4 Operator Precedence and Associativity

Expression grammars in their natural form are often ambiguous. Consider the expression grammar in Figure 6 (left). For this grammar, the input string `a+a*a` is ambiguous with two derivation trees that correspond to the following groupings: `(a+(a*a))` and `((a+a)*a)`. Given that `*` normally has higher precedence than `+`, the first derivation tree is desirable. We use `>`, **`left`**, and **`right`** to define priority and left- and right-associativity, respectively [3]. Figure 6 (right) shows the disambiguated version of this grammar by specifying `>` and **`left`**, where `-` has the highest precedence, and `*` and `+` are left-associative.

Ambiguity in expression grammars is caused by derivations from the left- or right-recursive ends in a rule, i.e., $E ::= \alpha \underline{E}$ and $E ::= \underline{E}\beta$. We use `>`, **`left`**, and **`right`** to specify which derivations from the left- and right-recursive ends are not valid with respect to operator precedence. For example, `E ::= '-' E > E '*' E` specifies that `E` in the `'-'`-rule

```
E ::= '-' E                      |E ::= '-' E
  | E '*' E                      |  > E '*' E       left
  | E '+' E                      |  > E '+' E       left
  | 'if' E 'then' E 'else' E     |  > 'if' E 'then' E 'else' E
  | a                            |  | a
```

Figure 6. An ambiguous expression grammar (left), and the same grammar disambiguated with `>` and **`left`** (right).

(parent) should not derive the `'*'`-rule (child). The `>` construct only restricts the right-recursive end of a parent rule when the child rule is left-recursive, and vice versa. For example, in Figure 6 (right) the right `E` in the `'+'`-rule is not restricted because the `'if'`- rule is not left-recursive. This is to avoid parse error on inputs that are not ambiguous, e.g., `a + if a then a else a`. Note that `'if' E 'then' E 'else'` in the `'if'`-rule acts as a unary operator. In addition, the `>` operator is transitive for all the alternatives of an expression nonterminal. Finally, **`left`** and **`right`** only affect binary recursive rules and only at the left- and right-recursive ends.

Although `>` is defined as a relationship between a parent rule and a child rule, its application may need to be arbitrary deep in a derivation tree. For example, consider the input string `a * if a then a else a + a` for the grammar in Figure 6 (right). This sentence is ambiguous with two derivation trees that correspond to the following groupings:

```
(a * (if a then a else a)) + a
 a * (if a then a else (a + a))
```

The first grouping is not valid as `'if'` binds stronger than `'+'`, but we defined `'+'` to have higher priority than `'if'`. This example shows that restricting derivations only at one level cannot disambiguate such cases. A correct implementation of `>` thus also restricts the derivation of the `'if'`-rule from the right-recursive end of the `'*'`-rule if the `'*'`-rule is derived from the left-recursive end of the `'+'`-rule.

We now show how to implement an operator precedence disambiguation scheme using data-dependent grammars. We first demonstrate the basic translation scheme using binary operators only, and then discuss the translation of the example in Figure 6. Figure 7 (left) shows a simple example of an expression grammar that defines two left-associative binary operators `*` and `+`, where `*` is of higher precedence than `+`. Figure 7 (right) shows the result of the translation into the data-dependent counterpart. The basic idea behind the translation is to assign a number, a *precedence level*, to each left- and/ or right recursive rule of nonterminal `E`, to parameterize `E` with a precedence level, and based on the precedence level passed to `E`, to exclude alternatives that will lead to derivation trees that violate the operator precedence.

In Figure 7 (right) each left- and right-recursive rule in the grammar gets a precedence level (shown in comments), which is the reverse of the alternative number in the definition of `E`. The precedence counter starts from 1 and increments for each encountered `>` in the definition. The number 0 is reserved for the unrestricted use of `E`, illustrated using the round bracket rule. Nonterminal `E` gets parameter `p` to pass the precedence level, and for each left- and right-recursive rule, a predicate is added at the beginning of the rule to exclude rules by comparing the precedence level of the rule with the precedence level passed to the parent `E`. Finally, for each use of `E` in a rule, an argument is passed.

```
E ::= E '*' E   left |E(p) ::= [2 >= p] E(2) * E(3)  //2
   >  E '+' E   left |       |  [1 >= p] E(0) + E(2)  //1
   |  '(' E ')'      |       |  '(' E(0) ')'
   |  a              |       |  a
```

Figure 7. An expression grammar with `>` and **`left`** (left), and its translation to data-dependent grammars (right).

```
E(l,r) ::= [4 >= l] '-' E(l,4)                              //4
  | [3 >= r, 3 >= l] E(3,3) '*' E(l,4)                       //3
  | [2 >= r, 2 >= l] E(2,2) '+' E(l,3)                       //2
  | [1 >= l] 'if' E(0,0) 'then' E(0,0) 'else' E(0,0) //1
  | a
```

Figure 8. Operator precedence with data-dependent grammars (binary and unary operators).

In the `'*'`-rule, its precedence level (`2`) is passed to the left `E`, and its precedence level plus one (`3`) is passed to the right `E`. This allows to exclude the rules of lower precedence from the left `E`, and to exclude the rules of lower precedence and the `'*'`-rule itself from the right `E`. Excluding the `'*'`-rule itself allows only the left-associative derivations, e.g., `(a*a)*a`, as specified by **`left`**. In the `'+'`-rule, its precedence level plus one (`2`) is passed to the right `E`, excluding the `'+'`-rule. The value `0` is passed to the left `E`, permitting any rule. Note that passing `0` instead of `1` to the left `E` of the `'+'`-rule achieves the same effect but enables better sharing of calls to `E`, as the sharing of calls (using GSS) is done based on the name of the nonterminal and the list of arguments. In the round bracket rule, `0` is passed to `E` as the use of `E` is neither left- nor right-recursive, hence, the precedence does not apply.

Now we discuss the translation of the example shown in Figure 6 that contains both binary and unary operators. For this we need to distinguish between the rules that should be excluded from the left and from the right `E`. This is achieved as follows. First, `E` gets two parameters, `l` and `r` (Figure 8), to distinguish between the precedence level passed from left and right, respectively. Second, a separate condition on `l` is added to a rule when the rule can be excluded from the right `E` (i.e., rules for binary operators and unary postfix operators). A separate condition on `r` is added to a rule when the rule can be excluded from the left `E` (i.e., rules for binary operators and unary prefix operators). Third, `l`- and `r`-arguments are determined for the left and right `E`'s as follows. An `l`-argument to the left `E` and `r`-argument to the right `E` are determined as in the example of Figure 7. For example, `E(3,_) '*' E(_,4)`, where `3` is the precedence level of the `'*'`-rule, and `4` is the precedence level plus one. Note that `r=4` does not exclude the unary operators of `E`. Now, an `l`-argument to the right `E`'s is propagated from the

```
Decls ::= align (offside Decl)*
        | ignore('{' Decl (';' Decl)* '}')
Decl  ::= FunLHS RHS
RHS   ::= '=' Exp 'where' Decls
```

Figure 9. Simplified version of Haskell's `Decls`.

parent `E`. This effectively excludes a unary prefix rule from the right `E`'s when the parent `E` is the left `E` of a rule of higher precedence than the unary operator. Finally, given that there are no unary postfix operators, an `r`-argument to the left `E`'s is not propagated from the parent `E` and can be the same as the respective `l`-argument.

We have also extended this approach for grammars that allow rules of the same precedence and/or associativity groups. For example, binary `+` and `-` operators have the same precedence, but are left-associative with respect to each other.

Our translation of operator precedence to data-dependent grammars resembles the precedence climbing technique [5, 31]. In contrast to precedence climbing that requires a non-left recursive grammar, our approach works in presence of both left- and right-recursive rules.

3.5 Indentation-sensitive Constructs

In this section we show how the offside rule can be translated into data-dependent grammars. We use Haskell as the running example, but our approach is also applicable for other programming languages that implement the offside rule.

In Haskell, one can write a `where` clause consisting of a block of declarations, where the structure of the block is defined by using either explicit delimiters or indentation (column number). For example, the structure of the following blocks, one written with explicit delimiters, such as curly braces and semicolons (left), and the other written using indentation (right), is the same:

```
{ x = 1 * 2 + 3; y = x + 4 }          x = 1 * 2
                                            + 3
                                      y = x + 4
```

Figure 9 shows a simplified excerpt of the Haskell grammar, defined using our parsing framework. The first alternative explicitly enforces indentation constraints on a declaration block. First, it requires that all declarations of a block are aligned (**`align`**) with respect to each other, i.e., each declaration starts with the same indentation. Second, it requires that the offside rule applies to each declaration, i.e., all non-whitespace tokens of a declaration are strictly indented to the right of its first non-whitespace token. In contrast, the second alternative of `Decls` enforces the use of curly braces and semicolons, and explicitly ignores (**`ignore`**) indentation constraints even when imposed by an outer scope.

In our meta-notation, **`align`** only affects regular definitions (EBNF constructs) such as lists and sequences, **`offside`** affects nonterminals, and **`ignore`** applies to a sequence of symbols. The translation of these high-level constructs into data-dependent grammars is illustrated in Figures 10 and 11.

```
Decls ::= a0:Star1(a0.l)
        | ignore('{' Decl Star2 '}')
Decl  ::= FunLHS RHS
RHS   ::= '=' Exp 'where' Decls

Star1(v) ::= Plus1(v) | ϵ
Plus1(v) ::= offside a1:Decl [col(a1.l) == col(v)]
           | Plus1(v) offside a1:Decl [col(a1.l) == col(v)]
Star2 ::= Plus2 | ϵ
Plus2 ::= Plus2 Seq2 | Seq2
Seq2  ::= ';' Decl
```

Figure 10. Desugaring of **`align`**.

```
Decls(i,fst) ::= a0:Star1(a0.l,i,fst)
               | '{' Decl(-1,0) Star2 '}'
Decl(i,fst)  ::= FunLHS(i,fst) RHS(i,0)
RHS(i,fst)   ::= o0:'=' [f(i,fst,o0.l)] Exp(i,0)
                 o1:'where' [f(i,0,o1.l)] Decls(i,0)

Star1(v,i,fst) ::= Plus1(v,i,fst) | ϵ
Plus1(v,i,fst)
  ::= Plus1(v,i,fst) a1:Decl(a1.l,1)
                     [col(a1.l) == col(v), f(i,0,a1.l)]
    | a1:Decl(a1.l,1) [col(a1.l) == col(v), f(i,fst,a1.l)]
Star2 ::= Plus2 | ϵ
Plus2 ::= Plus2 Seq2 | Seq2
Seq2  ::= ';' Decl(-1,0)

f(i,fst,l) = i == -1 || fst == 1 || col(l) > col(i);
```

Figure 11. Desugaring of **`offside`** and **`ignore`**.

The basic idea of translating **`align`** is to use the start index of a declaration list, and constrain the start index of each declaration in the list by an equality check on indentation at the respective indices. Figure 10 shows the result after first desugaring **`align`** and then translating EBNF constructs (Section 3.1). Desugaring **`align`** alone results in:

```
Decls ::= a0:(offside a1:Decl [col(a1.l) == col(a0.l)])*
```

Labels `a0` and `a1` are introduced to refer to the start index of a declaration list, `a0.l`, and to the start index of each declaration in the list, `a1.l`, respectively, and the constraint checks whether the respective column numbers (given tabs of 8 characters) are equal. As in case of precede restrictions in Section 3.3, this constraint only depends on the start indices and can be applied before parsing `Decl`. The EBNF translation introduces nonterminals for each EBNF construct, where `Star1` and `Plus1` also get parameter `v` as the use of `a0` has to be lifted during the translation.

Figure 11 shows the result of desugaring **`offside`** and **`ignore`** from Figure 10. The basic idea is to pass down `Decl`'s start index and constrain the indentation of any non-whitespace terminal that can appear under the `Decl`-node, except for the leftmost one, to be greater than the indentation of `Decl`'s start index. Two parameters, `i` and `fst`, are introduced to `Decl` and to all nonterminals reachable from it. The first parameter is used to pass `Decl`'s start index, calculated at the **`offside`** application site (`a1.l`), to any nonterminal reachable from `Decl`, and to constrain terminals reachable from `Decl`.

The second parameter, `fst`, which is either `0` or `1`, is used to identify and skip the leftmost terminal that should not be constrained. The value `1` is passed at the application site of **`offside`** and propagated down to the first nonterminal of each reachable rule if the rule starts with a nonterminal. The value `0` is passed to any other nonterminal of a reachable rule when the first symbol of the rule is not nullable. Our translation also accounts for nullable nonterminals (not shown here), and in such cases the value of `fst` also depends on a dynamic check whether the left and right extents of the node corresponding to a nullable nonterminal are equal.

Finally, each terminal reachable from `Decl` gets a label (labels starting with `o`), to refer to its start index, and a constraint, encoded as a call to boolean function `f`. Note that in the definition of `f`, condition `i == -1` corresponds to the case when `Decl` appears in the context where the offside rule does not apply or is ignored, and condition `fst == 1` to the case of the leftmost terminal.

The **`offside`**, **`align`** and **`ignore`** constructs are examples of reasonably complex desugarings to data-dependent grammars. Their existence and their aptness to describe the syntax of Haskell is a witness of the power of data-dependent grammars and the parsing architecture we propose.

3.6 Conditional Directives

In this section we present our solution for parsing conditional directives in C#. As discussed in Section 2.5, most directives can be regarded as comment, but conditional directives have to be evaluated during parsing, as they may affect the syntactic structure of a program.

Conditional directives can appear anywhere in a program. Therefore, it is natural to define them as part of the layout nonterminal. Figure 12 shows relevant parts of the layout definition (`Layout`)[2] we used to parse C# (follow restrictions enforce longest match). In addition to whitespace characters (`Whitespace`) and comments (starting with `'/*'` or `'//'`), the layout consists of declaration directives (`Decl`) and conditional directives (`If`).

According to the C# language specification, the scope of symbols introduced by declaration directives `#define` and `#undef` is the file they appear in. Therefore, we need to maintain a global symbol table `defs` to declare (see `Decl`) and access (see `If` and `Elif`) symbol definitions while parsing. In C# one can define/undefine a symbol, but a value cannot be assigned to a symbol. Thus, the symbol table needs to associate a boolean value with a symbol.

To enable global definitions, our parsing framework supports **global variables** that can be declared using the **`global`** keyword. e.g., `defs` in Figure 12. In our parsing framework, a global variable is implemented by using parameters and return values to thread a value through a parse. In this case,

```
global defs = {}

Layout ::= (Whitespace | Comment | Decl | If | Gbg)*
              !>> [\ \t\n\r\f] !>> '/*' !>> '//' !>> '#'

Decl ::= '#' 'define' id:Id
                        {defs=put(defs,id.yield,true)} PpNL
       | '#' 'undef' id:Id
                        {defs=put(defs,id.yield,false)} PpNL

If   ::= '#' 'if' v=Exp(defs) [v] ? Layout
                             : (Skipped (Elif|Else|PpEndif))
Elif ::= '#' 'elif' v=Exp(defs) [v] ? Layout
                             : (Skipped (Elif|Else|PpEndif))
Else ::= '#' 'else' Layout

Gbg     ::= GbgElif* GbgElse? '#' 'endif'
GbgElif ::= '#' 'elif' Skipped
GbgElse ::= '#' 'else' Skipped

Skipped ::= Part+
Part     ::= PpCond | PpLine | ... // etc.
PpCond   ::= PpIf PpElif* PpElse? PpEndif
PpIf     ::= '#' 'if' PpExp PpNL Skipped?
PpElif   ::= '#' 'elif' PpExp PpNL Skipped?
PpElse   ::= '#' 'else' PpNL Skipped?
PpEndif ::= '#' 'endif' PpNL
```

Figure 12. The grammar of conditional directives in C#.

each nonterminal that directly or indirectly accesses a global variable should get an extra parameter, and each nonterminal that can directly or indirectly update a global variable should return the new value of the variable if the variable is used after an occurrence of the nonterminal in a rule. Note that, assuming immutable values, such implementation of global variables properly accounts for the nondeterministic nature of generalized parsing. This way updates to a variable made along one parse do not interfere with updates made along an alternative parse.

The basic idea of single-phase parsing of C# in presence of conditional directives is as follows. Recall that in our parsing strategy (Section 3.2) layout is inserted between symbols in a grammar rule. Conditional directives are evaluated as part of the layout nonterminal, and based on the result of the evaluation, the next lines of source code are either treated as the actual source code (true case), or as a sequence of valid C# tokens (false case), also consuming directives that should not be evaluated. To achieve this, the grammar of Figure 12 uses two different definitions for `#if`, `#elif` and `#else`. The bottom definition (`PpIf`, `PpElif` and `PpElse`), which is found in the C# specification, simply defines directives as part of valid C# tokens (`Skipped`), while the top definition (`If`, `Elif` and `Else`) uses data dependency. Note that conditional directives can be nested. This is expressed by using `Layout` in `If`, `Elif` and `Else`, and `Skipped` in `PpIf`, `PpElif` and `PpElse`.

Whenever an `#if`-directive and its expression are parsed as part of `If`, the expression is evaluated using the symbol table (`defs`). `Exp` (not shown in Figure 12) defines a simple boolean expression. To enable evaluation of expressions

[2] For readability reasons, we omit uses of `Whitespace?` (optional whitespace) before and after terminal `'#'`, and uses of `Whitespace` after terminals `'define'`, `'undef'`, `'if'`, `'elif'`.

```
Element ::= STag Content ETag
STag    ::= '<' Name Attribute* '>'
ETag    ::= '</' Name '>'

Element ::= s=STag Content ETag(s)
STag    ::= '<' n:Name Attribute* '>' { n.yield }
ETag(s) ::= '</' n:Name [n.yield == s] '>'
```

Figure 13. Context-free grammar of XML elements (top) and the data-dependent version (bottom).

while parsing, Exp uses data dependency and extends the PpExp rules, found in the C# specification, with return values and boolean computation. If the expression evaluates to true (note the use of conditional selection), the parser first continues consuming layout, including the nested directives, and then, after no layout can be consumed, the parser returns to the next symbol in the alternative.

If the expression evaluates to false, the parser consumes part of the input as a list of valid C# tokens (Skipped) until it finds the corresponding #elif-, #else- or #endif-part. Note that Skipped also consumes nested #if-directives (PpCond), if any, but in this case, conditions are not evaluated. The definition of Skipped also allows to consume invalid C# structure (only valid token-wise) when the condition is false, see Figure 3 (right). Finally, when all #if, #elif and #else directives are present, there will be dangling #elif, #else, and #endif parts remaining if one of the conditions evaluates to true. These dangling parts should be also consumed by the layout. The Gbg (garbage) nonterminal, defined as part of layout, does exactly this.

3.7 Miscellaneous Features

In this section we discuss the use of data-dependent grammars for parsing XML and resolving the infamous typedef ambiguity in C. XML has a relatively straightforward syntax. Figure 13 (top) shows the context-free definition of Element in XML, where Content allows a list of nested elements. The problem with this definition is that it can recognize inputs with unbalanced start and end tags, for example:

```
<note>
  <to>Bob</from>
  <from>Alice</to>
</note>
```

Using data-dependent grammars, the solution to match start and end tags is very intuitive. Figure 13 (bottom) shows a data-dependent grammar for XML elements. As can be seen, inside a starting tag, STag, the result of parsing Name is bound to n, and the respective substring, n.yield, is returned from the rule. The returned value is assigned to s in the Element rule, and is passed to the end tag, ETag. Finally, in the ETag, the name of the end tag is checked against the name of the starting tag. If the name of the starting tag is not equal to the name of the end tag, i.e., n.yield == s does not hold, the parsing pass dies.

```
global defs = [{}]

Declaration ::= x=Specifiers Declarators(x)

Specifiers ::= x=Specifier y=Specifiers {x || y}
             | x=Specifier {x}
Specifier  ::= "typedef" {true} | ...

Declarators(x) ::= s=Declarator {h=put(head(defs),s,x);
                                 defs=list(h,tail(defs))}
                   ("," Declarators(x))*
Declarator ::= id:Identifier {id.yield}
             | x=Declarator "(" ParameterTypeList ")" {x}
             | ...
Expr ::= Expr "-" Expr
       | "(" n:TypeName [isType(defs,n.yield)] ")" Expr
       | "(" Expr ")"
       | ...
       | Identifier [!isType(defs,n.yield)]
```

Figure 14. Resolving typedef ambiguity in C.

Now, we consider the problem of typedef ambiguity in C. For example, expression (T)+1 can have two meanings, depending on the meaning of T in the context: a cast to type T with +1 being a subexpression, or addition with two operands (T) and 1. If T is a type, declared using typedef, the first parse is valid, otherwise the second one.

To resolve the typedef ambiguity, type names should be distinguished from other identifiers, such as variables and function names, during parsing. In addition, the scoping rules of C should be taken into account. For example, consider the following C program:

```
typedef int T;
main() {
  int T = 0, n = (T)+1;
}
```

In this example, T is first declared as a type alias to int and then redeclared as a variable of type int in the inner scope introduced by the main function.

Figure 14 shows a simplified excerpt of our data-dependent C grammar. The excerpt shows the declaration and expression parts of the C grammar. As can be seen, a C declaration consists of a list of specifiers followed by a list of declarators. Each declarator declares one identifier. Keyword typedef can appear in the list of specifiers, for example, along with the declared type. A declarator can be either a simple identifier or a more complicated syntactic structure, e.g., array and function declarators, nesting the identifier. It is important to note that an identifier should enter the current scope when its declarator is complete. The expression part of Figure 14 shows the cast expression rule (the second rule from top), and the primary expression rule (the last one). Note that to resolve the typedef ambiguity, illustrated in our running example, an identifier should be accepted as an expression if it is not declared as a type name.

To distinguish between type names and other identifiers, we record names, encountered as part of declarators, and associate a boolean value with each name: true for type names and `false` otherwise. To maintain this information during parsing, we introduce global variable `defs`, holding a list of maps to properly account for scoping. At the beginning of parsing, `defs` is a list containing a single, empty map. At the beginning of a new scope, i.e., when `"{"` is encountered, an empty map is prepended to the current list resulting in a new list which is assigned to `defs` (not shown in the figure). At the end of the current scope, i.e., when `"}"` is encountered, the head of the current list is dropped by taking the tail of the list and assigning it to `defs`.

To communicate the presence of `typedef` in a list of specifiers, we extend each rule of `Specifier` to return a boolean value: `"typedef"`-rule returns `true`, and the other rules return `false`. `Specifiers` computes disjunction of the values associated with the specifiers in the resulting list. This information is passed via variable `x` to `Declarators`. We also extend the rules of `Declarator` to return the declared name, `id.yield`. After a declarator is parsed, the declared name can be stored in `defs`: pair `(s,x)` is added to the map taken from the head of the current list, and a new list, with the resulting map as its head, is created and assigned to `defs`.

Finally, `isType` function is used to check whether the current identifier is a type name in the current scope or not: `isType` iterates over elements in `defs`, starting from the first element, to look up the given name. If the name is not found in the current map, `isType` continues the search with the next element, representing the outer scope. If the name is found, `isType` returns the boolean value associated with the name. If none of the maps contains the name, `isType` returns `false`.

In our running example, after parsing the second declaration of `T`, appeared in the scope of the `main` function, pair `("T",false)` will be added to the map in the head of `defs`, effectively shadowing the previous typedef declaration of `T`, and causing the condition in the cast expression rule to fail.

4. Implementation

In this section we present our extension of the GLL parsing algorithm [33] to support data-dependent grammars. GLL parsing is a generalization of recursive-descent parsing that supports all context-free grammars, and produces a binarized Shared Packed Parse Forest (SPPF) in cubic time and space. GLL uses a Graph-Structured Stack (GSS) [35] to handle multiple function calls in recursive-descent parsing. The problem of left recursion is solved by allowing cycles in GSS. As GLL parsers are recursive-descent like, the handling of parameters and environment is intuitive, and the implementation remains very close to the stack-based semantics, which eases the reasoning about the runtime behavior of the parser. More information on GLL parsing over ATN, GSS, and SPPF is provided in Appendix A.

We use a variation of GLL parsing that uses a more efficient GSS [2]. GLL parsing can be seen as a grammar traversal process that is guided by the input. At each point during parsing, a GLL parser is at a grammar slot (grammar position before or after a symbol in a rule) and executes the code corresponding to this slot. Because of the nondeterministic nature of general parsing, a GLL parser needs to record all possible paths and process them later, and at the same time eliminate duplicate jobs. The unit of work in GLL parsing is a *descriptor* which captures a parsing state. Descriptors allow a serialized, discrete view of tasks performed during parsing. GLL parsing has a main loop, in a trampolined style, that executes the descriptors one at a time until no more descriptors left.

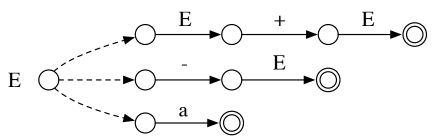


Figure 15. ATN Grammar for $E ::= E + E \mid -E \mid a$.

The standard way of implementing a GLL parser is to generate code for each grammar slot [33]. Such implementation relies on *dynamic goto*s to allow arbitrary jumps to the main loop or other grammar slots. In our GLL implementation, a grammar is modeled as a connected graph of grammar slots. This model of context-free grammars resembles Woods' *Recursive Augmented Transition Networks* (ATN) [40] grammars. As such, our implementation of GLL over ATN grammars (Appendix A) provides an interpreter version of GLL parsing.

4.1 ATN Grammars

ATN grammars are an automaton formalism developed in the 70s to parse natural languages, and are similar to nondeterministic finite automata.

An ATN grammar is a tuple $(Q, F, \rightarrow)$ where

- Q is a finite set of states representing grammar slots;
- $F \subset Q$ is a finite set of states representing *final* grammar slots; and
- $\rightarrow$ is a transition relation of the form $\xrightarrow{A}$ (nonterminal), $\xrightarrow{t}$ (terminal), or $\xrightarrow{\epsilon}$ (epsilon).

For example, the ATN grammar for $E ::= E + E \mid -E \mid a$ is shown in Figure 15. In an ATN, there is a one-to-many relation, $S \subset \text{String} \times Q$, from a nonterminal name to a set of start states, each representing the initial state of an alternative.

Constructing an ATN grammar from a CFG is straightforward. For each nonterminal in the grammar, and for each alternative of the nonterminal, a pair consisting of the nonterminal's name and a state representing the start state of the alternative is added to S. Finally, for each symbol in the alternative, a next state is created, and a transition, labeled with the symbol, from the previous state to this state is added. The last state of the alternative is marked as a final grammar slot.

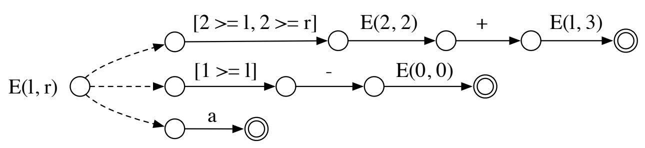


Figure 16. Data-dependent ATN grammar for $E ::= E + E > -E \mid a$ after desugaring operator precedence.

4.2 Data-dependent ATN Grammars

To support data-dependent grammars, we extend ATN grammars with the following forms of transitions:

- $\xrightarrow{x=l:A(e)}$ (parameterized, labeled nonterminals) and $\xrightarrow{l:t}$ (labeled terminals)
- $\xrightarrow{x=e}$ (variable binding), $\xrightarrow{[e]}$ (constraint) and $\xrightarrow{e}$ (return expression).

Two additional mappings are maintained: $L, X : Q \rightarrow \text{String}$ that map a state, representing a grammar slot after a labeled nonterminal, to the nonterminal's label (l) and to the nonterminal's variable (x), respectively. Here, as in Section 3.1, for simplicity of presentation and without loss of generality, we assume that nonterminals can have at most one parameter. We also only consider cases of labeled terminals and nonterminals, and when a return expression is present. Finally, we assume that expression language e is a simple functional programming language with immutable values and no side-effects, that labels and variables are scoped to the rules they are introduced in, and that labels and variables introduced by desugaring have unique names in their scopes.

An example of a data-dependent ATN is shown in Figure 16. This ATN grammar is the disambiguated version of the grammar shown in Figure 15 after desugaring operator precedence.

4.3 Data Dependency in GLL Parsing

In the following, p, q, s represent ATN states in Q, i is an input index, u, u' represent GSS nodes, and w, n, y represent SPPF nodes. To support data-dependent grammars, we introduce an environment, E, into GLL parsing. Here, we assume that E is an immutable map of variable names to values. In the data-dependent setting, a descriptor, the unit of work in GLL parsing, is of the form (p, i, E, u, w). Now, a descriptor contains an environment E that has to be stored and later used whenever the parser selects the descriptor to continue from this point. GSS is also extended to store additional data. A GSS node and a GSS edge are now of the forms (A, i, v) and (u, p, w, E, u'), respectively. That is, in addition to the current input index i, a GSS node stores an argument v, passed to a nonterminal A, to fully identify the call. A GSS edge additionally stores an environment E, to capture the state of the parser before a call to a nonterminal is made.

Finally, a GLL parser constructs a binarized SPPF (Appendix A.1), creating terminal nodes (**nodeT**), and nonterminal and intermediate nodes (**nodeP**). In GLL parsing intermediate nodes are essential. In particular, they allow the parser to carry a single node at each time by grouping the symbols of a rule in a left-associative manner. Nonterminal and intermediate nodes can be ambiguous. To properly handle ambiguities under nonterminal and intermediate nodes, we include environment and return values into the SPPF construction. Specifically, arguments to nonterminals and return values are part of nonterminal nodes, and environment is part of intermediate nodes.

Figure 17 presents the semantics of GLL parsing over ATN, defining it as a transition relation on configuration $(\mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P})$ where the elements are four main structures maintained by a GLL parser:

- $\mathcal{R}$ is a set of pending descriptors to be processed
- $\mathcal{U}$ is a set of descriptors created during parsing. This set is maintained to eliminate duplicate descriptors
- $\mathcal{G}$ is a GSS, represented by a set of GSS edges
- $\mathcal{P}$ is a set of parsing results (SPPF nodes created for nonterminals) associated with GSS nodes, i.e., a set of elements of the form (u, w)

During parsing, a descriptor is selected and removed from $\mathcal{R}$, represented as $\{(p, i, E, u, w)\} \cup \mathcal{R}$, and given the rules, a deterministic choice is made based on the next transition in the ATN. The simplest rules are Eps, Cond-1, Cond-2 and Bind. Eps creates the ϵ-node (via call to **nodeT**) and an intermediate node (via call to **nodeP**), and adds a descriptor for the next grammar slot. Cond-1 and Cond-2 depend on the evaluation of expression e in a constraint. If the expression evaluates to true, a new descriptor is added to continue with the next symbol in the rule (Cond-1), otherwise no descriptor is added (Cond-2). Bind evaluates the expression in an assignment and creates a new environment containing the respective binding. This environment is used to create the new descriptor added to $\mathcal{R}$.

Term-1 and Term-2 deal with labeled terminals. If terminal t matches (Term-1) the input string (represented by an array I) starting from input position i, a terminal node is created (assuming t is of length 1). Then, the properties, i.e., the left and right extents, and the respective substring, are computed from the resulting node (props(y)). Finally, a new environment, containing binding $[l = \text{props}(y)]$, is created and used to construct an intermediate node and a new descriptor. If the terminal does not match (Term-2), no descriptor is added.

Call-1 and Call-2 deal with labeled calls to nonterminals. First, argument e is evaluated, where E_1 allows the use of the left extent in e (lprop constructs properties with only left extent). If a GSS node, representing this call, already exists (Call-1), the parsing results associated with this GSS node are reused, and a possibly empty set of new descriptors ($\mathcal{D}$)

$$\boxed{(\mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R}', \mathcal{U}', \mathcal{G}', \mathcal{P}')}$$

$$\text{Eps}\;\frac{\begin{array}{c} p \xrightarrow{\epsilon} q \\ n = \textbf{nodeP}(q, w, \textbf{nodeT}(\epsilon, i, i), E) \quad d = (q, i, E, u, n) \end{array}}{(\{(p, i, E, u, w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R} \cup \{d\}, \mathcal{U}, \mathcal{G}, \mathcal{P})}$$

$$\text{Term-1}\;\frac{\begin{array}{c} p \xrightarrow{l:t} q \quad I[i] = t \\ y = \textbf{nodeT}(t, i, i+1) \quad E_1 = E[l = \text{props}(y)] \\ n = \textbf{nodeP}(q, w, y, E_1) \quad d = (q, i+1, E_1, u, n) \end{array}}{(\{(p, i, E, u, w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R} \cup \{d\}, \mathcal{U}, \mathcal{G}, \mathcal{P})}$$

$$\text{Term-2}\;\frac{p \xrightarrow{l:t} q \quad I[i] \neq t}{(\{(p, i, E, u, w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P})}$$

$$\text{Call-1}\;\frac{\begin{array}{c} p \xrightarrow{x = l:A(e)} q \\ E_1 = E[l = \text{lprop}(i)] \quad [\![e]\!]E_1 = v \quad u' = (A, i, v) \in \mathcal{N}(\mathcal{G}) \\ \mathcal{D} = \{d \mid (u', y) \in \mathcal{P}, E_2 = E[l = \text{props}(y), x = \text{val}(y)], \\ d = (q, \text{rext}(y), E_2, u, \textbf{nodeP}(q, w, y, E_2)), d \notin \mathcal{U}\} \end{array}}{\begin{array}{c}(\{(p, i, E, u, w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R} \cup \mathcal{D}, \mathcal{U} \cup \mathcal{D}, \\ \mathcal{G} \cup \{(u', q, w, E, u)\}, \mathcal{P})\end{array}}$$

$$\text{Call-2}\;\frac{\begin{array}{c} p \xrightarrow{x = l:A(e)} q \\ E_1 = E[l = \text{lprop}(i)] \quad [\![e]\!]E_1 = v \quad u' = (A, i, v) \notin \mathcal{N}(\mathcal{G}) \\ \mathcal{D} = \{(s, i, [p_0 = v], u', \$) \mid s \in S(A)\} \end{array}}{\begin{array}{c}(\{(p, i, E, u, w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R} \cup \mathcal{D}, \mathcal{U}, \\ \mathcal{G} \cup \{(u', q, w, E, u)\}, \mathcal{P})\end{array}}$$

$$\text{Ret}\;\frac{\begin{array}{c} p \xrightarrow{e} q \quad q \in F \\ [\![e]\!]E = v \quad n = \textbf{nodeP}(q, w, \text{arg}(u), v) \\ \mathcal{D} = \{d \mid (u, s, y, E_1, u') \in \mathcal{G}, E_2 = E_1[L(s) = \text{props}(n), X(s) = v], \\ d = (s, i, E_2, u', \textbf{nodeP}(s, y, n, E_2)), d \notin \mathcal{U}\} \end{array}}{(\{(p, i, E, u, w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R} \cup \mathcal{D}, \mathcal{U} \cup \mathcal{D}, \mathcal{G}, \mathcal{P} \cup \{(u, n)\})}$$

$$\text{Cond-1}\;\frac{\begin{array}{c} p \xrightarrow{[e]} q \quad [\![e]\!]E = \text{true} \\ d = (q, i, E, u, w) \end{array}}{(\{(p, i, E, u, w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R} \cup \{d\}, \mathcal{U}, \mathcal{G}, \mathcal{P})}$$

$$\text{Cond-2}\;\frac{p \xrightarrow{[e]} q \quad [\![e]\!]E = \text{false}}{(\{(p, i, E, u, w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P})}$$

$$\text{Bind}\;\frac{\begin{array}{c} p \xrightarrow{x = e} q \quad [\![e]\!]E = v \\ d = (q, i, E[x = v], u, w) \end{array}}{(\{(p, i, E, u, w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R} \cup \{d\}, \mathcal{U}, \mathcal{G}, \mathcal{P})}$$

Figure 17. GLL for data-dependent ATN grammars.

is created. Each descriptor in the set corresponds to a result, nonterminal node y, retrieved from $\mathcal{P}$, so that the index of the descriptor is the right extent of y (rext), its environment contains bindings $[l = \text{props}(y)]$ and $[x = \text{val}(y)]$ (val retrieves the value from y), and its SPPF node is a new intermediate node. Note that $d \notin \mathcal{U}$ ensures that no duplicate descriptors are added at this point. If the corresponding GSS node does not exist, Call-2 creates one descriptor for each start state of the nonterminal ($s \in S(A)$). Each descriptor gets a new environment with binding $[p_0 = v]$, where p_0 is the nonterminal's parameter that we assume to have a unique name in the scope of a rule. Both Call-1 and Call-2 add a new GSS edge capturing the previous environment to $\mathcal{G}$.

Finally, in Ret-rule, the return expression is evaluated, and the nonterminal node is created which stores both the argument of the current GSS node ($\text{arg}(u)$) and the return value. This node is recorded in $\mathcal{P}$ as a result associated with the GSS node. For each GSS edge directly reachable from the current GSS node, a new descriptor is created. Note that labels and variables at call sites, represented by the current GSS node, are retrieved via mappings L and X, respectively.

5. Evaluation

Our data-dependent parsing framework is implemented as an extension of the Iguana parsing framework [2]. The addition of data-dependency is at the moment a prototype and most of the effort was put into correctness, rather than performance optimization. As a frontend to write data-dependent grammars, we extended the syntax definition of Rascal [24], a programming language for meta-programming and source code analysis, and provided a mapping to Iguana's internal representation of data-dependent grammars.

In Section 2 we enumerated a number of challenges in parsing programming languages, and in Section 3, we provided solutions based on data-dependent grammars (directly or via desugaring) that address these challenges. For each challenge we selected a programming language that exhibits it, and wrote a data-dependent grammar[3], derived from the specification grammar of the language. For evaluation, we parsed real source files from the source distribution of the language and some popular open source libraries, see Table 2. Table 1 summarizes the evaluation results. In the following we discuss these results in detail, and provide an analysis of the expected performance in practice.

Java To evaluate the correctness of our declarative operator precedence solution using data-dependent grammars, we used the grammar of Java 7 from the main part of the Java language specification [11]. This grammar contains an unambiguous left-recursive expression grammar, in a similar style to the expression grammar in Figure 1 (middle).

We replaced the expression part (consisting of about 30 nonterminals) of the Java specification grammar with a single `Expression` nonterminal that declaratively expresses operator precedence using `>`, **`left`** and **`right`**. The resulting grammar, which we refer to as the *natural* grammar, is much more concise and readable, see Table 1. The resulting parser parsed all 8067 files successfully and without ambiguity.

The natural grammar of Java produces different parse trees compared to the original specification grammar, and therefore it is not possible to directly compare the parse trees. To test the correctness of parsers resulting from the desugaring of `>`, **`left`**, and **`right`** to data-dependent grammars, we tested their resulting parse trees against a GLL parser for the same natural grammar of Java, using our previous work on rewriting operator precedence rules [3]. Both parsers, using desugaring to data-dependent grammars and rewriting operator precedence rules, produced the same

[3] `https://github.com/iguana-parser/grammars`

Table 1. Summary of the results of parsing with character-level data-dependent grammars of programming languages.

Language	Challenge	Solution	Spec. Grammar		Data-dep. Grammar		# Files	% Success
			# Nont.	# Rules	# Nont.	# Rules		
Java	Operator precedence	>, **`left`** and **`right`**	200	485	169	435	8067	100% (8067)
C#	Conditional directives	**`global`** variables and dynamic layout	387	1000	395	1013	5839	99% (5838)
Haskell	Indentation sensitivity	**`align`**, **`offside`** and **`ignore`**	143	431	152	452	6457	72% (4657)

parse trees for all Java files, providing an evidence that our desugaring of operator precedence to data-dependent grammars implements the same semantics as the rewriting in [3].

Despite its prototype status, the data-dependent parser is at the moment on average only 25% slower than the rewritten one. The main reason for performance difference is that in the rewriting technique [3] the precedence information is statically encoded in the grammar, and therefore there is no runtime overhead, while in the data-dependent version passing arguments and handling environment is done at runtime. The problem with the rewriting technique is that the rewriting process itself is rather slow and the resulting grammar is very large.

C# To evaluate our data-dependent framework on parsing conditional directives, we used the grammar of C# 5 from the C# language specification [30]. As mentioned in Section 2.5, existing C# compilers resolve preprocessor directives in the lexing phase, and the parser is not aware of directives. However, the C# language specification has context-free rules that describe the syntax of directives. Our solution to parsing conditional directives (Section 3.6) leverages layout that is automatically inserted between symbols in grammar rules. We used the context-free syntax of directives in C# as the starting point. We extended the layout definition to include directives. Then, the conditional directive rules were modified to allow parse-time evaluation of conditions and selection of the corresponding path.

The resulting data-dependent grammar is only different from the specification grammar in the layout definition, and the difference is minimal. As can be seen in Table 1 there are only 8 additional nonterminals and 13 additional rules (about 1.3% of the whole grammar). Using the character-level grammar of C# we could parse 5838 files out of 5839. The parser timed out after 30 seconds on a very large source file from the Roslyn framework. The file, which appears to be automatically generated, contains 156033 lines of code and is of size 4.8 MB.

Although the grammar of C# is near deterministic, the reason for time out is that character-level grammars generate very large parse trees, a node for each character. Nevertheless, this file could be parsed using a context-aware parser of C#. We discuss the performance gain of using a context-aware scanner in Section 5.1.

Haskell To evaluate our parsing framework for indentation sensitive programming languages, we used the grammar of Haskell [28]. The specification grammar of Haskell is written using explicit blocks, as if no indentation sensitivity exists, and the lexer translates indentation to physical block delimiters. We took the Haskell grammar as written in the specification as the starting point and added extra rules that specify layout sensitivity using **`align`**, **`offside`** and **`ignore`** constructs. As shown in Table 1, the data-dependent version has only 21 additional rules (about 4% of the whole grammar). From the total number of 6457 Haskell files, we could successfully parse 4657 files (72%). The reason for the parse error in other files is that they contained some syntactic constructs from GHC extensions that we do not support yet.

Table 2. Summary of the projects used in the evaluation.

Lang.	Projects	Version	Description
Java	JDK	1.7.0_60-b19	Java Development Kit
	JUnit	4.12	Unit testing framework
	SLF4J	1.7.12	A Java logging framework
C#	Roslyn	build-preview	.NET Compiler Platform
	MVC	6.0.0-beta5	ASP.NET MVC Framework
	EntityFramew.	7.0.0-beta5	Data access for .NET
Haskell	GHC	7.8	Glasgow Haskell Compiler
	Cabal	1.22.4.0	Build System for Haskell
	Git-annex	5.20150710	File manager based on Git
	Fay	0.23.1.6	Haskell to JavaScript compiler

Besides numerous undocumented GHC extensions we found in the source files, many Haskell files contained CPP directives which were resolved by running the C preprocessor, `cpp`, before parsing. In the future, we plan to deal with C directives during parsing, the same way we did for C#. One last issue about parsing Haskell is that indentation rules alone are not sufficient to unambiguously parse Haskell, and there is a need for a syntactic longest match that uses indentation information. For example, the following input string is ambiguous, where both derivations are correct regarding the indentation rules:

```
f x = do print x
           + 1
```

In the first derivation, the right hand side is an infix plus-expression, consisting of a `do`-expression and `1`. The second derivation consists of only a `do`-expression that has `print x + 1` as its subexpression. According to the Haskell language specification the second interpretation is valid, as in `do` expressions longest match should be applied. We resolved this issue by defining a special kind of follow re-

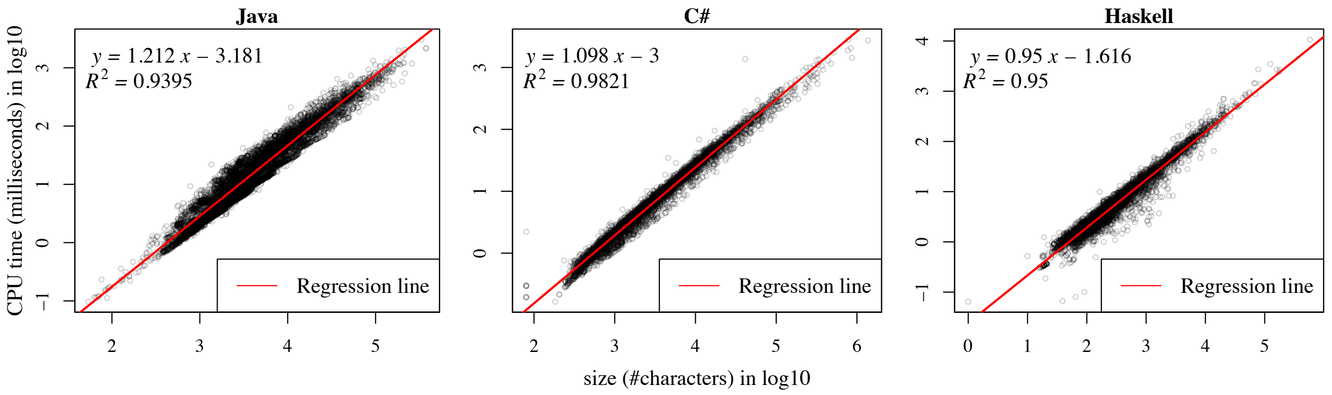


Figure 18. Running time of the character-level parsers for Java, C#, and Haskell against the input size (number of characters) plotted as log-log base 10. The red line is the linear regression fit. The goodness of each fit is indicated by the adjusted R^2 value in each log-log plot. The equation in each plot describes a power relation with original data, and as all the coefficients (1.212, 1.098, 0.950) are close to one, we can conclude the running time is near-linear on these grammars.

striction, similar to Section 3.3, that bypasses the layout and checks for the indentation level of the next non-whitespace token when the token is not a keyword or a delimiter.

OCaml We used excerpts of OCaml source files to test our operator precedence translation against deep and problematic operator precedence cases. OCaml, in contrast to other three programming languages we used for the evaluation, uses a natural, ambiguous expression grammar in its language specification. The data-dependent grammar of OCaml is basically the same as the reference manual, where the alternative operator in the expression part is replaced with `>` and additional **`left`** and **`right`** operators added. We are not yet able to unambiguously parse full OCaml programs, as they contain operator precedence ambiguities across indirect nonterminals. An example is `pattern-matching` which can derive `expr` on its right-most end:

```
expr ::= expr '+' expr
       | 'function'  pattern-matching
pattern-matching ::= pattern '->'  expr
```

For example, the input string `function x -> x + 1` is ambiguous with the following derivations: `(function x -> x) + 1` or `function x -> (x + 1)`. As the `function` alternative has `pattern-matching` and not `expr` on its right-most end, the operator precedence rules do not apply in our current scheme. The translation of operator precedence in presence of indirect nonterminals to data-dependent grammars seems possible with an additional analysis of indirect nonterminals, but is left as future work.

5.1 Running Time and Performance

Data-dependent grammars [16] provide a *pay-as-you-go* model. If a pure context-free grammar is specified, the worst-case complexity of the underlying parsing technology is retained. However, in the general case no guarantees can be made. Our data-dependent parsers, implemented on top of GLL parsing, are worst-case $O(n^3)$ on pure context-free grammars [33]. The more practical question, however, is how data dependency affects the runtime performance of parsing real programming languages.

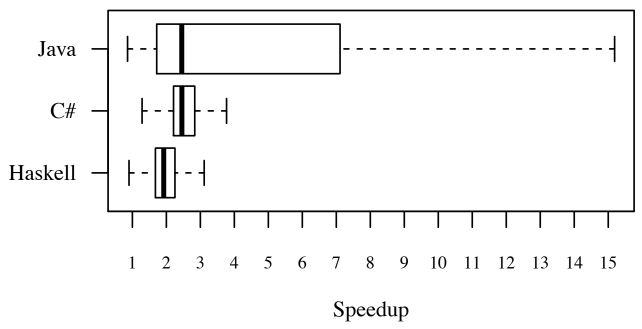


Figure 19. The relative speedup using context-aware scanning instead of character level grammars.

In this section, we provide empirical results showing that parsers for data-dependent grammars can behave nearly linearly on grammars of real programming languages. We ran the experiments on a machine with a quad-core Intel Core i7 2.6 GHz CPU and 16 GB of physical memory running Mac OS X 10.10.4. We used a 64-Bit Oracle HotSpot™ JVM version 1.8.0_25. Each file was executed 10 times and the mean running time (CPU user time) was reported. The three first runs of each file were skipped to allow for JIT optimizations.

Figure 18 shows the log-log plots (log base 10) of running time (ms) against file size (number of characters) for all the files we parsed (See Table 1 and 2). For showing the linear behavior we used the character-level grammars, as they exhibit the relationship between the running time and the number of characters better than the context-aware version. As can be seen, all three parsers exhibit near-linear behavior on grammars of programming languages.

To compare the performance difference between character-level and context-aware parsing, we ran both context-aware and character-level parsers on all the source files. Figure 19 shows the relative performance gain (speedup) using a context-aware parser compared to a character-level parser for each file. For a better visualization we omitted the outliers from the box plots. The median and maximum speedup for Java is (2.45, 15.1), for C# (2.45, 4) and for Haskell (1.9, 3). The precise impact of context-aware scanning for general parsing and data-dependent grammars is future work, but our preliminary investigation revealed that using character-level grammars for parsing layout is very expensive, as it is a very common operation, see Section 3.2.

6. Related Work

Parsing is a well-researched topic, and many features of our parsing framework are related in one or another way to other existing systems. Throughout this paper we have discussed some related work, which we do not repeat here. In this section we discuss direct related work and our inspirations.

Data dependency implementation Data-dependent grammars have many similarities with attribute grammars [27] and attribute directed parsing [39]. A detailed discussion of related systems is provided by Jim *et al.* [16]. From the implementation perspective, Jim *et al.* present the Yakker parser generator [16], which is based on Earley's algorithm [7], but we have a GLL-based interpretation of data-dependent grammars. We also extend the SPPF creation functionality of GLL parsing (taking environments into account), while SPPF creation is not discussed in Jim *et al.* 's approach. Another difference between our implementation and Yakker is that Yakker directly supports regular operators, by applying longest match. We, however, believe that all ambiguities should be returned by the parser, and avoid such implicit heuristics. Therefore, we desugar regular operators to data-dependent BNF rules.

We use an interpretative model of parsing based on Woods' ATN grammars [40]. Woods used an explicit stack to run ATN grammars, similar to a pushdown automata. However, as with any top-down parser, such execution of ATN grammars does not terminate in presence of left recursion. Jim *et al.* 's data-dependent framework operates on a data-dependent automata [15], which is a variation of ATN grammars interpreted with Earley's algorithm.

Indentation-sensitive parsing Besides modification to the lexer which has been used in GHC and CPython, there are a number of other systems that provide a solution for indentation-sensitive parsing. Parser combinators [13] are higher-order functions that are used to define grammars in terms of constructs such as alternation and sequence. This approach has been used in parsing indentation-sensitive languages [14]. Traditional parser combinators do not support left-recursion and can have exponential runtime. Another main difference between parser combinators and our approach is that we do not give the end user access to the internal workings of the parser. Since parser combinators are normal functions, the user can modify them. Our approach provides an external DSL for defining parsers, while parser combinators provide an internal DSL. Therefore, our approach compared to parser combinators provides more control over the syntax definition.

Erdweg *et al.* present an extension of SDF to define layout constraints on grammar rules [8]. These constraint-based approach is implemented by modifying the underlying SGLR [38] parser. Most constraints can be solved during parsing. Constraints that are not resolved will lead to ambiguity which can be removed by post-parse filtering. Adams presents the notion of indentation-sensitive grammars [1], where symbols in a rule are annotated by the relative position to the immediate parents. This technique is implemented for LR(k) parsing.

We do not offer a customized solution for indentation-sensitivity for a specific parsing technology, rather we use the general data-dependent grammars framework, and map indentation rules to them. In addition, we define high-level constructs such as **`align`**, **`offside`**, and **`ignore`** which are desugared to lower-level data-dependent grammars. This enables a syntax definition model that is closer to what the user has in mind. We think the use of high-level constructs leads to cleaner, more maintainable grammars.

Operator precedence SDF2 uses a parser-independent semantics of operator precedence which is based on parent-child relationship on derivation trees [38]. This semantics is implemented in SGLR parsing [38] by modifying parse tables. Although the SDF2 semantics for operator precedence works for most cases, in some cases it is too strong, i.e., rejecting valid sentences, and in some cases it cannot disambiguate the expression grammar.

In earlier work [3] we discussed the precedence ambiguity, and proposed a grammar rewriting that takes an ambiguous grammar with a set of precedence rules and produces a grammar that does not allow precedence-invalid derivations. Our current solution has the same semantics: it does not remove sentences when there is no precedence ambiguity, and can deal with corner cases found in programming languages such as OCaml. In addition, our operator precedence solution is desugared to data-dependent grammars, thus it is independent of the underlying parsing technology.

Conditional directives Recent work in parsing conditional directives target all variations [10, 21]. Gazzillo and Grimm [10] give an extensive overview of related work in this area. However, to the best of our knowledge, none of the existing systems employ a single-phase parsing scheme, rather they use a separate scanner and annotate the tokens based on the conditional directives they appear in. Our approach in using data-dependent grammars to evaluate the conditional directives is new. The treatment of other features of preprocessors, such as macros, is future work.

7. Conclusion

We have presented our vision of a parsing framework that is able to address many challenges of declarative parsing of real programming languages. We have built an implementation of data-dependent grammars based on the GLL parsing algorithm. We also have shown how to map common idioms in syntax of programming languages, such as lexical disambiguation filters, operator precedence, indentation-sensitivity, and conditional directives to data-dependent grammars. These mappings provide the language engineer with a set of out of the box constructs, while at the same time, new high-level constructs can be added. The preliminary experiments with our parsing framework show that it can be efficient and practical. To fully realize our vision we will explore more syntactic features, and further optimize the implementation of our framework.

Acknowledgments

We are thankful to Jurgen Vinju, Paul Klint, and the anonymous reviewers for their constructive feedback on earlier versions of this paper.

References

[1] M. D. Adams. Principled Parsing for Indentation-sensitive Languages: Revisiting Landin's Offside Rule. In *Principles of Programming Languages*, POPL '13, pages 511–522. ACM, 2013.

[2] A. Afroozeh and A. Izmaylova. Faster, Practical GLL Parsing. In *Compiler Construction, 24th International Conference*, CC '15, pages 89–108. Springer, 2015.

[3] A. Afroozeh, M. van den Brand, A. Johnstone, E. Scott, and J. J. Vinju. Safe Specification of Operator Precedence Rules. In *Software Language Engineering*, SLE '13, pages 137–156. Springer, 2013.

[4] A. V. Aho, S. C. Johnson, and J. D. Ullman. Deterministic Parsing of Ambiguous Grammars. In *Principles of Programming Languages*, POPL '73, pages 1–21, 1973.

[5] K. Clarke. The Top-down Parsing of Expressions. Technical report, Dept. of Computer Science and Statistics, Queen Mary College, London, June 1986.

[6] F. L. DeRemer. *Practical Translators for LR(k) Languages*. PhD thesis, Massachusetts Institute of Technology, 1969.

[7] J. Earley. An Efficient Context-free Parsing Algorithm. *Commun. ACM*, 13(2):94–102, Feb. 1970. ISSN 0001-0782.

[8] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Layout-Sensitive Generalized Parsing. In *Software Language Engineering*, SLE'12, pages 244–263. Springer, 2012.

[9] K. Fisher and R. Gruber. PADS: A Domain-specific Language for Processing Ad Hoc Data. In *Programming Language Design and Implementation*, PLDI '05, pages 295–304. ACM, 2005.

[10] P. Gazzillo and R. Grimm. SuperC: Parsing All of C by Taming the Preprocessor. In *Programming Language Design and Implementation*, PLDI '12, pages 323–334. ACM, 2012.

[11] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. The Java Language Specification Java SE 7 Edition, 2013.

[12] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The Syntax Definition Formalism SDF–Reference Manual–. *SIGPLAN Not.*, 24(11):43–75, Nov. 1989.

[13] G. Hutton. Higher-order Functions for Parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[14] G. Hutton and E. Meijer. Monadic Parsing in Haskell. *J. Funct. Program.*, 8(4):437–444, 1998.

[15] T. Jim and Y. Mandelbaum. Efficient Earley Parsing with Regular Right-hand Sides. 253(7):135 – 148, 2010. LDTA'09.

[16] T. Jim, Y. Mandelbaum, and D. Walker. Semantics and Algorithms for Data-dependent Grammars. In *Principles of Programming Languages*, POPL'10, pages 417–430. ACM, 2010.

[17] M. Johnson. The Computational Complexity of GLR Parsing. In *Generalized LR Parsing*, pages 35–42. Springer US, 1991.

[18] S. C. Johnson. Yacc: Yet Another Compiler-Compiler. Technical report, AT&T Bell Laboratories, 1979.

[19] A. Johnstone and E. Scott. Modelling GLL Parser Implementations. In *Software Language Engineering - 3rd International Conference*, SLE '10, pages 42–61, 2010.

[20] A. Johnstone, E. Scott, and M. van den Brand. Modular Grammar Specification. *Sci. Comput. Prog.*, 87:23–43, 2014.

[21] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 805–824, 2011.

[22] L. C. Kats and E. Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 444–463. ACM, 2010.

[23] L. C. Kats, E. Visser, and G. Wachsmuth. Pure and Declarative Syntax Definition: Paradise Lost and Regained. In *Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 918–932. ACM, 2010.

[24] P. Klint, T. van der Storm, and J. Vinju. RASCAL: a Domain Specific Language for Source Code Analysis and Manipulation. SCAM'09. IEEE, 2009.

[25] D. E. Knuth. On the Translation of Languages from Left to Right. *Information and control*, 8(6):607–639, 1965.

[26] P. J. Landin. The Next 700 Programming Languages. *Commun. ACM*, 9(3):157–166, Mar. 1966.

[27] P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns. Attributed Translations. *J. Comput. Syst. Sci.*, 9(3):279–307, 1974.

[28] S. Marlow. Haskell 2010 language report, 2010.

[29] A. Melnikov. Collected Extensions to IMAP4 ABNF, 2006.

[30] Microsoft Corp. C# Language Specification 5.0. 2013.

[31] T. Parr, S. Harwell, and K. Fisher. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. In *Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, pages 579–598. ACM, 2014.

[32] D. J. Salomon and G. V. Cormack. Scannerless NSLR(1) Parsing of Programming Languages. In *Programming Language Design and Implementation*, PLDI '89, pages 170–178, 1989.

[33] E. Scott and A. Johnstone. GLL Parse-tree Generation. *Science of Computer Programming*, 78(10):1828–1844, Oct. 2013.

[34] E. Scott, A. Johnstone, and R. Economopoulos. BRNGLR: A Cubic Tomita-style GLR Parsing Algorithm. *Acta informatica*, 44(6):427–461, 2007.

[35] M. Tomita. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, USA, 1985. ISBN 0898382025.

[36] M. G. J. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In *Compiler Construction*, CC '02, pages 143–158. Springer, 2002.

[37] E. R. Van Wyk and A. C. Schwerdfeger. Context-aware Scanning for Parsing Extensible Languages. GPCE '07, pages 63–72. ACM, 2007.

[38] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[39] D. A. Watt. Rule Splitting and Attribute-directed Parsing. In *Semantics-Directed Compiler Generation*, pages 363–392. 1980.

[40] W. A. Woods. Transition Network Grammars for Natural Language Analysis. *Commun. ACM*, 13(10):591–606, 1970.

A. GLL Parsing

In this section we first describe GLL parsing, SPPF construction and GSS. Then, we define the semantics of GLL parsing over ATN grammars as a transition relation.

A.1 SPPF

It is known that any parsing algorithm that constructs Tomita-style SPPF is of unbounded polynomial complexity [17]. To achieve parsing in cubic time and space, GLL uses a *binarized* SPPF [33] format, which has additional *intermediate* nodes. Intermediate nodes allow grouping of the symbols of a rule in a left-associative manner, thus allowing the parser to always carry a single node at each time, instead of a list of nodes. This is the key in preserving the cubic bound. The use of intermediate nodes effectively achieves the same as restricting a grammar to have rules of length at most two, but without requiring rewriting the original grammar, and transforming back the resulting derivation trees to the ones of the original grammar.

Definition 1. A binarized SPPF is a compact representation of a parse forest that has the following types of nodes.

- *nonterminal* nodes of the form (A, i, j) where A is a nonterminal, and i and j are the left and right extents;
- *terminal* nodes of the form (t, i, j) where t is a terminal, and i and j are the left and right extents;
- *packed* nodes of the form (L, k) where L is a grammar slot and k is the pivot of the node; and

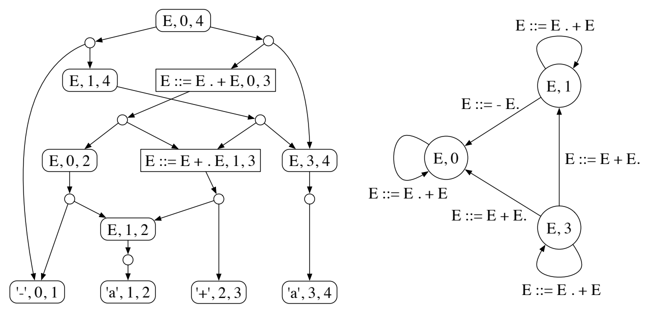


Figure 20. SPPF (left) and GSS (right) for the input `-a+a`.

- *intermediate* nodes of the form (L, i, j) where L is the grammar slot, and i and j are the left and right extends.

The left and right extents of a node represent the substring in the input, associated with the node. As GLL parsing is context-free, nodes with the same label, the same left and the same right extents can be shared. Nonterminal and intermediate nodes have packed nodes as their children. Packed nodes represent a derivation, and can have at most two children, which are non-packed nodes. If a non-packed node is ambiguous, it will have more than one packed node. The pivot of a packed node is the right extent of its left child, and is used to distinguish between packed nodes under a non-packed node.

The binarized SPPF resulting from parsing the input string `-a+a` with the grammar $E ::= -E \mid E + E \mid a$ is shown in Figure 20 (left), where packed nodes are depicted with small circles. For a better visualization, we have omitted the labels of packed nodes. The input is ambiguous and has the following two derivations: `(-(a+a))` or `((-a)+a)`. This can be observed by the presence of two packed nodes under the root node. The left and right packed nodes under the root node correspond to the first and second alternatives, respectively.

SPPF construction is delegated to two functions **nodeT** and **nodeP**. The **nodeT**(t, i, j) function takes terminal t, and two integer values i and j (left and right extents) and returns an existing node with these properties, otherwise a new node. **nodeP**(L, w, z) takes a grammar slot L, and two non-packed nodes w and z. **nodeP** returns an existing non-packed node labeled L with two children w and z. If no such node exists, then a non-packed node labeled L will be created, and w and z are connected to the newly created non-packed node via a packed node. The details of GLL parse tree construction is discussed in [33], and implementation techniques for efficient sharing of nodes are presented in [2, 19].

A.2 GSS

At the core of GLL parsing is the Graph-Structured Stack data structure. We use a variation of GLL parsing that uses a more efficient GSS [2].

$$\boxed{(\mathcal{R},\mathcal{U},\mathcal{G},\mathcal{P}) \Rightarrow (\mathcal{R}',\mathcal{U}',\mathcal{G}',\mathcal{P}')}$$

$$\text{Eps}\ \frac{\begin{array}{c} p \xrightarrow{\epsilon} q \\ n = \textbf{nodeP}(q, w, \textbf{nodeT}(\epsilon, i, i)) \end{array}}{(\{(p,i,u,w)\}\cup\mathcal{R},\mathcal{U},\mathcal{G},\mathcal{P}) \Rightarrow (\mathcal{R}\cup\{(q,i,u,n)\},\mathcal{U},\mathcal{G},\mathcal{P})}$$

$$\text{Term-1}\ \frac{\begin{array}{c} p \xrightarrow{t} q \quad I[i] = t \\ n = \textbf{nodeP}(q, w, \textbf{nodeT}(t, i, i{+}1)) \end{array}}{(\{(p,i,u,w)\}\cup\mathcal{R},\mathcal{U},\mathcal{G},\mathcal{P}) \Rightarrow (\mathcal{R}\cup\{(q,i{+}1,u,n)\},\mathcal{U},\mathcal{G},\mathcal{P})}$$

$$\text{Term-2}\ \frac{p \xrightarrow{t} q \quad I[i] \neq t}{(\{(p,i,u,w)\}\cup\mathcal{R},\mathcal{U},\mathcal{G},\mathcal{P}) \Rightarrow (\mathcal{R},\mathcal{U},\mathcal{G},\mathcal{P})}$$

$$\text{Call-1}\ \frac{\begin{array}{c} p \xrightarrow{A} q \quad v{=}(A,i) \in \mathcal{N}(\mathcal{G}) \\ \mathcal{D} = \{d \mid (v,y) \in \mathcal{P}, d{=}(q, \text{rext}(y), u, \textbf{nodeP}(q,w,y)), \\ d \notin \mathcal{U}\} \end{array}}{\begin{array}{c} (\{(p,i,u,w)\}\cup\mathcal{R},\mathcal{U},\mathcal{G},\mathcal{P}) \Rightarrow (\mathcal{R}\cup\mathcal{D},\mathcal{U}\cup\mathcal{D}, \\ \mathcal{G}\cup\{(v,q,w,u)\},\mathcal{P}) \end{array}}$$

$$\text{Call-2}\ \frac{\begin{array}{c} p \xrightarrow{A} q \quad v{=}(A,i) \notin \mathcal{N}(\mathcal{G}) \\ \mathcal{D} = \{(s,i,v,\$) \mid s \in S(A)\} \end{array}}{(\{(p,i,u,w)\}\cup\mathcal{R},\mathcal{U},\mathcal{G},\mathcal{P}) \Rightarrow (\mathcal{R}\cup\mathcal{D},\mathcal{U},\mathcal{G}\cup\{(v,q,w,u)\},\mathcal{P})}$$

$$\text{Ret}\ \frac{\begin{array}{c} p \in F \\ \mathcal{D} = \{d \mid (u,q,y,v) \in \mathcal{G}, d{=}(q,i,v,\textbf{nodeP}(q,y,w)), d \notin \mathcal{U}\} \end{array}}{(\{(p,i,u,w)\}\cup\mathcal{R},\mathcal{U},\mathcal{G},\mathcal{P}) \Rightarrow (\mathcal{R}\cup\mathcal{D},\mathcal{U}\cup\mathcal{D},\mathcal{G},\mathcal{P}\cup\{(u,n)\})}$$

Figure 21. GLL parsing over ATN grammars.

Definition 2. A Graph-Structured Stack (GSS) in GLL parsing is a directed graph where

- nodes are of the form (A, i), where A is a nonterminal and i is an input position; and
- edges are of the form (u, L, w, v), where u and v are GSS nodes, L is a grammar slot, and w is an SPPF node recorded on the edge.

GSS was originally developed by Tomita [35] for GLR parsing to merge different LR stacks. Although GLL parsing uses the same term, there are two main differences between GSS in GLL parsing and GLR. First, in GLL parsing GSS represents function calls in recursive-descent parsing, similar to memoization of functions in functional programming, and therefore has the input position at which the nonterminal is called. Second, in GLL parsing GSS allows cycles in the graph that solve the problem of left-recursion in recursive-descent parsing.

The GSS resulting from parsing `-a+a` using the grammar $E ::= -E \mid E + E \mid a$ is shown in Figure 20 (right). As can be seen there is a cycle on all nodes, as they represent the left recursive calls to E at different input positions. In case of indirect left recursion, there will be a cycle in the GSS involving multiple nodes.

A.3 GLL Parsing over ATN Grammars

In this section, we define GLL parsing over ATN grammars as a transition relation. In contrast to the imperative style used in [2, 33], we use the declarative rules of Figure 21. Such GLL formulation is concise and easy to extend to support data-dependent grammars. The rules in Figure 21 use notation similar to one in [2, 33].

The unit of work of a GLL parser is a descriptor. A descriptor is of the form (p, i, u, w), where p is an ATN state representing a grammar slot, u is a GSS node, i is an input position, and w is an SPPF (non-packed) node. A GLL parser maintains a set $\mathcal{U}$ that holds descriptors created during parsing and is used to eliminate duplicate descriptors. In addition to $\mathcal{U}$, a set $\mathcal{R}$ is used to hold pending descriptors that are to be processed. Note that GLL parsing does not impose any order in which the descriptors in $\mathcal{R}$ are processed. Figure 21 defines the semantics of GLL parsing over ATN grammars as a transition relation on configuration $(\mathcal{R},\mathcal{U},\mathcal{G},\mathcal{P})$, where $\mathcal{G}$ represents GSS (a set of GSS edges), such that $\mathcal{N}(\mathcal{G})$ gives a set of GSS nodes, and $\mathcal{P}$ is a set of parsing results that are associated with GSS nodes, i.e., a set of elements of the form (u, w).

During parsing a descriptor is selected and removed from $\mathcal{R}$, represented as $\{(p, i, u, w)\} \cup \mathcal{R}$, and given the rules, a deterministic choice is made based on the next transition in the ATN. The first three rules of Figure 21 are straightforward. An ϵ transition creates an ϵ-node (via call to **nodeT**) and intermediate node[4] (via call to **nodeP**), and adds a descriptor for the next grammar slot. The terminal rules (Term-1 and Term-2) try to match terminal t at the current input position, where I is an array representing the input string. If there is a match (Term-1), a terminal node (via **nodeT**) and intermediate node (via **nodeP**) are created, and a descriptor for the next grammar slot is added. If there is no match (Term-2), no descriptor is added.

Call-1 and Call-2 correspond to nonterminal transitions $\xrightarrow{A}$. Similar to calling a memoized function, a GLL parser first checks if a GSS node (A, i) exists. If such a node exists (Call-1), the parsing results associated with this GSS node are reused. These results are retrieved from $\mathcal{P}$, and for each result, nonterminal node y, a descriptor d is created (rext returns the right extent of y), and if the same descriptor has not been processed before ($d \notin \mathcal{U}$), it is added to $\mathcal{R}$. If the GSS node does not exist (Call-2), the call to the nonterminal is made, i.e., for each start state of the nonterminal ($s \in S(A)$), a descriptor is added. Both Call-1 and Call-2 add a new GSS edge to $\mathcal{G}$.

Finally, Ret corresponds to a final grammar slot (final states in ATNs) in which the parser returns from the current nonterminal call. First, the tuple with the current SPPF node and the current GSS node is added to $\mathcal{P}$. Second, for each outgoing GSS edge of the current GSS node, a descriptor is created and, if the same descriptor has not been processed before ($d \notin \mathcal{U}$), it is added to $\mathcal{R}$.

[4] In fact, when the next state is an end state, **nodeP** creates a nonterminal node, instead of an intermediate node. However, in the current discussion, this is not essential, therefore, we always refer to the result of **nodeP** as an intermediate node.