Extensible syntax

Erik Joseph Silkensen
University of Colorado Boulder

Follow this and additional works at: https://scholar.colorado.edu/csci_ugrad

Recommended Citation
https://scholar.colorado.edu/csci_ugrad/39
Extensible Syntax

by

Erik Silkensen

A thesis submitted to the Faculty of the
University of Colorado in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Computer Science

Department of Computer Science

2011
This thesis entitled:
Extensible Syntax
written by Erik Silkensen
has been approved for the Department of Computer Science

Jeremy Siek

Prof. Bor-Yuh Evan Chang

Prof. Sriram Sankaranarayanan

Date ________________

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.
Abstract

Domain-specific embedded languages (DSELs) leverage underlying general-purpose languages to provide high productivity for programmers in many domains, such as computer systems, linear algebra, physics, and other sciences. Writing DSELs in the current generation of general-purpose languages requires a host of tricks, for example, using templates in C++, macros in Scheme, and type classes in Haskell. However, the resulting DSELs are leaky abstractions: the syntax is not quite right, and compile-time error messages expose the internals of the DSEL. Alternatively, domain experts can implement their language by hand, or use legacy tools such as YACC, wherein further pitfalls await in the form of shift-reduce and reduce-reduce conflicts, not to mention language interoperability issues.

In this thesis, we present the design and implementation of a system that supports the straightforward definition, composition, and parsing of concrete syntax for DSELs. Specifically, we present a system that uses chart parsing technology and type-based disambiguation to provide a user-friendly method for specifying and parsing the syntax of DSELs.
1 Introduction

Domain-specific embedded languages (DSELs) leverage underlying general-purpose languages to provide high productivity for programmers in many domains, such as computer systems, physics, linear algebra, and other sciences. DSELs offer a nice separation of concerns: programming language and compiler experts build the general-purpose languages, and domain experts build the DSELs (and not the general-purpose features). Writing DSELs in the current generation of general-purpose languages requires a host of tricks, for example, using templates in C++, macros in Scheme, and type classes in Haskell. However, the resulting DSELs are leaky abstractions: the syntax is not quite right, and compile-time error messages expose the internals of the DSEL. Alternatively, domain experts can implement their language by hand, or use legacy tools such as YACC, wherein further pitfalls await in the form of shift-reduce and reduce-reduce conflicts, not to mention language interoperability issues.

In this thesis, we present the design and implementation of a system that supports the straightforward definition, composition, and parsing of concrete syntax for DSELs. Grimm [13] identifies three properties beneficial to a practical solution for such a system: (1) the employed syntax formalism should be closed under language union in order to enable modularity, that is, programmers can design languages $A$ and $B$ in isolation and then combine them without modification; (2) the module system should provide encapsulation of related productions into syntactic units and support modification and composition of existing units; and (3) for convenience, the system should allow the expression of syntax as code. We agree with Grimm, and add the further requirement that the system must decide how it will deal with ambiguity that arises when defining and extending languages. With these properties in mind, we present preliminary work on a system that uses chart parsing technology and type-based disambiguation to provide a user-friendly method for specifying and parsing the syntax of DSELs.
As a motivating example, we consider the input file in Figure 1, which shows how a user might use our system to define a language of matrix algebra expressions. The syntax of the embedded language is described using a context-free grammar (CFG); the grammar `MatrixAlgebra` and curly-brace notation enable naming, encapsulation, and composition of related productions (e.g., grammars may be nested inside `{···}`); and the `declare A:Matrix;` statement shows how typed variables are declared. We use these types to provide type-based disambiguation. For example, the fully-parenthesized addition of the $n$ terms above is exponentially ambiguous: if each $A$ can be either a `Scalar`, `Vector`, or `Matrix`, then there are $2^n + 1$ possible interpretations. By syntactically declaring the type of $A$ to be `Matrix`, we can remove the ambiguity during parsing.

In the following sections, we expand on the ideas introduced in the previous example. We begin with a summary of background information concerning the theoretical foundations of grammars and parsing, and then discuss some of the numerous related work on extensible syntax. Next, we describe our implementation in detail, including the syntax and semantics of the input language, our strategy for providing type-based disambiguation, and the parsing algorithm tying it all together. Finally, we evaluate the performance of our system, discuss its limitations, and suggest ideas for future innovations.
2 Background Information and Related Work

In this section, we give a brief account of the theoretical foundations of grammars and parsing, and then summarize some of the existing approaches to extensible syntax, considering how they relate to our work.

2.1 Background Information

Parsing is the process of analyzing a sequence of tokens to identify its structure with respect to some syntax formalism; many parsing algorithms for programming languages are created with reference to context-free grammars (CFGs), written in the familiar Backus-Naur Form (BNF) [3] notation. Following Sipser [21], a CFG $G$ is formally defined as a 4-tuple $(V, \Sigma, R, S)$, where

- $V$ is a finite set of variables, called nonterminals;
- $\Sigma$ is a finite set of basic symbols, called terminals;
- $R$ is a finite set of rules that map nonterminals to any combination of nonterminals and terminals, written $A \rightarrow \alpha$;
- and $S \in V$ is the start symbol for the grammar.

If $A \rightarrow \gamma \in R$, then the sequence $\alpha A \beta$ yields the sequence $\alpha \gamma \beta$, written $\alpha A \beta \Rightarrow \alpha \gamma \beta$, and $A$ derives $\gamma$. The language generated by a grammar $G$ is $L(G) = \{ w \in \Sigma^* \mid S \Rightarrow^* w \}$, where $S \Rightarrow^* w$ means $S$ derives $w$ in zero or more steps, i.e., there is a sequence $u_1, u_2, \ldots, u_k$ for $k \geq 0$ such that $S \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow w$. The output of a parsing algorithm for a successful parse of $w$ given a grammar $G$ is usually a structural representation, or parse tree, of the derivation $S \Rightarrow^* w$. Context-free languages are closed under union: given CFGs $A$ and $B$, the union $L(A) \cup L(B)$ is also a context-free language.
Some of the popular algorithms for parsing full general-purpose programming languages, such as LL($k$) [19], LR($k$) [18], and LALR($k$) [10], are restricted to subsets of CFGs, which increases the difficulty of writing grammars and limits their compositional properties—LL($k$), LR($k$), and LALR($k$) grammars are not closed under union. Generalized LR (GLR) [23] parsers sacrifice the linear-time complexity of the LL and LR algorithms, extending the LR algorithm to allow the parsing of arbitrary, potentially ambiguous CFGs in $O(n^3)$ time (where $n$ is the length of the parsed string). Chart parsing, an approach due to Martin Kay [17], uses the dynamic programming technique to also allow the parsing of unrestricted CFGs in $O(n^3)$ time. The well-known Cocke-Younger-Kasami (CYK) [16] and Earley [11] algorithms may be expressed in terms of the chart parsing framework; CYK and Earley use bottom-up and top-down approaches, respectively.

2.2 Related Work

There have been numerous approaches to the problem of providing extensible syntax for programming languages. In this section, we summarize some of them and discuss how they relate to our work.

MetaBorg [6, 5] is a method of embedding DSELs in general-purpose languages based on the Stratego/XT toolset [8]. It relies on the syntax definition framework SDF [15], which allows definitions of arbitrary CFGs, and it uses the scannerless GLR parsing technology. MetaBorg features special purpose disambiguation facilities such as separate priority definitions, follow restrictions, and other disambiguation filters [4]. Bravenboer et al. [7] have also explored type-based disambiguation as a separate phase after parsing and assimilation, i.e., a type checker for the meta language that operates on an abstract syntax forest; this work offers a “jumping-off” point for our research into applying type-based disambiguation during the parsing phase.

Aasa et al. [1] introduce syntax for concrete data types, a syntactic construction cor-
responding to CFGs, in ML. Their system allows the embedding of arbitrary context-free languages using fixed symbols for quotation and anti-quotation. They generalize the Earley algorithm, integrating the parsing and type checking phases for ML.

Several approaches use Ford’s [12] recognition-based Parsing Expression Grammar (PEG) formalism. PEGs are stylistically similar to CFGs; however, PEGs avoid ambiguity by introducing a prioritized choice operator for rule alternatives, and do not allow left-recursive rules. Rats! [13] is a tool for extensible parsing based on PEGs that uses memoizing recursive-descent “Packrat” [14] parsing technology. Allen et al. [2] use Rats! in their modular hygienic macro system for the Fortress programming language. Katahdin [20] is a platform for specifying and implementing programming languages where the syntax and semantics are mutable at runtime. It uses a modified form of PEGs where rules are ordered based on a longest match strategy instead of by the prioritized choice operator.

Cardelli et al. [9] develop extensible grammars as a syntax-definition formalism for incremental extensions and restrictions to programming languages. Like Scheme’s hygienic macros, their system preserves lexical scoping rules. Their implementation is based on LL(1) parsing.

Our work is most related to the MetaBorg project. We are currently using a bidirectional variant of chart parsing, and are investigating whether it may offer advantages over the other GLR, CYK, and Earley algorithms by allowing us to mix bottom-up and top-down approaches, or to work outwards from “islands” in the input if necessary. While we share many of the same goals as the PEG and LL based approaches to extensible syntax, we think it may be more beneficial from a user’s perspective to support unrestricted CFGs.
Program → Body* Extension Body* | Body
Body → Declaration | Text+
Declaration → “declare” Name “::=” Type “;” Program
Extension → Grammar “{” Program “}”
Grammar → “grammar” Name Rule+
   Rule → Name “::=” Expression “;”
Expression → Term+ | Term+ “|” Expression
   Term → Name | String | RegExp
Type → Name

Figure 2: The syntax of the input language.

3 A System for Extensible Syntax

In this section, we present our system for extensible syntax. We describe the syntax of the input language and the details parsing algorithm we use to process input files, and show we use typed variable declarations to disambiguate during parsing.

3.1 Syntax and Semantics of the Input Language

The syntax of the input language currently consists of allowing the definition, composition, and use of CFGs, written in BNF-style notation. Figure 2 shows the formal syntax of the language. A grammar is defined with the syntax “grammar Name” followed by a sequence of rules. In addition to nonterminals and literal strings, the right-hand sides of rules can include regular expressions, written

\[
RegExp \rightarrow “#rx” String \mid “#px” String.
\]

We use the Racket programming language to provide regular expressions; the “#px” syntax supports POSIX character classes, in addition to standard regular expressions (“#rx”).
3.1.1 Lexical Syntax and Reserved Words

Before parsing, we use whitespace and the characters \langle . ; : :: = \rightarrow ( ) [ ] \{ \}\rangle, in addition to literal strings and regular expressions as described above, to separate tokens of input files. The words grammar and declare are reserved and cannot be used in user-defined languages.

3.1.2 Composition of Languages

Figure 3 shows an example of an input file that defines a small language for lambda calculus expressions. It extends the language twice, incrementally adding syntax for simply typed abstractions, and for System F style abstractions and applications of types. Each grammar and its curly-brace enclosed body is parsed as an Extension; the Text body of an Extension is parsed with the union of all grammars from enclosing Extensions.

3.1.3 Type-Based Disambiguation

We currently implement type-based disambiguation by associating typed variables with new grammar rules: the semantic action of a typed variable declaration “declare Name : Type” is simply to add the rule Type \rightarrow Name to the grammar. For example, the program in Figure 1 uses the declaration “declare A:Matrix;” to syntactically give the variable A the type Matrix. Here the rule Matrix \rightarrow “A” is added to the grammar, which allows the parsing algorithm to avoid what would otherwise be an exponentially ambiguous grammar (e.g., with rules deriving A as either Matrix, Vector, or Scalar). By inserting the new rules into the current grammar, typed variables follow lexical scoping rules.
grammar Lambda

S ::= Stmt S | Stmt;
Stmt ::= Assign ";" | Term ";";
Assign ::= Var "=" Term;
Term ::= Var | Abs | App;
Var ::= #rx"[a-z]+";
Abs ::= "lambda" Var "." Term;
App ::= "(" Term Term ")";
{
    id = lambda x. x;
    tru = lambda t. lambda f. t;
}

grammar TypedLambda

Abs ::= "lambda" Var ":" Type "." Term;
Type ::= Type "->" Type | Base;
Base ::= #rx"[A-Z][a-zA-Z]*";
{
    lambda f:Int->Int. (f x);
}

grammar SystemF

Term ::= TypeAbs | TypeApp;
TypeAbs ::= "lambda" Type "." Term;
TypeApp ::= "(" Term "[" Type "]" ")";
{
    double = lambda X. lambda f:X->X. lambda a:X. (f (f a));
    doubleNat = (double [Nat]);
    quadruple = lambda X. ((double [X->X]) doubleNat);
}
(f x);
}

two = lambda s. lambda z. (s (s z));
}

Figure 3: An example of an input file defining and extending a simple lambda calculus.
3.2 The Chart Parsing Algorithm

Here we describe the traditional chart parsing algorithm and data structures, and then introduce a variant of the island parsing and bidirectional extension. We also give a short, concrete example to illustrate the steps of the algorithm.

3.2.1 Traditional Chart Parsing

Chart parsing algorithms work by progressively filling in a chart data structure, which functions as well-formed sub-string table: partial parses of the input are stored in the chart, or memoized, so that they don’t need to be parsed again later. The chart may be represented by a directed graph, where the vertices of the graph are the input tokens and the edges connecting them are labeled with dotted rules from the grammar; Figure 5 shows an example of the directed graph representation of a chart. Formally, an active, or incomplete edge is written \( \langle A \rightarrow \alpha \cdot \beta, i, j \rangle \), where \( A \rightarrow \alpha \beta \) is a rule in the grammar, and the input spanning from \( i \) to \( j \) has been so far parsed as \( \alpha \). An inactive, or completed edge is written \( \langle A \rightarrow \gamma., i, j \rangle \), where \( A \rightarrow \gamma \) is a rule in the grammar, and the input spanning from \( i \) to \( j \) has been parsed as \( \gamma \). The completed parse \( S \Rightarrow w \) of an input string of length \( |w| = n \) is represented by all edges \( \langle S \rightarrow \gamma., 0, n \rangle \); i.e., when the parse is ambiguous, there will be more than one completed edge for \( S \) from 0 to \( n \).

In general, chart parsing consists of iteratively removing edges from a queue, called the agenda, processing them, and entering them into the chart until the agenda is empty:

```plaintext
function CHART-PARSE(tokens, G = (V, \Sigma, R, S)) returns chart
   chart \leftarrow \text{INITIAL-CHART}(tokens)
   agenda \leftarrow \text{INITIAL-AGENDA}(tokens, G)
   repeat
      edge \leftarrow \text{DEQUEUE}(agenda)
      PROCESS-EDGE(edge, chart, agenda)
   until \text{EMPTY}(agenda)
```

9
The initial definitions of the chart and agenda determine whether the algorithm begins top-down or bottom-up. For example, if the chart is empty and the agenda is initialized with the input tokens, then processing begins bottom-up from the tokens. Similarly, if the chart is initialized with the input tokens and the agenda is instead initialized with incomplete edges for $S$, then processing begins top-down from $S$; a common trick to get started is to add $\langle S' \rightarrow S, 0, 0 \rangle$ to the initial agenda.

The PROCESS-EDGE procedure is responsible for inserting the current edge into the chart, and then adding all edges to the agenda that can be deduced via the application of a set of rules, or deductive schema, directing the chart parsing algorithm. For example, CYK and Earley style algorithms can be implemented in terms of the general chart parsing framework given above by defining different sets of rules to apply or combine in different ways in PROCESS-EDGE. The algorithm requires that no duplicate edges are added to the chart or agenda.

Chart parsing algorithms usually involve at least two basic types of rules: (1) rules (sometimes known as “scanner” or “completer” rules) based on what Kay calls the fundamental rule of chart parsing, which join together an active edge spanning from $i$ to $j$ with an inactive edge already in the chart from $j$ to $k$, to deduce a new edge from $i$ to $k$; and (2) predictive rules that deduce new edges by combining a completed edge in the chart with a rule from the grammar, e.g., rules for top-down and bottom-up prediction. The order in which these rules add and remove edges from the agenda determines the style of search through the possible edges. For example, a depth-first or breadth-first approach is possible by appending new edges to the beginning or ending of the agenda, respectively. The agenda could also be implemented as a priority queue, enabling a best-first approach.
\[
\begin{align*}
\frac{e \in a}{\langle c, a \rangle \rightarrow \langle c \cup \{e\}, a\setminus\{e\} \rangle} & \quad \text{(INSERT)} \\
\frac{\langle A \to \alpha . B \beta, i, j \rangle \in c \quad \langle B \to \gamma, j, k \rangle \in c}{\langle c, a \rangle \rightarrow \langle c, a \cup \{\langle A \to \alpha B . \beta, i, k \rangle\} \rangle} & \quad \text{(FUND)} \\
\frac{\langle A \to \alpha . B \beta, i, j \rangle \in c \quad \langle B \to \gamma \in R \rangle}{\langle c, a \rangle \rightarrow \langle c, a \cup \{\langle B \to \gamma, j, j \rangle\} \rangle} & \quad \text{(TD-PRED)} \\
\frac{A \to B \beta \in R \quad \langle B \to \gamma, i, j \rangle \in c}{\langle c, a \rangle \rightarrow \langle c, a \cup \{\langle A \to B . \beta, i, j \rangle\} \rangle} & \quad \text{(BU-PRED)}
\end{align*}
\]

Figure 4: Operational semantics of traditional chart parsing.

### 3.2.2 Operational Semantics of Chart Parsing

We can formalize chart parsing algorithms by defining an abstract machine that takes one configuration, a pair of the chart and agenda \(\langle c, a \rangle\), and steps to a new configuration \(\langle c', a' \rangle\), representing the application of one rule, such as the fundamental rule described above. This step relation is written as:

\[
\langle c, a \rangle \rightarrow \langle c', a' \rangle
\]

A terminal configuration for a chart parsing algorithm is one with an empty agenda, \(\langle c', \emptyset \rangle\), and one that can no longer take a step by any of the rules. We write \(\langle c, a \rangle \rightarrow^* c'\) if there is a sequence of zero or more steps that produce a terminal configuration \(\langle c', \emptyset \rangle\).

Figure 4 shows the operational semantics of traditional chart parsing. The INSERT rule is responsible for removing an edge from the agenda and inserting it into the chart; FUND is Kay’s fundamental rule of chart parsing; TD-PRED is a rule for top-down prediction; and BU-PRED is a rule for bottom-up prediction.
3.2.3 Island Parsing and Bidirectional Charts

Island parsing [22] is an extension of traditional chart parsing that has its origin in speech and language processing. It allows parsing to proceed alternately from left-to-right or right-to-left. The motivation for this extension is that it may be advantageous for parsing to work outwards from “islands” instead of proceeding like normal from the leftmost position, from left-to-right.

In island parsing, edges are bidirectional, that is, the edges are double-dotted. Formally, an active edge may now have remaining tokens on both the left and right sides of its currently parsed input, and is written \( \langle A \rightarrow \alpha . \beta . \delta , i, j \rangle \). Similarly, an inactive edge is written with a second dot \( \langle A \rightarrow . \gamma . , i, j \rangle \). Figure 5 shows an example of a chart including some of the inactive edges for a parse of “\texttt{id = lambda x. x;}” (from the program in Figure 3). In order to handle the new bidirectional edges, the fundamental rule must be split into two separate rules for joining edges together from the left and right:

![Figure 5: A chart showing inactive edges for a parse of “\texttt{id = lambda x. x;}”.

\[ \langle \text{Stmt} \rightarrow . \text{Assign “;” , } 0,7 \rangle \]
\[ \langle \text{Assign} \rightarrow . \text{Var} \_id \text{ “=” Term , } 0,6 \rangle \]
\[ \langle \text{Term} \rightarrow . \text{Abs , } 2,6 \rangle \]
\[ \bullet_0 \text{ id } \bullet_1 = \bullet_2 \text{ lambda } \bullet_3 \text{ x } \bullet_4 \text{ . } \bullet_5 \text{ x } \bullet_6 ; \bullet_7 \]
\[ \langle \text{Abs} \rightarrow . \text{“lambda” Var “;” Term , } 2,6 \rangle \]
\[\langle A \rightarrow \alpha B . \beta . \delta, j, k \rangle \in c\]
\[\langle B \rightarrow . \gamma ., i, j \rangle \in c\]
\[\langle c, a \rangle \rightarrow \langle c, a \cup \{\langle A \rightarrow \alpha . B \beta . \delta, i, k \rangle\}\rangle\]  \hspace{1cm} \text{(L-FUND)}

\[\langle A \rightarrow \alpha . \beta . B \delta, i, j \rangle \in c\]
\[\langle B \rightarrow . \gamma ., j, k \rangle \in c\]
\[\langle c, a \rangle \rightarrow \langle c, a \cup \{\langle A \rightarrow \alpha . \beta B . \delta, i, k \rangle\}\rangle\]  \hspace{1cm} \text{(R-FUND)}

### 3.2.4 Constrained Bottom-up Prediction

We implement a variant of bottom-up island parsing. The initial chart is empty, while the initial agenda contains an edge for each token,

\[\text{agenda}_0 = \{\langle t_i \rightarrow . t_i ., i, i + 1 \rangle | t_i \in \text{tokens}\}.\]

We use the fundamental rule as defined above, but modify the traditional rule for bottom-up prediction as shown below, formally:

\[A \rightarrow u_1 u_2 \cdots u_l \in R\]
\[\langle B \rightarrow . \gamma ., i, j \rangle \in c\]
\[B \in V \vee \exists C \in V. C \Rightarrow B\]
\[u_k \Rightarrow B, 1 \leq k \leq j - i\]
\[\alpha = u_1 u_2 \cdots u_{k-1}\]
\[\delta = u_{k+1} u_{k+2} \cdots u_l\]
\[\langle c, a \rangle \rightarrow \langle c, a \cup \{\langle A \rightarrow \alpha . B . \delta, i, j \rangle\}\rangle\]  \hspace{1cm} \text{(BU-PRED*)}

In the third line of the BU-PRED* rule, we add a constraint to the type of completed edge that can deduce new edges: for a completed rule of the form \(\langle B \rightarrow . \gamma ., i, j \rangle\), we say that \(B\) must either be a nonterminal \((B \in V)\), or that it must be derived in one step by another rule in the grammar \((\exists C \in V. C \Rightarrow B)\). Intuitively, the goal of this constraint is to avoid the prediction of new edges from edges for literal strings that are part of a rule’s right-hand side. Instead, we would like to build up the parse using nonterminals around these strings, and then connect them with an application of the fundamental rule. For example, considering the
matrix algebra program in Figure 1, an edge such as \(\langle\text{"(" } \rightarrow \text{ "(", 0,1}\rangle\) would normally lead to the prediction of edges for Scalar, Vector, and Matrix; however, with the constraint described above, prediction is delayed until the edge for \("A\) (because Matrix \(\rightarrow \text{"A}\)), and only Matrix is predicted.

We also require in the fourth line that \(k \leq j - 1\). Our heuristic here is that if \(B\) spans a distance smaller than the potentially remaining work to the left, we should work from left-to-right first, instead of right-to-left. As a small example, suppose we were parsing a single \(A\) with the matrix algebra grammar. With this restriction, we can avoid adding the edge \(\langle\text{Matrix }\rightarrow\text{ Matrix "+", Matrix ,, 0,1}\rangle\). In Section 4, we present empirical results showing the effect of these changes on the number of edges generated by the algorithm.

### 3.2.5 An Example

Here we walk through a brief, concrete example of parsing the expression \("(A + A)\) from the program in Figure 1.

The first step is to initialize the agenda with edges for each input token:

<table>
<thead>
<tr>
<th>Agenda</th>
<th>Chart</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle\text{&quot;(&quot; } \rightarrow \text{ &quot;(&quot;, 0,1}\rangle)</td>
<td></td>
</tr>
<tr>
<td>(\langle\text{&quot;A&quot; } \rightarrow \text{ &quot;A&quot;, 1,2}\rangle)</td>
<td>Chart</td>
</tr>
<tr>
<td>(\langle\text{&quot;+&quot; } \rightarrow \text{ &quot;+&quot;, 2,3}\rangle)</td>
<td></td>
</tr>
<tr>
<td>(\langle\text{&quot;A&quot; } \rightarrow \text{ &quot;A&quot;, 3,4}\rangle)</td>
<td></td>
</tr>
<tr>
<td>(\langle\text{&quot;)&quot; } \rightarrow \text{ &quot;, 4,5}\rangle)</td>
<td></td>
</tr>
</tbody>
</table>

Next, we iteratively apply the INSERT, L-FUND, R-FUND, and BU-PRED\(^*\) rules. For the remaining configuration diagrams, we don’t display edges added to the chart that aren’t part of the successful parse. Instead of showing each edge being added to the agenda and then moved over to the chart via the INSERT rule, we write the edge directly in the chart, labeled with the rule that put it in the agenda.

After adding \("(" and \("A\) to the chart, we can invoke BU-PRED\(^*\) to add an edge for Matrix, and then once again to add an edge for matrix addition:
After adding "+" to the chart, we apply the fundamental rule R-FUND to build the addition edge one step to the right:

<table>
<thead>
<tr>
<th>Agenda</th>
<th>Chart</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&quot;+&quot; → &quot;+&quot;, 2, 3)</td>
<td>⟨Matrix → . Matrix .&quot;+&quot; Matrix , 1, 2⟩ (BU-PRED*)</td>
</tr>
<tr>
<td>(&quot;A&quot; → &quot;A&quot;, 3, 4)</td>
<td>⟨Matrix → . &quot;A&quot;, 1, 2⟩ (BU-PRED*)</td>
</tr>
<tr>
<td>(&quot;)&quot; → &quot;)&quot;, 4, 5)</td>
<td>⟨&quot;A&quot; → . &quot;A&quot;, 1, 2⟩</td>
</tr>
<tr>
<td>(&quot;(&quot; → &quot;)&quot;, 0, 1)</td>
<td>⟨&quot;(&quot; → . &quot;)&quot;, 0, 1⟩</td>
</tr>
</tbody>
</table>

Now we complete the parse of "A + A" by adding the final "A", applying BU-PRED* to it, and then joining it with the incomplete addition edge with R-FUND:

<table>
<thead>
<tr>
<th>Agenda</th>
<th>Chart</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&quot;)&quot; → &quot;)&quot;, 4, 5)</td>
<td>⟨Matrix → . Matrix &quot;+&quot; . Matrix , 1, 4⟩ (R-FUND)</td>
</tr>
<tr>
<td>(&quot;+&quot; → &quot;+&quot;, 2, 3)</td>
<td>⟨Matrix → . &quot;+&quot; . Matrix , 1, 4⟩ (R-FUND)</td>
</tr>
</tbody>
</table>

The only remaining task is to include the parentheses; we can now apply BU-PRED* and then L-FUND to parse the left parenthesis. Finally, we complete the parse by adding ")" to the chart and applying R-FUND one last time:

<table>
<thead>
<tr>
<th>Agenda</th>
<th>Chart</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&quot;(&quot; → . &quot;(&quot; Matrix &quot;)&quot;, 0, 5)</td>
<td>⟨Matrix → . &quot;(&quot; Matrix &quot;)&quot;, 0, 5⟩ (R-FUND)</td>
</tr>
<tr>
<td>(&quot;)&quot; → &quot;)&quot;, 4, 5)</td>
<td>⟨&quot;)&quot; → . &quot;(&quot; , 4, 5⟩ (L-FUND)</td>
</tr>
<tr>
<td>(&quot;+&quot; → &quot;+&quot;, 2, 3)</td>
<td>⟨&quot;+&quot; → . &quot;+&quot;, 2, 3⟩ (BU-PRED*)</td>
</tr>
<tr>
<td>(&quot;A&quot; → . &quot;A&quot;, 1, 2)</td>
<td>⟨&quot;A&quot; → . &quot;A&quot;, 1, 2⟩ (BU-PRED*)</td>
</tr>
<tr>
<td>(&quot;(&quot; → . &quot;(&quot; , 0, 1)</td>
<td>⟨&quot;(&quot; → . &quot;(&quot; , 0, 1⟩</td>
</tr>
</tbody>
</table>

15
Parse time (sec.)

\[ (A_1 + (A_2 + \cdots + (A_{n-1} + A_n) \cdots) ) \]

- MetaBorg, ambiguous
- type-based disambiguation

4 Evaluation

In this section, we evaluate the empirical performance of our system, showing the benefit of adding type-based disambiguation. We also consider the effect of our modified rule for bottom-up prediction on the number of edges generated.

4.1 Performance of the Matrix Algebra Example

Figure 6 shows the performance of the matrix algebra example, with and without type-based disambiguation, for \( n = 1 \) up to 25. For this comparison, we remove the type declaration and adjust the grammar so that any Name matches Scalar, Vector, and Matrix; the program has \( 2^n + 1 \) parses for the adjusted grammar. The parse times reflect what we would expect: even when parsing with the MetaBorg project, the time is \( O(2^n) \) (both axes use logarithmic scales); and the version with type-based disambiguation returns the single parse tree in \( O(n^3) \) time, as expected of chart parsing algorithms. For reference, we collected these times using a 2.16 GHz Intel Core 2 Duo computer with 2 GB RAM.
<table>
<thead>
<tr>
<th>Test</th>
<th>Normal</th>
<th>BU-PRED*</th>
<th>% Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MA_1</td>
<td>15</td>
<td>11</td>
<td>26.67</td>
</tr>
<tr>
<td>MA_5</td>
<td>203</td>
<td>142</td>
<td>30.05</td>
</tr>
<tr>
<td>MA_{10}</td>
<td>438</td>
<td>307</td>
<td>29.91</td>
</tr>
<tr>
<td>MA_{15}</td>
<td>673</td>
<td>472</td>
<td>29.87</td>
</tr>
<tr>
<td>MA_{20}</td>
<td>908</td>
<td>637</td>
<td>29.85</td>
</tr>
<tr>
<td>MA_{25}</td>
<td>1143</td>
<td>802</td>
<td>29.83</td>
</tr>
<tr>
<td>LC</td>
<td>915</td>
<td>673</td>
<td>26.45</td>
</tr>
<tr>
<td>TIL</td>
<td>2877</td>
<td>2075</td>
<td>27.88</td>
</tr>
</tbody>
</table>

Table 1: The effect of constrained BU-PRED* on the number of edges created.

### 4.2 Edge Reduction by BU-PRED*

We measure the effect of our constrained version of BU-PRED* by looking at the number of edges created by the algorithm with and without the constraints. Table 1 shows the number of edges created normally, the number of edges created when using our modified BU-PRED*, and the percent reduction in the number of edges due to BU-PRED* for several tests. The MA_n test is the matrix algebra example with n terms, the LC test is the lambda calculus program from Figure 3, and the TIL test is an example program taken from the Stratego/XT (MetaBorg) documentation.¹ TIL is a Tiny Imperative Language that includes variable declarations, assignments, procedure calls, and control-flow statements, in addition to basic arithmetic and relational expressions. Figure 7 shows the TIL test program, omitting most of its grammar definition. For all of these tests, our constrained BU-PRED* reduces slightly under 30% of the edges generated by the island parsing algorithm.

¹[http://strategoxt.org/Stratego/StrategoDocumentation](http://strategoxt.org/Stratego/StrategoDocumentation)
grammar TIL

Program ::= Stat+;
Stat ::= Declaration | DeclarationTyped
     | Assign | Block | IfThen | IfElse
     | While | For | ProcCall;
Declaration ::= "var" Id ";";
DeclarationTyped ::= "var" Id ":" Type ";";
Type ::= Id;
Assign ::= Id ":=" Exp ";";
Block ::= "begin" Stat+ "end";
IfThen ::= "if" Exp "then" Stat+ "end";
IfElse ::= "if" Exp "then" Stat+ "else" Stat+ "end";
While ::= "while" Exp "do" Stat+ "end";
For ::= "for" Id ":=" Exp "to" Exp "do" Stat+ "end";
True ::= "true";
False ::= "false";
Id ::= #rx"^[A-Za-z][A-Za-z0-9]*$";
Int ::= #rx"^[0-9]+$";
String ::= #rx"[\"][^\"]*[\"]$";
{
  var n;
  n ::= readint();
  var x;
  var fact;
  fact ::= 1;
  for x ::= 1 to n do
    fact ::= x * fact;
  end
  write("factorial of ");
  writeint(n);
  write(" is ");
  writeint(fact);
  write("\n");
}

Figure 7: The Tiny Imperative Language (TIL).
5 Conclusions and Future Work

In this thesis, we have presented preliminary work on a system for extensible syntax. In particular, with this work we hope to provide a user-friendly method for the definition, composition, and parsing of concrete syntax for DSELS. There are many existing approaches to the problem of providing extensible syntax for programming languages, however, some of the most familiar options, such as the metaprogramming features in popular general-purpose languages, or writing parsers by hand or with a parser generator, present the DSEL designer with several difficulties and potential pitfalls. To address this problem, we have developed a system that uses a variation of chart parsing technology to parse languages with extensible syntax, specified with unrestricted CFGs.

In order to support unrestricted CFGs, we must have some way to deal with ambiguity. Our solution is to allow the user to declare syntactically typed variables that the parsing algorithm can use to disambiguate during parsing; our goal is to use type-based disambiguation to make the problem of parsing unambiguous programs with possibly highly ambiguous grammars a manageable one. In this thesis, we considered a small example in the language of matrix algebra expressions, demonstrating how a user might use types to disambiguate their input program given an ambiguous grammar.

There are many opportunities for future innovations. First, we are interested in developing more sophisticated type-based disambiguation in the form of parameterized rules, for example. While we have had some success applying a more constrained version of bottom-up prediction for our parsing algorithm, we need to verify its correctness more rigorously, identifying any restrictions it places on grammars. We will also define and implement syntax for specifying precedence and associativity in grammars. Finally, we are interested in offering a more useful output; for example, generating Typed Racket code instead of only abstract syntax trees.
References


