Christian Plesner Hansen

An Efficient, Dynamically Extensible ELL Parser Library

Master's Thesis • Computer Science • University of Aarhus, Denmark
May 2004

This thesis was set using KOMA-Script and $\LaTeX$
All diagrams were produced using the Dia diagram editor
All graphs were produced using GnuPlot

Abstract

The ability to dynamically extend or configure a parser can be very useful, for instance in the implementation of macro mechanisms or extensible compilers. Yet, practically all current parser technology focuses on parsers as something that is constructed statically, and only allows very limited extensions.

This thesis develops a library that allows dynamic generation of parsers for arbitrary non-left-recursive context-free grammars, specified through EBNF. The generated parsers can be extended dynamically, while they are parsing, yet are shown to parse input at least as efficiently as statically generated parsers. To implement this, two new algorithms are introduced: an algorithm for incremental construction of LL parsers and a practical adaptation of breadth-first LL parsing. A prototype of the library has been implemented in the Java programming language, which has shown that the design is practical and can be implemented efficiently.

The result is a general solution to the problem of allowing syntax extension, which can be applied directly in the implementation of macro systems and other mechanisms that require parsers to be extensible.

Acknowledgments

I would like to thank my advisor Mads Torgersen for great support during the process of writing this thesis. I would also like to thank Stephanie Munck for helping with the LaTeX layout. Finally, I wish to thank Laust Rud Jensen, Jonas Auken and Claus Brabrand valuable comments on drafts of this thesis.

Contents

List of Figures

1. Introduction

This thesis develops a library that allows parsers to be generated, at runtime, for grammars specified by a running program. The generated parsers can be extended dynamically, while they are parsing, yet are shown to parse input at least as efficiently as parsers generated by traditional parser generators.

1.1. Extensible parsing

Parsing is the process of recognizing the grammatical structure of a flat sequence of symbols. A concrete example of parsing is what you are doing right now: reading a flat sequence of letters and punctuation and recognizing them as sentences written according to the grammar of the English language. As humans, parsing is something that comes so natural to us that we hardly even notice doing it.

Because structured written text comes so natural to us, it is very widely used when specifying data to be read by computer programs. Natural languages are unsuitable for such purposes, though, because ambiguities and irregularities make it very hard to parse them automatically. Instead, a simpler form of language is often used that is not too inconvenient for us to use and not too hard for a program to parse: the context-free languages. Context-free languages, defined through context-free grammars, is a successful compromise between what can be conveniently read and written by a human and unambiguously understood by a program.

Parsing, and context-free parsing in particular, is one of the most studied and best understood areas of computer science. It has been an active research topic for more than 40 years, and the most important algorithms, the ones that are predominantly used today, were all discovered at least 20 or 30 years ago. This fact alone can be taken as evidence that anything new developed in this field could not possibly have practical relevance – if it did, why wasn't it done 20 years ago? Whatever the reason, the fact remains that there are situations that are not adequately handled by current parser technology.

Usually, the language understood by a program is fixed throughout the life of the program. While it might be possible to extend the vocabulary of the language, the basic grammar of the language cannot be changed or extended. Correspondingly, the parser used in such programs usually is either written by hand or generated by a parser generator, and compiled together with the program that uses it. The far majority of parser tools and algorithms are targeted at this kind of use.

As processors are becoming faster and faster, CPU cycles are quickly becoming less and less expensive. This means that providing higher levels of abstraction is a good investment, even though it might make the program execute less efficiently, if it helps the programmer to focus on the problem at hand. It also becomes more and more important for a programming language to not only supply a fixed set of abstractions, but to allow the programmer to define new, domain-specific abstractions.

When dealing with a particular domain, having the right notation can be very important. An example of this is mathematical notation: doing serious mathematics in plain English, without any specialized notation, is almost impossible. A key element in the development of mathematics has been the parallel development of an appropriate and suggestive notation. A programming language that allows the programmer to not only develop new abstractions, but also introduce new notation or syntax for those abstractions is a very powerful tool indeed.

This brings us back to the point about current parser technology. If the parser is fixed throughout the life of the program, how can the programmer be allowed to extend the syntax of the language? It may be possible to allow some degree of extensibility, for instance by allowing simple macros or operator overloading, but the extensions must be kept within the limits of what the parser can already handle. Of course, even if the parser were extensible, it is probably healthy to have certain limits for how the language could be extended. But those limitations should certainly not be dictated by the limited extensibility of the parser.

The subject of this thesis is the design and implementation of a library for generating dynamically extensible parsers. Such a library provides the flexibility beyond static parsers that is necessary to allow a program to parse input according to user-defined or user-configured grammars.

1.2. Applications

This section gives three concrete examples where an extensible parser library is useful. Besides demonstrating the usefulness of the library, this section is also meant to give an impression of the problems this library is expected to solve and which functionality the library is expected to provide.

What was described in the first section was my main motivation for working with this subject, but a parser library can be useful for many other purposes than extensible languages. In the first example, the library is used to solve a simple modularity problem.

1.2.1. Modular parser construction

Many database systems use SQL (*structured query language*) to specify queries and updates to a relational database. Often, a system implements some standard subset of the SQL language and extends it with a number of non-standard features specific for that system. This means that there is a great number of SQL dialects that are similar to some degree, but have different extensions. Since the extensions are non-standard, this also means that similar features can have different syntax in the dialects used by different systems.

If a system implements a non-standard feature that is also available in other systems, which syntax should be chosen for the feature[1]? One possibility is to invent yet another syntax, but that would make the system incompatible with the existing systems. That means that the user cannot reuse existing SQL code, which lessens the system's appeal as a replacement for an existing system. Alternatively, the syntax used by one of the other systems could be used. That would make the it compatible with one other system, but incompatible with the others. What is frustrating here is that the limiting factor is not the functionality – the system implements the feature – but a purely syntactic problem: it may not be able to parse the local dialect used to interact with the feature.

What we really want is to make it possible for the system to understand more than one SQL dialect, and allow the exact dialect to be configured by the user. That way, code written in any of the other dialects can be used directly with this system, which makes dealing with the different systems much easier for the users. But if we don't know exactly which dialect the parser will be used with, how can we construct a static parser for the system?

One possibility is to statically construct a parser for each dialect. The problem with that is that the space of SQL dialects is large. All the considerations above were concerned with the syntax of a single non-standard feature. If the system implements three non-standard features and there are three different variants of the syntax of each feature, that totals to 27 different combinations. It might be that only five of these combinations give dialects that are in actual use, but that still means that five slightly different parsers have to be maintained. That seems a waste, especially because once the system has been configured, it is likely to only use one of the dialects.

This is really a modularity problem. A clean solution would be to define a number of modules for each feature, where each module describes one variant of the syntax of that feature. When the system has been configured, and the user has selected the syntax to use for each feature, the corresponding syntax modules can be picked out and used to construct a parser for the chosen dialect. Allowing this is one possible use of the parser library presented here. This way, instead of presenting the user with a list of five choices of dialects, the user can freely combine the modules into any of the 27 possible dialects, and the system will be able to put together the modules and construct an appropriate parser on the fly:

$$\mathsf{activeGrammar} = \mathsf{baseGrammar} \oplus \mathsf{oracleOuterJoinSyntax} \oplus \mathsf{postgresqlLimitOffsetSyntax}$$

This model allows a clean and modular implementation of the parser, and makes the system far more flexible. The library developed here supports this by allowing grammar

[1]This example is inspired by a weblog posting and the following discussion (Waldhoff 2003) related to the Axion database engine.

modules to be specified and combined freely, and allowing parsers to be generated on the fly for the specified grammars.

1.2.2. Library language extension

The previous example used the library's ability to define grammar modules and produce a parser for their combination to solve a modularity problem. The following example demonstrates how the library can be used to implement "selective extension" of a programming language.

When using a large programming language, for instance C++, it is likely that the programmer will only use some subset of the language for a given application. When writing a compiler, for instance, you can easily do without operator overloading, while operator overloading can be very convenient in a matrix algebra library. On the other hand, a matrix algebra library can probably do without function pointers, whereas function pointers are important in an application that uses system signals and interrupts.

How is a language to deal with this? One possibility is to include a construct if there are enough situations where it is useful, even if there are many situations where it is not. That way, the feature is always available, and the programmer is free to decide whether or not to use it for a particular application. An "inclusive" design strategy carries the danger of resulting in a bloated and complex language that is hard to learn and use. Another approach is to keep it simple and only include a feature if it is generally hard to live without. This can potentially give a cleaner language design, and a language with fewer constructs to learn, but also gives a language that lacks some of the conveniences of the complex language.

A third approach is to place these features outside the language, as libraries, and allow the programmer to include them when they are convenient. A program can use a particular construct by explicitly importing the constructs[2]. For instance, the language could provide a list library that not only defines list data structures, but also extends the language with a compact list syntax:

```
use java.util.Lists;
class SomeClass {
    List⟨String⟩ strs = ["a", "b", "c"];
}
```

This way, the base language can be kept clean and simple and not only provide the convenient abstractions of a more complex language, but provide a much wider range of constructs. Even in the inclusive approach to language design, a construct has to be useful in a relatively broad range of situations to be included. When constructs can be put outside the core language, it becomes reasonable to include mechanisms that

[2]A mechanism similar to this is suggested in (Østerbye 2002), where certain import directives cause plugins to be installed in the compiler

are much more specific, because the inclusion of such a mechanism has much less of an impact on the core language. With this approach, it might even be possible to allow the programmer to specify language extensions.

Before any of this can be made possible, however, it has to be possible to parse input written in such a language. In this case, each input file can include any number of different language features, which means that potentially, each file must be parsed according to a different extended dialect of the base language. In the SQL example, it might be possible to statically generate a parser for each dialect and just select the right one when the system is configured. With a large library of language constructs that can be combined freely, this is impossible, especially if the programmer is allowed to define new extensions. For this, it is necessary to be able to extend the grammar of the language dynamically and produce a parser for the extension on the fly. As mentioned in the previous example, the library presented here makes it possible to combine language extensions, use them to extend an existing language, and to dynamically generate a parser for the result.

The parser library that will be described in this thesis is actually only a small part of making such a mechanism possible. A language extension mechanism can be implemented with different degrees of generality, ranging from a simple macro mechanism to the general mechanism described here. In either case, designing the language mechanisms and the appropriate syntax is as complex an issue as being able to parse the extended program text. On the other hand, having good solutions for the other issues is useless if there is no way to extend the parser.

1.2.3. Local language extensions

In the example above, the scope of a dialect was a single file: the dialect to use in the file is specified in the header of the file and then used for the rest of the file. In some cases, it is convenient to only extend the language locally. It might for instance be that, in a large file, a single function is responsible for reading input and extracting data from it by doing regular expression matching. Inside this function, it would be very convenient to be able to use a regular expression construct like the one available in the Perl language:

```
syntax (java.util.regex.Regex) {
    if (line ~= /[0-9]+:[0-9]+/) {
        /* ... whatever ... */
    }
}
```

Here, the language is extended with a regular expression syntax, but only within the body of the **syntax** statement. To allow this, the parser must not only be extensible, but it must be possible to extend the parser while it is in the middle of parsing, based on input it has already read. In this example, the extension was specified by its name, java.util.regex.Regex, but it could also be useful to allow the programmer to specify the syntax of the extension directly. As exotic at this might seem, several languages in the Lisp family actually provide such a construct.

In the following, we will see how the library not only supports the generation of extended grammars, as described in the previous example, but also allows grammars to be extended dynamically.

1.3. Issues

The examples above have given an impression of what the library must be able to deal with: it must be able to allow grammars to be specified and combined, generate parsers for these grammars on the fly, and allow the generated parsers to be extended dynamically. Designing such a library raises a number of interesting issues, and dealing with these issues is what this thesis is all about.

The first issue is algorithmic: which basic parsing algorithm should be use, that allows dynamic extension. There are a few candidates for this choice, but when considering the properties of the possible algorithms, there is really only one choice[3]. The main issue turns out not to be which algorithm to choose, but how to adapt it to work efficiently.

Besides the question of algorithms, there is the question of interfaces. Two different interfaces must be considered.

Traditionally, parser tools have been aimed at and used by experts, and it has been unnecessary for the user of the program or language to know how the parser was specified, or which algorithm was used. By allowing the user to define syntax extensions, the process of defining the parser has been moved from the expert domain into the domain of the user. This means that it is not acceptable to require that the user is familiar with the peculiarities of any particular parsing algorithm. Anyone who has been through the experience of defining an LALR(1) parser will probably agree that it is not acceptable to require the casual or even the expert user to have the level of knowledge about parsing that is required to define a LALR(1) parser. Generally, usability concerns become much more important, and must be taken into account when designing the basic functionality of the library.

The second interface issue is the developer's interface to the library. The library's role is really as a single component of a larger system, for instance a compiler or interpreter. The second interface to consider is the library's interface as a single component in this larger system.

[3]That choice, if you cannot take the suspense, is top-down, breadth-first parsing.

1.4. Contributions

This thesis represents one of the first systematic studies of dynamically extensible parsers, and makes a number of contributions to that area.

First of all, this thesis develops a coherent design of a library for generating dynamically extensible parsers. While there are existing parser libraries, this is the first one that has been specifically designed to be dynamically extensible. This means that the design presented here deals with a number of issues that have not previously been considered. Also, unlike many parser tools, this design places a great deal of emphasis on giving an exact semantics of the behavior of the generated parser. The library is presented in two parts: a general, language-independent description, and a concrete Java binding of the general design.

Secondly, this thesis develops two new algorithms that are necessary for implementing such a library. The parsing algorithm used here is tailored to allow dynamic extension, and to not require grammars to conform to any specific grammar class, which would make the library much harder to use for non-experts. The algorithm presented here appears to be the first practical adaptation of generalized top-down breadth-first parsing.

The second algorithm presented here is the algorithm for incremental construction of parsers, from grammar specifications. It is incremental in the sense that the parser for an extended grammar can be produced very efficiently from an existing parser because the algorithm only generates the parts of the extended parser that are different from the parser that is being extended. The algorithm presented here is not the first algorithm for incremental parser construction, but it is the first that allows general extensions (both adding, modifying and removing productions), generates top-down parsers, and is non-destructive (that is, the algorithm generates a new parser rather than changing an existing parser).

1.5. Thesis overview

The rest of the thesis is organized as follows:

- Section 2 gives some background on the basics of context-free grammars, context-free parsing and the EBNF grammar notation.
- Chapter 3 and chapter 4 describe and motivate the design of the library. Chapter 3 gives a language-neutral description of the general functionality, and a precise semantics of parsing and parser extension. Based on this, a concrete binding of the library, for the Java language, is given in chapter 4.
- Chapter 5 and chapter 6 explain the algorithms used to implement the library. Chapter 5 describes the incremental compilation process that produces a parser from a specification and chapter 6 shows how the generated parser is executed to parse input.

- Chapter 7 evaluates the efficiency of the prototype library, and compares the efficiency of the generated parsers with parsers generated by two popular parser generators.
- Even though there has been very little work on dynamically extensible parsers, many of the issues discussed here have previously been treated in other contexts. Chapter 8 reviews the relevant related work.
- Finally, chapter 9 concludes and points out possible directions for future work.

2. Background

This chapter gives an overview of the relevant aspects of context-free grammars and context-free parsing. This introduction is not meant to be comprehensive, but is really more a reminder; the reader is expected to already be familiar with the concepts introduced here, at least superficially. For a comprehensive treatment of context-free grammars and context-free parsing, see Grune & Jacobs (1990).

2.1. Context-free parsing

Context-free languages are defined by context-free *grammars*. A context-free grammar defines the language by specifying how sentences in that language can be constructed. The grammar in figure 2.1 defines a language of simple, fully parenthesized expressions. A sentence is derived by starting with a single nonterminal, here $\langle\mathsf{expr}\rangle$, and successively replacing the occurrence of a nonterminal within the sentence with one of its productions. An example of the derivation of a simple expression is shown in figure 2.2 on the next page. There are typically many ways to derive a given sentence; in the example, the derivation always expands the leftmost nonterminal, which makes it a *leftmost derivation.*

A context-free grammar is hence a generative device, a description of how sentences of a language should be constructed. Parsing is the reverse of this process: given a flat string, a parser must determine which steps were taken to generate that string. For instance, if we take the raw text of the string generated in figure 2.2, `(1 + (2 * 3))`, the parser must process the string and recognize its structure according to the grammar, as shown in figure 2.3 on page 11.

Note that usually, the parser does not work with the raw text input directly, but with a tokenized representation produced by a *scanner* that reflects the lexical structure of

$$
\begin{array}{rcl}
\langle\mathsf{expr}\rangle & \rightarrow & \langle\mathsf{int}\rangle \\
 & | & \mathbf{(}\langle\mathsf{expr}\rangle \mathbf{+} \langle\mathsf{expr}\rangle\mathbf{)} \\
 & | & \mathbf{(}\langle\mathsf{expr}\rangle \mathbf{*} \langle\mathsf{expr}\rangle\mathbf{)} \\
 & & \\
\langle\mathsf{int}\rangle & \rightarrow & \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \cdots
\end{array}
$$

Figure 2.1.: A simple expression grammar

the string.

There are many different approaches to parsing strings, but only two of them are widely used: the top-down and the bottom-up approach. Each of these two approaches give rise to a whole family of algorithms that all share the same basic structure. The next two sections describe the basics of top-down and bottom-up parsing.

2.2. Top-down parsing

Top-down parsing is the simplest approach to parsing but is also, as we will see, a less powerful approach than bottom-up parsing. Top-down parsing is also known as *LL* or *predictive* parsing.

A top-down parser parses input by essentially trying to repeat the process of building the input string. The parser starts with a single nonterminal and then successively replacing occurrences of nonterminal with one of their productions. Each time the parser replaces a nonterminal with a production, it consults the input to help it guess which production was used when the input was originally generated.

Consider this simple statement grammar (using the ⟨expr⟩ nonterminal defined in figure 2.1):

$$\begin{array}{lcl} \langle\mathsf{stmt}\rangle & \rightarrow & \textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \langle\mathsf{stmt}\rangle\ \textbf{else}\ \langle\mathsf{stmt}\rangle \\ & | & \{\ \langle\mathsf{stmt}\rangle\ \langle\mathsf{stmt}\rangle\ \} \\ & | & \langle\mathsf{expr}\rangle\ \textbf{;} \end{array}$$

An example of a sentence generated by the grammar is this simple statement:

if 1 then { 2; 3; } else 4;

A figurative view of a few steps of the top-down parsing algorithm, applied to this example, are shown in figure 2.4. The parser uses two structures: a prediction stack, which contains the parser's guess at how the input looks, and the actual input. To begin

$$\begin{array}{lcl} \langle\mathsf{expr}\rangle & \rightarrow & \textbf{(}\langle\mathsf{expr}\rangle\ \textbf{+}\ \langle\mathsf{expr}\rangle\textbf{)} \\ & \rightarrow & \textbf{(}\langle\mathsf{int}\rangle\ \textbf{+}\ \langle\mathsf{expr}\rangle\textbf{)} \\ & \rightarrow & \textbf{(1 +}\ \langle\mathsf{expr}\rangle\textbf{)} \\ & \rightarrow & \textbf{(1 + (}\langle\mathsf{expr}\rangle\ \textbf{*}\ \langle\mathsf{expr}\rangle\textbf{))} \\ & \rightarrow & \textbf{(1 + (}\langle\mathsf{int}\rangle\ \textbf{*}\ \langle\mathsf{expr}\rangle\textbf{))} \\ & \rightarrow & \textbf{(1 + (2 *}\ \langle\mathsf{expr}\rangle\textbf{))} \\ & \rightarrow & \textbf{(1 + (2 *}\ \langle\mathsf{int}\rangle\textbf{))} \\ & \rightarrow & \textbf{(1 + (2 * 3))} \end{array}$$

Figure 2.2.: Derivation of a simple expression

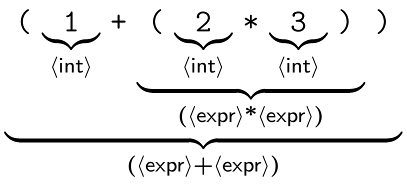


Figure 2.3.: The lexical structure of `(1 + (2 * 3))`.

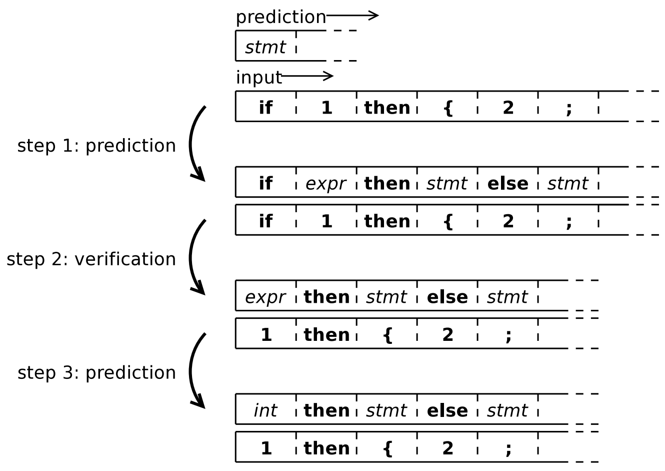


Figure 2.4.: Three steps of the bottom-up parsing process

with, the prediction stack contains ⟨stmt⟩, which reflects the fact that all the parser knows at this point is that the input should be a statement.

In the first step, the parser tries to guess or *predict* which of ⟨stmt⟩'s productions was used when the string was generated, by looking at the input. The first token of the input is **if**, which tells the parser that the nonterminal should be expanded using the **if** production. This means that after the first step, the parser believes that the input is an **if** statement, which is now on the parser's prediction stack.

In the next step, the top of the prediction stack contains a token, **if**, which means that according to the parser's prediction, the next input token should be **if**. It is, and so we have verified that so far the parser's prediction is correct, and it can skip past the **if** token. In the next step, the parser has predicted that the next part of the input is an ⟨expr⟩, and must now predict which of ⟨expr⟩'s productions was used. Looking at the first token of the input, it predicts that the production used was the one generating an ⟨int⟩, and so replaces the ⟨expr⟩ with the body of that production.

This process of prediction and verification goes on until all predictions have been verified and the end of the sentence has been reached. At that point, the parser will have discovered the structure of the input, since it has just derived the input itself, and verified that the derivation was correct. As we saw, the parser processes the input and the prediction from left to right, and always expands the leftmost nonterminal of its sentence first. The term LL means exactly that: it scans the input from left to right (the first L), and always expands the leftmost nonterminal first (the second L).

Of course, it might be that during parsing, the parser makes a prediction that turns out to be wrong. There are different ways of handling this situation. The simplest option is to just give up and report a syntax error. Another option is that the parser can backtrack and try an alternative guess. Only if all alternatives fail, will the parser have to abort and report that the input could not possibly have been derived using the grammar.

Each time the parser guesses which production to use for a nonterminal, it does so by comparing the next token of the input (or possibly the next few tokens) with productions of the nonterminal. The set of tokens that can occur first in a production is called the *start set* of that production. In the example above, all the productions of ⟨stmt⟩ have different start sets. This means that when parsing a sentence as a ⟨stmt⟩, we can predict the right production by simply choosing the one whose start set contains the next token in the input. A grammar with the property that we can choose the right production by only looking at the first token of the sentence is said to be LL(1). In general, an LL(k) grammar is one where we can determine the right production by looking at the next k tokens of input.

2.2.1. Left recursion

Predictive parsers are popular and easy to implement (actually, they are probably popular mainly *because* they are easy to implement), but as mentioned in the beginning, they also have an inherent weakness. Consider a grammar specifying a list of names:

$$\begin{array}{lcl} \langle\textsf{names}\rangle & \rightarrow & \varepsilon \\ & \mid & \langle\textsf{names}\rangle\ \textbf{\textsf{name}} \end{array}$$

If a predictive parser attempts to parse a non-empty list of names as ⟨names⟩, it will go into an loop. To begin with, its prediction will be ⟨names⟩. The first step will be to replace this nonterminal with one of its productions. Since the list of names is non-empty, it cannot take the first production. Hence, the parser replaces ⟨names⟩ with its second production, and the prediction is now ⟨names⟩ **name**. Since the parser goes from left to right, the next step is to replace ⟨names⟩ with one of its productions. But since no input has been consumed, the parser must expand ⟨names⟩ based on exactly the same lookahead as before. This means that it must do the same as in the step before: replace ⟨names⟩ with its second production. After the second step, the prediction is ⟨names⟩ **name name**. From here, the parser will loop and keep expanding the ⟨names⟩ nonterminal into its second production, growing its prediction without consuming any input.

The problem is that the grammar is *left recursive* and, as this demonstrates, left recursive grammars can cause a top-down parser to go into a loop of predicting and predicting without ever getting to verifying the predictions. The second approach, bottom-up parsing, is more complex and harder to generalize, but does not have this limitation.

2.3. Bottom-up parsing

The source of the weakness in top-down parsing is that a top-down parser has to make it's prediction in advance, when the parser has only seen the very beginning of the production. A bottom-up parser postpones this decision until it has seen input corresponding to the whole production. Two steps of the algorithm, which will be described in a moment, are shown in figure 2.5 on the next page.

A top-down parser divides the input into two parts: the stack, containing the input seen so far, partially parsed, and the input that remains to be parsed. At each step, the parser has two options: it can either *shift* a token or *reduce* a production.

Shifting means moving a token from the input to the stack. The first step in the example in figure 2.5 shifts the **}** token. Reducing means recognizing that a number of elements on the top of the stack correspond to a production in the grammar, and replacing them with the corresponding nonterminal. In the second step of the example, the **{**⟨stmt⟩⟨stmt⟩**}** form is recognized as a production and replaced by the corresponding nonterminal, ⟨stmt⟩. Each nonterminal on the stack has a representation of the part of the parse tree they represent. Here, the ⟨stmt⟩ inserted is associated with the information that it represents a **{**⟨stmt⟩⟨stmt⟩**}**.

How does the parser decide which action to take? The most popular approach, LR parsing, which was invented by Knuth (1965), makes the decisions using a finite state

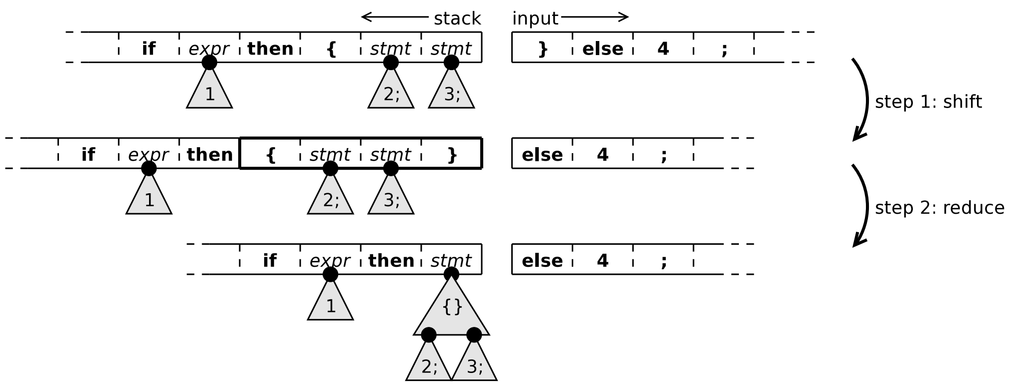


Figure 2.5.: Two steps of the top-down parsing process

automaton. At each step, the parser can use the current state in the automaton, together with the next token (or possibly the next few tokens) of the input, as an index into a table that tells the parser which action to take. The action tells the parser which state to go to, and either to shift, reduce corresponding to a specified production, or not do anything other than change state. An LR parser that uses the next k input tokens to look up the action is called an LR(k) parser. LR parsers almost always use lookahead 1 since the size of the lookup tables grows exponentially as k increases.

It sometimes happens that an LR parser is unable to decide whether to shift or reduce; this is known as a shift/reduce conflict. It might also happen that there are several ways to reduce, which is known as a reduce/reduce conflict. There are several ways for a parser to deal with conflicts. Some parsers simply refuse to parse according to a grammar if conflicts can occur. The generalized LR parsers, which will be described later, deal with conflicts by simply trying all the alternatives in parallel.

Generating the automaton and lookup table to control an LR parser is a complex process. An LR parser is certainly not something you would write by hand, they must be generated by a parser generator. Also, the lookup tables generated tend to be very large. Instead, LALR(1) parsers, which are much smaller, are often used instead. LALR(1) parsers save space by selectively discarding lookahead information and turn out to be much smaller but almost as powerful as LR(1) parsers.

We have now seen two very different approaches to context-free parsing, and discussed some of their differences and limitations. Later, in section 3.12, we will return to this and discuss how these differences affect the choice of algorithm used in the implementation of the library. The last section of this chapter will review an often-used extension of BNF, which was the formalism used above to specify grammars.

2.4. BNF and EBNF

The algorithms described above relied on the fact that the grammar had a particular form: all the productions were a sequence of terminal and nonterminal symbols. This form is known as BNF, which can either stand for Backus Normal Form or Backus Naur Form, depending on taste[1]. All context-free grammars can be expressed as BNF, and it is often used in practice. The problem with BNF is that many of the idioms used in grammars cannot be expressed directly. For instance, a non-empty list of $\langle \mathsf{x} \rangle$s, would be expressed as

$$\begin{array}{lcl} \langle \mathsf{x\text{-}list} \rangle & \rightarrow & \varepsilon \\ & \mid & \langle \mathsf{x} \rangle \ \langle \mathsf{x\text{-}list} \rangle \end{array}$$

Some idioms, like lists, are often used in practical grammars, and having to introduce a new auxiliary nonterminal for each list turns out to be somewhat awkward in practice. Extended BNF, or EBNF, allows productions to be written using a number of regular expression operators. The most commonly used EBNF operators are

- φ^* which represents a possibly empty list of φs. This operator is sometimes written as $\{\varphi\}$.
- φ^+ which represents a nonempty list of φs.
- $[\varphi]$ which represents an optional φ. The notation $\varphi^?$ also widely used for this operator.

Using EBNF, a non-empty list of $\langle \mathsf{x} \rangle$s can be expressed directly as $\langle \mathsf{x} \rangle^*$, rather than through an auxiliary nonterminal. It is important to notice, however, that any grammar written using EBNF can be straightforwardly translated in to a grammar in BNF. The EBNF operators are purely a convenience, they do not add any expressive power.

A standard exists for EBNF (ISO/IEC 1996) but, as indicated above, a number of alternative notations are in widespread use.

[1]The term Backus Naur Form was suggested by Knuth (1964), because BNF is not a normal form in the usual sense of the term.

3. Language

This chapter describes and motivates the design of the library. In the following, the library will often be referred to as *Tedir*. The name Tedir has been chosen for the library because of its Tolkien-ish appeal, and the fact that it does not mean something offensive in any language I know. Actually, it does not mean anything; it was randomly generated.

Unlike most parser generators, Tedir is a library, which means that any actual implementation of the design presented here will be bound to a some host language. The underlying design of the library is, however, language neutral, and to be able to describe the design without tying it to a particular implementation language, we will use an artificial grammar definition language. The next chapter gives an example of a concrete implementation of the library, in Java.

The syntax of the language is essentially a slightly extended version of EBNF, which was described in the previous chapter. What the library adds is a number of operations that can be performed with grammars, a model for how to interact with the parser generated for the grammar, is used and a dynamic semantics of how input is parsed using the parser.

3.1. Defining a grammar

An example of the definition of a nonterminal, written in the syntax of the specification language, is given in figure 3.1. It defines a nonterminal, ⟨stmt⟩, that generates simple statements. Note that this definition uses the regular * operator, Kleene's star, which was described in section 2.4. In general, the right-hand side of a production can be an arbitrarily complex expression involving regular expression operators. The set of available operators is given in the next section.

$$\begin{array}{rcl}
\langle\textsf{stmt}\rangle & \rightarrow & \textbf{if}\ \langle\textsf{expr}\rangle\ \textbf{then}\ \langle\textsf{stmt}\rangle \\
 & | & \textbf{if}\ \langle\textsf{expr}\rangle\ \textbf{then}\ \langle\textsf{stmt}\rangle\ \textbf{else}\ \langle\textsf{stmt}\rangle \\
 & | & \textbf{while}\ \langle\textsf{expr}\rangle\ \textbf{do}\ \langle\textsf{stmt}\rangle \\
 & | & \{\ \langle\textsf{stmt}\rangle^*\ \} \\
 & | & \langle\textsf{expr}\rangle\ \textbf{;}
\end{array}$$

Figure 3.1.: Example of a grammar specification

Each production has an associated action. The form of the action strongly depends on the implementation language of the library, but the exact choice of representation for the actions is not important to the description of the library and the actions are not shown in the specifications. An example of a possible choice of representation is given in the next chapter.

Each production can be given name. The name is used to refer to the production when manipulating the grammar and can be used when, for instance, removing productions. How grammars are manipulated will be described in section 3.10.1. In the example in figure 3.1, none of the productions have a name; with names, the grammar would look like this:

$$\begin{array}{rcl} \langle\mathsf{stmt}\rangle[\mathsf{shortIf}] & \rightarrow & \textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \langle\mathsf{stmt}\rangle \\ [\mathsf{longIf}] & | & \textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \langle\mathsf{stmt}\rangle\ \textbf{else}\ \langle\mathsf{stmt}\rangle \\ & \vdots & \end{array}$$

3.2. Expressions

As mentioned before, the body of a production can be a general regular expression. This feature is what makes the it an ELL rather than just an LL parser generator. The specification in figure 3.1 on the page before gives an example of this: the block statement uses the * operator to specify a possibly empty list of ⟨stmt⟩s. A number of regular operators are available; they are shown in figure 3.2 on the facing page.

Most of these operators are well-known BNF and EBNF operators. The nonstandard operators are the two infix repetition operators, $\{\varphi_1 : \varphi_2\}^*$ and $\{\varphi_1 : \varphi_2\}^+$, and the suspend and ignore operators. The suspend and ignore operators are related to features that have not yet been described, and will be introduced later. The operator $\{\varphi_1 : \varphi_2\}^*$ represents a possibly empty sequence of φ_1s, separated by φ_2s. A list of terms separated by a terminal is very common in programming language syntax. For instance, the tuple syntax in ML is a possibly empty, comma-separated list of expressions enclosed in parentheses:

$$\langle\mathsf{expr}\rangle \quad \rightarrow \quad \textbf{(}\ \{\langle\mathsf{expr}\rangle{:}\textbf{,}\}^*\ \textbf{)}$$

The $\{\varphi_1 : \varphi_2\}^+$ operator is similar, except that it requires at least one occurrence of φ_1.

3.3. Grammar wellformedness

A parser can only be generated for a grammar if the grammar is well-formed. A grammar is well-formed if it conforms to three requirements. First of all, all the nonterminals that are referenced must be defined in the grammar. Secondly, a grammar is illegal if it

Syntax (φ)	**Description**
term	*terminal symbol*
$\langle$nonterm$\rangle$	*nonterminal*
ε	*empty*
$\varphi_1\,\varphi_2\,\cdots\,\varphi_n$	*sequencing*
$\varphi_1 \mid \varphi_2 \mid \cdots \mid \varphi_n$	*choice*
φ^*	*zero or more repetitions*
φ^+	*one or more repetitions*
$[\varphi]$	*optional*
$\{\varphi_1 : \varphi_2\}^*$	*zero or more infix repetitions*
$\{\varphi_1 : \varphi_2\}^+$	*one or more infix repetitions*
$\underline{\varphi}$	*ignore*
$[\![\varphi]\!]$	*suspend*

Figure 3.2.: Possible expressions in Tedir

contains a repetition expression, φ^* or φ^+, where φ can produce the empty string, or an infix repetition expression, $\{\varphi_1 : \varphi_2\}^+$ or $\{\varphi_1 : \varphi_2\}^*$, where both φ_1 and φ_2 can produce the empty string. This has a technical reason that will be explained later. Most importantly, however, any input parsed according to grammar containing such expressions cannot have a unique parse tree, which means that even if they were legal, such grammars would be meaningless.

The final requirement is that the grammar must not be left recursive. The parsers constructed by the library are based on the top-down approach, and as described in section 2, top-down parsers cannot handle left recursion. Back then, I gave an example of left recursion, but did not define it generally; we will do that now. A grammar is left recursive if it contain a left recursive nonterminal. A left recursive nonterminal is one that can produce a sentence starting with itself. In section 2 we saw an example of a left recursive nonterminal: $\langle$names$\rangle$ caused the parser to loop because it could produce a sentence starting with itself, $\langle$names$\rangle$ **name**. Left recursion can also be indirect, as in this example:

$$\begin{array}{lcl} \langle \mathsf{a} \rangle & \rightarrow & \langle \mathsf{b} \rangle\ \mathbf{x} \\ \langle \mathsf{b} \rangle & \rightarrow & \langle \mathsf{a} \rangle\ \mathbf{y} \end{array}$$

Here we have $\langle$a$\rangle$ → $\langle$b$\rangle$ **x** → $\langle$a$\rangle$ **y x**, which shows that $\langle$a$\rangle$ is left recursive, even though it does not reference itself directly.

We might hope that left recursion was something that you did not want or need to have an a grammar anyway. Unfortunately, having left recursion really is quite useful and convenient. For instance, in an LR parser, infix operators can be specified like this (given appropriate precedence directives):

$$\begin{array}{lcl} \langle\mathsf{expr}\rangle & \rightarrow & \langle\mathsf{expr}\rangle\ \mathsf{+}\ \langle\mathsf{expr}\rangle \\ & | & \langle\mathsf{expr}\rangle\ \mathsf{-}\ \langle\mathsf{expr}\rangle \\ & | & \langle\mathsf{expr}\rangle\ \mathsf{*}\ \langle\mathsf{expr}\rangle \\ & | & \langle\mathsf{expr}\rangle\ \mathsf{/}\ \langle\mathsf{expr}\rangle \\ & | & \textbf{num} \end{array}$$

This way of defining infix operators is very convenient, and defining them without left recursion requires the grammar to be rewritten. A left recursive grammar can always be rewritten to get rid of the left recursion, and even though it is a declared goal of the design that it should not be necessary to rewrite grammars, it is simply unavoidable here.

Why, then, use the top-down approach, with this weakness, when the bottom-up approach avoids this problem? I will postpone the discussion the exact choice of the underlying parsing algorithm until the end of the chapter, because this choice is mainly motivated by issues that have not yet been discussed.

3.4. Ambiguities

An important issue is illustrated by the fact that the grammar in figure 3.1 is even legal and can be used to parse input. The thing is that the grammar specified is ambiguous: it contains an unresolved “dangling **else**” conflict. When given input such as this:

$$\textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \langle\mathsf{expr}\rangle;\ \textbf{else}\ \langle\mathsf{expr}\rangle;$$

it can be parsed either as

$$\textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \Big(\ \textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \langle\mathsf{expr}\rangle;\ \Big)\ \textbf{else}\ \langle\mathsf{expr}\rangle;$$

or as

$$\textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \Big(\ \textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \langle\mathsf{expr}\rangle;\ \textbf{else}\ \langle\mathsf{expr}\rangle;\ \Big)$$

In some sense, the ambiguity in the grammar can most accurately be described as a *potential* ambiguity, since the ambiguity only shows itself when the parser is given certain input. So even though the grammar is ambiguous, it can still be used to meaningfully and unambiguously parse input, just not input that does not have a unique parse tree.

The decision to allow ambiguous grammars is fundamental and has far-reaching consequences, most of which are unfortunately complications. The reason for allowing this is simple: the library would probably be almost useless if ambiguous grammars were illegal.

This decision is actually not at all motivated by a desire to parse ambiguous grammars. The problem is that deciding whether a grammar is ambiguous is undecidable. The parsers that require grammars to be unambiguous do not actually do so directly. Rather, they make a stricter requirement which *implies* that the grammar is unambiguous. For instance LL(k) parsers require that it must be possible to choose the right production of a nonterminal by looking forward k tokens. The problem with this is that a grammar can very well be unambiguous without being LL(k) (or LALR(1) or any other such class). Only a very small part of the unambiguous grammars are LL(k) or LALR(1).

On the other hand, all unambiguous grammars can be rewritten and massaged into such a grammar. A good example of the considerations that go into such a restructuring can be found in the description of an LALR(1) grammar for Java in Gosling, Joy & Steele (1996, chapter 19). This rewriting is something that provably cannot be performed automatically and must be done by a human programmer. This is usually no problem since the parser is usually developed along with the program that uses it.

But when the grammars are constructed in a running program by adding together different grammar modules and applying user-defined macros, the result might well be unambiguous, but it is highly unlikely that the resulting grammar will be LALR(1) or LL(1). Even if the parser was extensible, any such restrictions on the extended grammars would make it almost impossible to develop extensions independently because modules that were developed independently are likely to be illegal when combined. Because of this, the library must be able to handle ambiguous grammars.

3.5. Expression values

The previous sections have described how a grammar is defined. The following sections describe how the parser constructed for a grammar parses input. This first section is concerned with how the value of an expression is formed while parsing.

3.5.1. Production actions

In most parser generators, a production is associated with an *action*. The following is part of a Bison specification, taken from the Bison manual:

```
exp     : NUM         { $$ = $1;      }
        | exp '+' exp { $$ = $1 + $3; }
        | exp '-' exp { $$ = $1 - $3; }
        | exp '*' exp { $$ = $1 * $3; }
```

When a production has been recognized in the input, the associated action is executed. In the action code, the parsed nonterminal is represented by `$$`, and can be assigned a value. The subterms of the parsed input can be are referred to through the term's index, using the form `$`*i*, so that the first subterm of the production is `$1`, the second is `$2` etc.

In the first production, the value of the nonterminal is taken to be the value of the terminal `NUM`; the value of the terminal symbols are supplied by the scanner. In the other productions, the value is computed based on the value of the sub-terms of the parsed input.

Tedir has a similar model. As mentioned, each production has an associated action. When a production has been recognized, the action is invoked and passed the value of the input parsed. Instead of giving the value of the nonterminal by assigning to `$$`, the value of the nonterminal is taken to be the value returned by the production action. The rest of this section is concerned with how the value that is passed to the production action is constructed.

3.5.2. Why are expression values an issue?

The most important difference between Bison and Tedir is that in Bison, it is the actions that extract the relevant parts of the input, whereas in Tedir, the value is automatically packaged by the parser and passed to the action. In the bison grammar above, the relevant information for the actions of the last three productions are the expressions, `$1` and `$3`, while the operator, `$2`, is irrelevant and not used in the action. If something is irrelevant, the action just doesn't access that part of the expression.

In Tedir, the parser is responsible for packaging the value of the production and passing it to the action. But the parser doesn't a priori know which parts of the parsed input will be relevant to the action, and so, in a grammar like the Bison expression grammar, it would have to pass the value of the operator to the action, just in case the user needed it. With simple expressions, that is only an annoyance, but with larger expressions the actions quickly end up spending a majority of their time extracting the relevant data from the value of the expressions. Instead, we want to be able to express to the parser which parts of the input are relevant, and have it build values that only contain relevant information.

Another thing that makes this more complex in Tedir is that productions can be general regular expressions. In Bison, using an index to reference the components of the parsed input is only possible because the productions are just a sequence of terminal and nonterminal symbols. With general regular expressions, the parsed input has a much more complex structure.

3.5.3. Relevant and irrelevant expressions

Consider this production:

$$\langle\textsf{import-clause}\rangle \quad\rightarrow\quad \textbf{import}\ \{\textbf{id}:\textbf{.}\}^{+}\ [\textbf{.}\ \textbf{*}]\ \textbf{;}$$

This production produces the keyword **import**, followed by a dot-separated list of names, possibly ending with **.***, and in any case terminated by a semicolon. When such an expression has been parsed, the relevant information is the list of identifiers and an indication of whether or not there was a **.*** at the end. For instance, for this input:

```
import java.lang.*;
```

we would want something like this:

Ids	.*
[java, lang]	true

Similarly, for this input

```
import org.xml.sax.HandlerBase;
```

the relevant information would be this:

Ids	.*
[org, xml, sax, HandlerBase]	false

To get this, we have to specify to the parser which parts of the input are relevant and which are not. To the parser, the **id** terminal is not seen as more relevant than the **import** terminal, even though we know that the **id** terminal represents a name we are interested in, and the **import** token is just dumb syntax. There is only one solution to this: it must be possible for the user to tell the parser explicitly which parts of the productions are are relevant and which irrelevant.

3.5.4. Value construction

To the parser, there are two kinds of expressions: relevant expressions and irrelevant expressions. When constructing the value of an expression, the parser will work to only include relevant information. An expression φ can be marked as irrelevant by using the *ignore* operator, $\underline{\varphi}$. All expressions have a default relevance, and while a relevant expression can be ignored and thereby made irrelevant, an irrelevant expression cannot be made relevant.

All expressions have a value, even irrelevant expressions; they have a dummy null value. The value of a composite expression, for instance a sequence or a repetition expression, is constructed from the values of it subexpressions. In the construction of such a composite value, the governing principle is that the value of irrelevant subexpressions should be discarded whenever possible. The value of each kind of expression is described in the following.

Atomic expressions

A terminal, **term**, is considered relevant. The value of a terminal must be supplied by the scanner. Usually, terminal symbols are actually irrelevant, and an implementation should probably provide a shorthand for specifying an irrelevant terminal. For uniformity, however, all irrelevant terminals will be marked explicitly here.

A nonterminal, $\langle\mathsf{nonterm}\rangle$, is considered relevant and its value is the value returned by the action that was invoked when the nonterminal was parsed.

The empty expression, ε, is considered irrelevant and always has the null value.

Basic composite expressions

The value of a sequence of expressions, $\varphi_1\ \varphi_2\ \ldots\ \varphi_n$, depends on the φ_is. If all the subexpressions are irrelevant, the whole sequence is considered irrelevant and has value null. If there is exactly one relevant subexpression, the value of the whole sequence if the value of that subexpression. If there are $k > 1$ relevant subexpressions, the value is a vector of length k where the ith entry contains the value of the ith relevant subexpression.

The value of a choice between expressions, $\varphi_1 \mid \varphi_2 \mid \cdots \mid \varphi_n$, is the value of the φ_i chosen. If all φ_i are irrelevant, the whole choice is considered irrelevant and will have value null.

The optional expression, $[\varphi]$, is semantically identical to $(\varphi|\varepsilon)$: its value is either the value of φ or null. If φ is irrelevant, the whole expression is considered irrelevant.

Repetition expressions

If an expression φ is relevant, φ^* and φ^+ are both considered relevant, and the value of each expression is a list containing the values of the φs of the sequence. Otherwise the repetitions are considered irrelevant.

If φ_1 is relevant, $\{\varphi_1 : \varphi_2\}^*$ and $\{\varphi_1 : \varphi_2\}^+$ are both considered relevant, and the value of each expression is a list containing the values of the φ_1s. Note that this means that there is no way to get access to the value of the separator in an infix repetition expression.

From this description, it is probably clear that the rules do what they were designed for: they discard the irrelevant parts of the input and, as much as possible, only return the relevant parts. It is not, however, clear if this works well in practice. In the next section, I will give an example of how these rules work to produce the value of a complex expression. Honestly, though, it is probably necessary to use this mechanism in practice to get a feel for the intuition behind these rules.

3.5.5. A simple example

This section gives a short example showing how the value of an expression is constructed using the rules above. The example is the value of the **import** statement shown before. This time, however, the irrelevant parts of the expression has been marked to be ignored.

$$\langle\textsf{import-clause}\rangle \quad \rightarrow \quad \underline{\textbf{import}}\ \{\textbf{id}:\underline{\textbf{.}}\}^+\ [\underline{\textbf{.}}\ \textbf{*}]\ \underline{\textbf{;}}$$

In this production, all terminals except **id** and ***** have been marked as irrelevant. Let's consider how the value of this expression is constructed. The outermost expression is a sequence. The first and last terms are marked as irrelevant, so the value of the sequence will be a vector with two entries: one for the second and one for the third subexpression, which are relevant. The second subexpression, $\{\textbf{id}:\underline{\textbf{.}}\}^+$, is a repetition of a relevant value, which results in a list of that value. In this case, that will be the list of identifiers. The second subexpression, [. *], is an optional relevant expression which gives either null, or the value of *. The total value of the expression will be a two-entry vector containing the list of identifiers and either null or the value of *:

Input	**Value**
`import java.lang.*;`	{[java, lang], *}
`import org.xml.sax.HandlerBase;`	{[org, xml, sax, HandlerBase], null}

The process of systematically deducing the value constructed is tedious, but the way relevant values are propagated outwards and irrelevant values are discarded turns out to be completely intuitive in practice. Deriving the value of an expression systematically is something you very rarely have to do.

Before, we took the value of a terminal symbol to be whatever value was specified by the scanner. In the next section, we will look at the interface between the parser and the scanner.

3.6. Scanner interface

As mentioned, the input is usually processed by a scanner before it reaches the parser. Some parser tools allow the scanner and parser to be specified together, but in Tedir, the scanner must be provided by the user, possibly using a scanner generator. How the scanner is generated is really irrelevant to the library – it can work together with any object that conforms to a simple interface, which will be specified in a moment.

The scanner is responsible for grouping the raw input into tokens. For instance, this input:

```
int square(int x) {
    return x * x;
}
```

would be translated into this sequence of tokens, before reaching the parser:

INT ID("square") LPAREN INT ID("x") RPAREN LBRACK RETURN ID("x") STAR
ID("x") SEMI RBRACK EOF

Some tokens are "atomic", while others have an associated value. In this example, identifier tokens, ID, are associated with a string, which gives the text of the identifier. The sequence is terminated by a special EOF token. This is how the parser sees the input: a sequence of tokens, each with an associated value, and terminated by the EOF token. In the example above, some tokens really have no meaningful value; they can be given a null value.

Any object that provides the following can be used by the parsers generated by the library:

- A definition of which token is the designated EOF token.
- A procedure with which the scanner can be advanced to the next token.
- A procedure for getting the current token and token value.

The exact form of this interface is highly implementation- and language dependent, but any object that supplies the above functionality can be used with the parser.

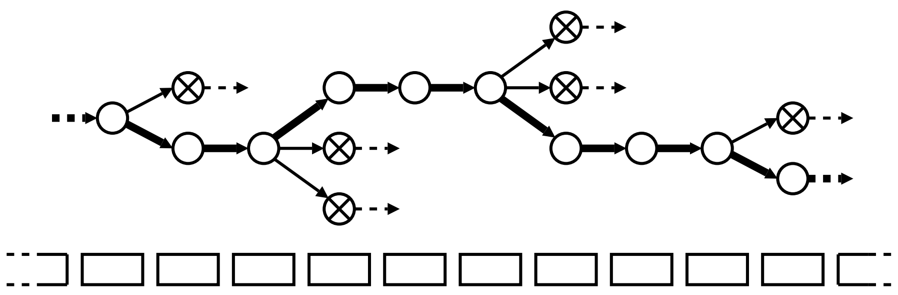
This interface gives complete separation between the parser and scanner. In many parser tools, the two are more closely connected. In some cases, the parser and scanner are specified together, and the same tool generates both. In other cases, the scanner can be made to interpret the input differently, depending on what is expected by the parser. The model with a separate parser and scanner has been chosen simply because it is not unreasonably inconvenient – it is still a widely used approach in practice – and factoring out the scanner makes the design of the parser much simpler.

3.7. Parser execution nodel

So far, we have been able to treat the parser without dealing with the complications caused by the decision to allow general grammars. That is the subject of this section. In the following, we will dig into some of the more subtle details of the dynamic behavior of the parser. Much of this might seem like implementation details and to some extent, it is. It is, however, necessary to know the underlying execution model to understand some of the design decisions described later, and explain some subtleties in the way actions are executed. It is not necessary to know this in order to use the library; the extent to which the user has to be aware of this is discussed in 3.9.1.

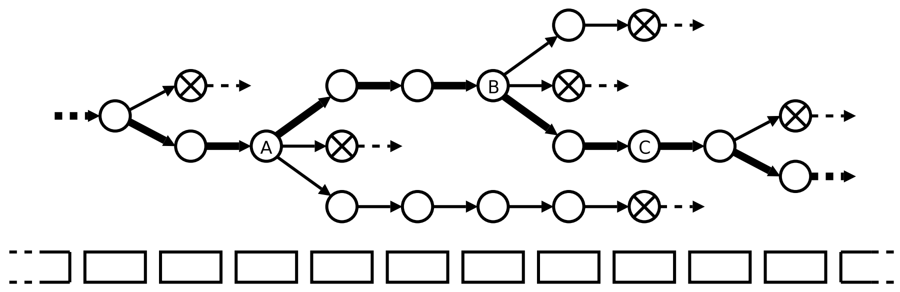
3.7.1. Parsing with ambiguities

As mentioned, in an LL(1) parser, is must be possible to make all choices using only a single token of lookahead. This means that it will never be necessary for an LL(1) parser to read ahead more than a single token to make a decision. This diagram shows an abstract view of the control of an LL(1) parser, where the boxes at the bottom represent tokens:



At each point where a decision must be made, the parser can use the next token to rule out all possible paths except one.

In Tedir, there is no such requirement on the grammar, and determining the right choice based on a single token of lookahead will in general be impossible. Instead, the parser sometimes has to try to parse the same input in several different ways before it finds the right way, as indicated by this diagram, which is explained below:

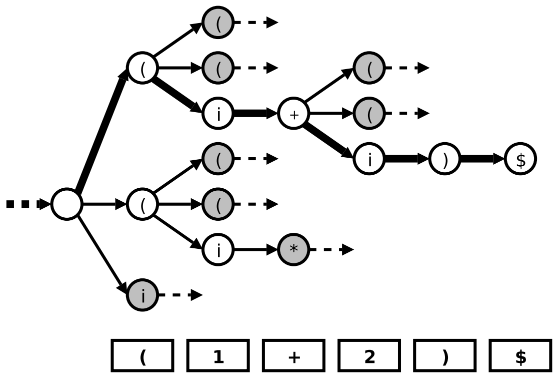


At the position marked A, the parser has to choose one of the three possible paths. One path can be ruled out by looking at the next token, but the choice between the two other paths cannot be made locally, so the parser continues along *both* possible paths. This situation will, in the following, be referred to as a *local ambiguity*. A few tokens later, at position B, another local ambiguity is encountered, which means that the parser must now explore three different paths. Finally, at position C, two of the paths turn out not to match the input, and the single correct path has been identified.

An concrete example is trying to parse using this simple grammar:

$$
\begin{array}{lcl}
\langle\mathsf{expr}\rangle & \rightarrow & \textbf{(}\langle\mathsf{expr}\rangle \;\textbf{+}\; \langle\mathsf{expr}\rangle\textbf{)} \\
 & | & \textbf{(}\langle\mathsf{expr}\rangle \;\textbf{*}\; \langle\mathsf{expr}\rangle\textbf{)} \\
 & | & \textbf{int}
\end{array}
$$

The diagram below shows what happens if we use this grammar to parse the expression **(1 + 2)** as an ⟨expr⟩ (here using **$** for **eof** and **i** for **int**).



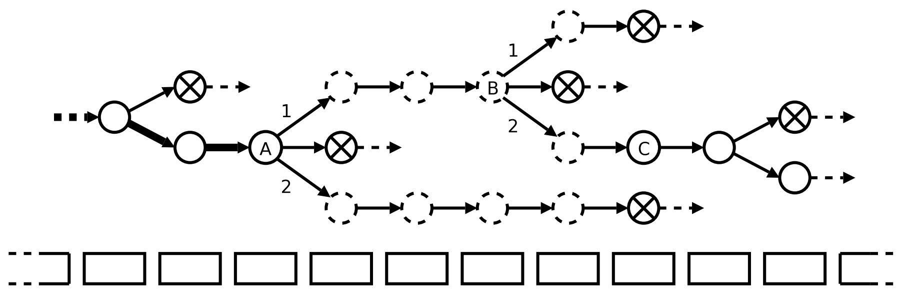
At the first token, it cannot be decided which of the two first productions of ⟨expr⟩ to use, so the parser tries out both productions in parallel, one "thread" parsing the input as (⟨expr⟩+⟨expr⟩) and the other as (⟨expr⟩*⟨expr⟩). Both threads parse the following input, but when they reach the + it becomes clear that the input cannot be parsed as (⟨expr⟩*⟨expr⟩), and the one thread fails, leaving the other to parse the rest of the input.

Note that the word "thread" is used in a limited sense, and not in the sense of threads used for parallel processing. Two threads cannot, for instance, be executed on different processors. As we will later see, the threads used here are similar, on some level, to ordinary threads or coroutines, which is why the term is used. Still, a clear distinction must be made between threads at the operating system or programming language level and the threads used here.

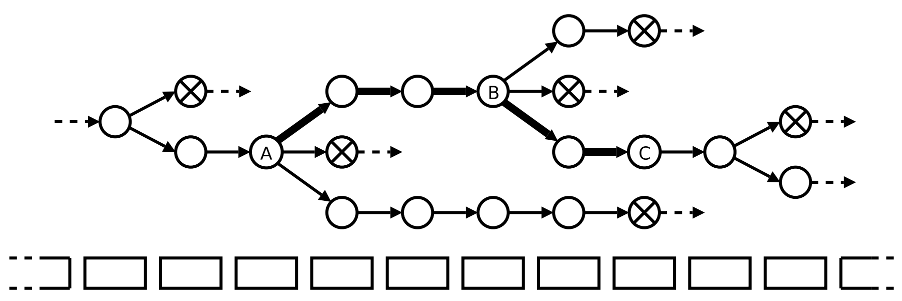
3.7.2. Tentative parsing

As described in section section 3.5.1, an action is invoked when a production has been recognized. What does that mean when the parser is parsing the same input in several different ways? When, as the example before, two threads are parsing the same input, does that mean that they also both invoke the user-supplied actions?

The answer is no: when the parser is exploring several different ways of parsing the input, no user actions are invoked. This is called *tentative* parsing. When an ambiguity is met, the parser turns off all actions and tentatively tries out several different ways of parsing the input, in parallel, until it has determined the correct path. Then it goes back to the position of the original ambiguity and "reparses", executing all user actions, following the path found during the tentative parsing. In the example from the previous section, here is a figurative view of how the disambiguation would play out:



At A, a local ambiguity is encountered, and the parser goes into tentative parsing mode. It then scans forward in two threads, each representing one of the possible ways of parsing the input. At B, another local ambiguity is found, which causes another new thread to be created. Finally, at C, two of the threads turn out not to match the input, leaving only a single thread is left. This thread has stored the information that it first took branch 1 and then branch 2. The parser then returns to A and parses the input again, this time going the same way as the surviving thread:



After this, the parsing can continue as before, starting from C.

3.7.3. Efficiency

There are two efficiency issues with this model, the "big issue" and the "little issue". The big issue is that as soon as we start spawning threads and parsing several paths in parallel, the time complexity of the parser becomes potentially exponential. As we will see in 7, this turns out not to be a problem in practice. The little issue has to do with the fact that is it clearly less expensive to parse input directly than first parsing it tentatively and then reparsing.

Most of the examples we have seen so far have actually required the parser to go into tentative parsing mode. For instance, in he the running ⟨stmt⟩ example:

$$\begin{array}{rcl} \langle\mathsf{stmt}\rangle & \rightarrow & \textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \langle\mathsf{stmt}\rangle \\ & | & \textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \langle\mathsf{stmt}\rangle\ \textbf{else}\ \langle\mathsf{stmt}\rangle \\ & \vdots & \end{array}$$

When parsing a statement, if the lookahead token is **if**, the parser is unable to decide whether to parse the input as a long if or short if statement, and must try both paths. That is clearly inefficient, especially since the first part of the productions are the same. There is an easy way to work around this, however:

$$\begin{array}{rcl} \langle\mathsf{stmt}\rangle & \rightarrow & \textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \langle\mathsf{stmt}\rangle\ [\textbf{else}\ \langle\mathsf{stmt}\rangle] \\ & \vdots & \end{array}$$

Here, the decision is postponed until after the the common prefix of the productions has been parsed. The parser may still have to parse the following input tentatively, but instead of doing that at the beginning of the production, it is postponed until just before the **else** terminal. In practice, this refactoring gives a considerable improvement in performance.

The conclusion is not, however, that the user should make sure to "fold" the common prefixes of productions to make the parser more efficient. As mentioned before, it is a clear goal that the user should not have to manipulate the grammar, but just write it in the way that is most convenient and readable. The example above demonstrates that even though the library allows grammars to be written in whatever form is most convenient, the combination of a potential efficiency issue and an easy workaround gives a strong incentive to refactor the grammar in a way that makes it less straightforward, but more efficient.

The way to resolve this is to simply require that the library recognizes this situation and performs the optimization automatically, so that there is no motivation for the user to do the optimization by hand. The library guarantees that *when making a choice between two expressions that share a common prefix, the parser will not actually make the choice until it has parsed that common prefix.* In the ⟨stmt⟩ example above, that means that since the two **if** statements share a common prefix, the choice will not actually be inserted until just before the **else** terminal, which corresponds to the manual optimization. A more complex example is the nonterminal that represents field and method declarations in Java,

$$\begin{array}{lcl}
\langle\mathsf{decl}\rangle & \to & \langle\mathsf{modifier}\rangle^* \ \langle\mathsf{type}\rangle \ \{\langle\mathsf{var\text{-}decl}\rangle\mathsf{:}\mathbf{,}\}^+ \ \mathbf{;} \\
 & | & \langle\mathsf{modifier}\rangle^* \ \mathbf{void} \ \mathbf{id} \ \langle\mathsf{params}\rangle \ [\mathbf{throws} \ \{\langle\mathsf{qual\text{-}ident}\rangle\mathsf{:}\mathbf{,}\}^+] \\
 & | & \langle\mathsf{modifier}\rangle^* \ \langle\mathsf{type}\rangle \ \mathbf{id} \ \langle\mathsf{params}\rangle \ [\mathbf{throws} \ \{\langle\mathsf{qual\text{-}ident}\rangle\mathsf{:}\mathbf{,}\}^+] \\
 & | & \langle\mathsf{type\text{-}decl}\rangle
\end{array}$$

which is guaranteed to be parsed as efficiently as this (for readability, the choices are stacked vertically):

$$\langle\mathsf{decl}\rangle \to \left|\begin{array}{l}
\langle\mathsf{modifier}\rangle^* \left|\begin{array}{l}
\langle\mathsf{type}\rangle \left|\begin{array}{l}
\{\langle\mathsf{var\text{-}decl}\rangle\mathsf{:,}\}^+ \ \mathbf{;} \\
\mathbf{id} \ \langle\mathsf{params}\rangle \ [\mathbf{throws} \ \{\langle\mathsf{qual\text{-}ident}\rangle\mathsf{:,}\}^+]
\end{array}\right. \\
\mathbf{void} \ \mathbf{id} \ \langle\mathsf{params}\rangle \ [\mathbf{throws} \ \{\langle\mathsf{qual\text{-}ident}\rangle\mathsf{:,}\}^+]
\end{array}\right. \\
\langle\mathsf{type\text{-}decl}\rangle
\end{array}\right.$$

This was the last part of the description of how expressions are parsed. The next few sections deal with the issue of suspending the parser, which is what the mechanism that makes it possible for other parsers to locally take over parsing, and for the parser to be dynamically extended.

3.8. Suspending the parser

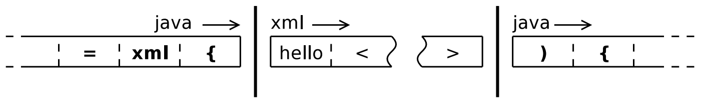
Sometimes it can be convenient to suspend the parser while it is parsing input. One possible application of this is to parse embedded languages. For instance, consider a variant of the Java language that allows embedded XML:

XML hello = **xml** { Hello <i>world</i> };

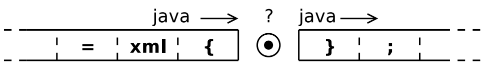
Writing a single parser that can both parse the Java code and the embedded XML is hard because the lexical structure of Java is quite different from the lexical structure of

XML. There really needs to be two scanners, one for the Java code and one for the XML. Also, on top of the lexical problems, the Java and XML parsers are logically separate modules, so it would be far preferable if it was possible to develop and maintain them separately, rather than having them thrown together in a single module.

If there were really two separate parsers for parsing the Java and XML, this is a possible scenario for how they would work together to parse the input: when the Java parser has read the **xml {**, it stops and turns over control to the XML parser. The XML parser parses the XML, and when it reaches the **}** it gives control back to the Java parser which continues to parse the input from where the XML parser left off. That is shown in this figure:



Actually, the Java parser doesn't know that the embedded language is XML. This is exactly the point: the XML parser and the Java parser are completely independent and don't know anything about each other. To the Java parser, the process would look more like this:



After the opening bracket the parser is interrupted, *something happens*, and afterwords parsing is resumed. In Tedir, this can be expressed using a *suspend* expression:

$$\langle\text{expr}\rangle \quad \rightarrow \quad \underline{\textbf{xml}}\ \underline{\{}\ \odot\ \underline{\}}$$

Actually, there is no ⊙ operator; it is a shorthand for a more general operator that will be described later. Until then, ⊙ should be considered as a real operator.

When the parser reaches the suspend expression, ⊙, it is suspended. Associated with each suspend expression is a *suspend action*, similar to the actions associated with the productions. After the parser has been suspended, it invokes the action. In the example of embedded XML, the action could then give over control to an XML parser that could parse the embedded XML. After that, the suspend action returns two values to the parser: one value specifying how to continue, either continue or replace, and a value that will be taken to be the value of the suspend expression. In the example, the action would return continue, which signifies that the parser should just continue parsing as before. The other possibility, replace, will be explained in section 3.11. As the value of the suspend expression, the action value could for instance return a representation of the embedded XML. Since the action returned continue, the parser rereads the current token from the scanner and continues to parse as usual, starting from immediately after the suspend expression.

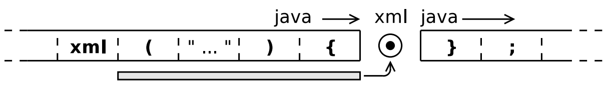
This way, a parser for an embedded language can be developed completely independently from the parser of the surrounding language. Suspend actions can be used for other purposes than parsing an embedded language. For instance, whenever a **typedef** is reached in C, the scanner must be notified that a new type name has been introduced. In this case, the suspend expression would be used to update the scanner.

3.8.1. Context-dependent suspend expressions

In some cases, the suspend actions need to have some information about the context in which they were used. For instance, imagine that we want to extend the embedded XML construct with the possibility of specifying a document type definition (DTD) for the XML:

XML hello = **xml (**“xhtml1-strict.dtd”**)** { Hello <i>world</i> };

In this case, it would be convenient if we could pass the name of the DTD to the XML parser, so that it can verify that the input conforms to the specified DTD, as it is being parsed. To do this, we need to be able to pass a part of the parsed input to the suspend action:

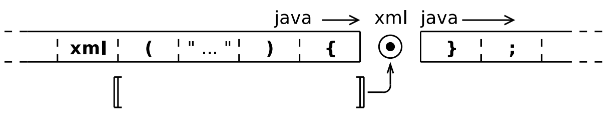


In this figure, the value of the underlined part of the input should be passed to the suspend action. To express this, it is not enough to specify that the parser should be suspended, it must also be specified which part of the input should be passed to the suspend action. This is done using the $[\![\varphi]\!]$ form:

$$\langle\text{expr}\rangle \quad \rightarrow \quad \underline{\textbf{xml}}\ [\![\ \underline{\textbf{(}}\ \textbf{string}\ \underline{\textbf{)}}\ \underline{\{}\ ]\!]\ \underline{\}}$$

This production might look confusing because it mixes expression-level and syntax-level delimiters. But if we compare this definition with the figure above, the left $[\![$ marks the beginning of the part of the input that must be passed to the suspend action, and the $]\!]$ marks the end, where the parser is suspended.

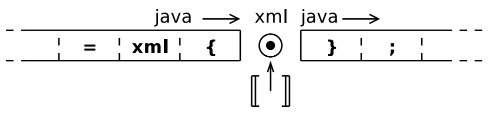
When this production is parsed, the suspend action is executed at the closing bracket and passed the value of the part of the input that is between the two brackets:



In general, a suspend expression must always specify the part of the input whose value should be passed to the suspend action. This means that in the first example, without the DTD, we must explicitly specify that the suspend action should not be passed any value. For uniformity, the suspend expression must have a body, so to specify that no value should be passed, the trivial ε expression is used:

$$\langle \mathsf{expr} \rangle \quad \rightarrow \quad \underline{\textbf{xml}}\ \underline{\{}\ [\![\varepsilon]\!]\ \underline{\}}$$

which is parsed essentially as



The form used in the first example, $\odot$, was actually just a shorthand for $[\![\varepsilon]\!]$. To sum up what has been so far been explained about suspend expressions: when a suspend expression is reached, the body is first parsed, and the value is then passed to the suspend action. The suspend action can then invoke another parser or manipulate the scanner or essentially do anything it wants with the input streams. Finally, the action can return a value, to be used as the value of the expression, and notify the parser that it should continue as before.

Why do we have two different ways of invoking user code, production actions and suspend actions? One reason is that the production actions are not allowed manipulate the input stream, so they are not allowed to invoke the parser for an embedded language or, in the C example, add new type names to the scanner. Another reason has to do with the model for how user actions are executed, which will be described in the next section.

3.9. Suspend and tentative parsing

At this point, you may have noticed a problem. In the section on ambiguities, we saw that when there were ambiguities, the parser first parsed the input tentatively and only afterwords reparsed it and invoked user actions. But how is the parser supposed to tentatively parse embedded XML without invoking the suspend action that invokes the XML parser? The answer is that is simply can't – suspend actions must be invoked while the input is being parsed tentatively. This is the real difference between production and suspend actions: with a suspend action, we can be sure that the parser has not read further ahead. With a production action, there is no guaranteed that the parser hasn't already tentatively parsed the rest of the input, and the production is being invoked during reparsing.

This brings up the same issue as with production actions: what if one path reaches a suspend expression, while another path doesn't? What if several paths reach different

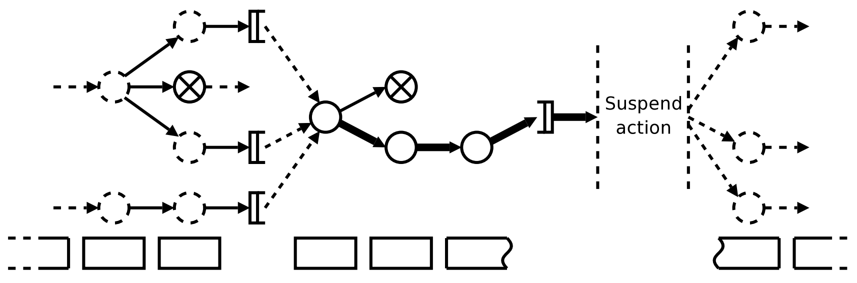


Figure 3.3.: Suspending the parser during tentative parsing

suspend expressions? Once a suspend action is invoked, it is allowed to have arbitrary side-effects and to and to consume as much input from the input stream as it wants. Once once an action has been invoked, we have essentially burned the bridge behind us: if the path turns out not to match the input, a suspend expression may have caused side-effects that cannot be undone.

The solution is to require that when one path reaches a suspend expression, *all* other paths must reach the same one. It is an error if only some threads reach a particular suspend expression. Before describing the implications of this, from the programmer's viewpoint, we will first see how a suspend expression is executed in tentative parsing mode.

Figure 3.3 illustrates what happens when the parser is suspended during tentative mode. When all threads have reached the same suspend expression they and are joined into a single thread. The body of the suspend expression is parsed as if there were only a single thread, and then the suspend action is invoked. When the suspend action returns, control is split up again and tentative parsing continues as before. In the example above, this means that if there is more than one thread when an XML expression is reached, the threads are collapsed and the XML is parsed only once.

Figure 3.4 on the facing page shows how the input is reparsed after disambiguation. When the correct path has been found and the input is reparsed, the value of the suspend expression will have been stored, and the body of the suspend expression will just be skipped. In the XML-in-Java example, this means that, since the XML was parsed during tentative mode, the Java parser will just skip over the XML when reparsing and continue from the **}**.

As with the model of tentative parsing, this affects the execution semantics of the parser. First of all, it means that the parser may generate an error if some threads reached a suspend expression and others didn't. It also means that it becomes harder to predict *when* the production actions are invoked, because some production actions may be executed during tentative parsing, while others will not be executed until reparsing. The next section describes how the programmer should deal with this.

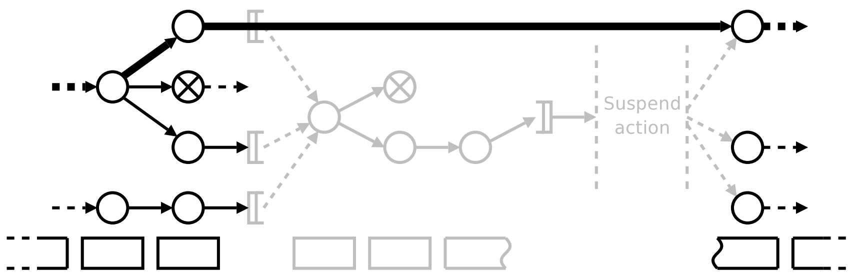


Figure 3.4.: Reparsing a suspend expression after disambiguation

3.9.1. Working with the semantics of suspend

As a programmer, it should not be necessary to know the details of how this scheme works. In general, the complications of these rules can be completely avoided if the user keeps a few simple guidelines in mind.

First of all, if the grammar contains suspend expression, the production actions should not depend on being executed in the same order as the corresponding syntax occurs in the input. All that can be assumed is that when a production action is invoked, the production actions of all subexpressions of that production have been invoked. Since parser actions are usually just concerned with constructing syntax trees, this will usually be a problem.

If some action *must* be executed at a certain point during parsing, that action should not be a production action but a suspend action. For instance, when a parser reaches a **typedef** in C, a new type name is introduced, which changes the way programs are parsed. Such an action cannot reliably implemented as a production action, but should be formulated as a suspend action.

The last guideline is that there should never be any ambiguities mixed with suspend expressions. One way of ensuring this is to make sure that each suspend expression is preceded by a unique preamble. In the XML example, the suspend expressions were preceded by **xml {**, and if this sentence does not occur anywhere else in the grammar, it is clear that we cannot have a situation where one thread reaches the suspend expression an another thread doesn't. This guideline is probably a good idea in general, since is is in general a good idea to make constructs stand out in the syntax if they change the way input is understood.

What was described above was how suspend expressions can for instance be used to parse embedded languages. The next sections explain how they can be used to extend a running parser. The first section describes how a grammar extension is defined.

3.10. Grammar extensions

Up until now I not mentioned anything about the extensibility aspect of the library. This section describes how grammar extensions are defined and applied.

3.10.1. "Delta", not "grammar"

In some parser tools, for instance ANTLR (Parr 2004), one grammar can be defined as an extension of another grammar. For instance, given the definition of the statement grammar from before

```
grammar SimpleStatement {
   ⟨stmt⟩[shortIf]  →  if ⟨expr⟩ then ⟨stmt⟩
         [longIf]   |  if ⟨expr⟩ then ⟨stmt⟩ else ⟨stmt⟩
                    ⋮
}
```

a new grammar could be defined as the extension of that grammar where a single production had been added:

```
grammar ForEachStatement extends SimpleStatement {
   ⟨stmt⟩[forEach]  →  for ( ⟨type⟩ id : ⟨expr⟩ ) ⟨stmt⟩
}
```

Note that the **grammar** ... **extends** ... syntax is not part of the specification language – it is only used here for illustrative purposes.

Grammar extension is often seen as being similar to object-oriented inheritance. With grammar extension, the extended grammar inherits the productions of the grammar it extends, and adds some productions of its own. With inheritance, the derived class inherits the members of the base class and adds some members of its own.

The weakness of this approach is that the extension is fixed to a single base grammar, which isn't a very good model for how we want to use grammar extensions. In Tedir, a grammar extension does not have to specify which grammar it extends. It is a *generic extension* that can be used to extend any grammar. Because of this, such a specification is called a *delta*.

```
delta ForEach extends • {
   ⟨stmt⟩[forEach]  →  for ( ⟨type⟩ id : ⟨expr⟩ ) ⟨stmt⟩
}
```

Using this model, a delta can be used to extend any grammar. A grammar is extended by *applying* the delta, as if it were a function. For instance, the ForEach extension can be applied to the grammar for simple statements, simpleStatement, to produce an extended grammar, extendedGrammar.

$$\mathsf{extendedGrammar} = \mathsf{ForEach}(\mathsf{simpleStatement})$$

Note that the expression above applied the delta to simpleStatement, with a lower-case s, instead of SimpleStatement. That was not a typo. So far, it might appear that there are two kinds of specifications: grammar specifications, like SimpleStatement, and delta specifications like ForEach. In reality, *all* specifications specify deltas, even SimpleStatement:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{delta}\ \mathsf{SimpleStatement}\ \textbf{extends}\ \bullet\ \{} \\
\quad\langle\mathsf{stmt}\rangle[\mathsf{shortIf}] & \rightarrow & \textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \langle\mathsf{stmt}\rangle \\
\qquad\qquad[\mathsf{longIf}] & | & \textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \langle\mathsf{stmt}\rangle\ \textbf{else}\ \langle\mathsf{stmt}\rangle \\
 & \vdots & \\
\}
\end{array}
$$

Of course, SimpleStatement is not meant to be an extension of anything. It can, however, be thought of as an extension anyway, namely the trivial extension of the empty grammar:

$$\mathsf{simpleStatement} = \mathsf{SimpleStatement}(\emptyset)$$

This is why we used simpleStatement before, instead of SimpleStatement: a delta must be applied to a grammar, not a delta, and SimpleStatement is a delta. In Tedir, a grammar is not something you specify, it is something that is constructed by applying a delta to an existing grammar, possibly the empty grammar $\emptyset$. Actually, in Tedir, applying a delta to a grammar not only produces a new grammar, it also produces a parser for the grammar.

Note that if we consider the object-oriented metaphor for grammar extension from before, the generic extension model can be seen as a generalization of the traditional grammar extension model in almost exactly the same way that the concept of mixins (Bracha & Cook 1990) generalizes traditional object-oriented inheritance.

3.10.2. Directives

Now that we have established what a delta is, we can review the anatomy of a delta specification. A delta specification is a sequence of directives that specify changes to apply to a grammar. The ForEach example specifies that a single production should be added to the ⟨stmt⟩ nonterminal, and the original specification of the ⟨stmt⟩ nonterminal in figure 3.1 on page 17 contains five directives, each adding a production to the ⟨stmt⟩ nonterminal. A single add-directive has the form

$$\langle\mathsf{nonterm}\rangle[n] \quad \rightarrow \quad \varphi$$

where φ is the body of the production and n is the name. The name can be used to override an existing production: if the production is added to a nonterminal that already has a production named n, that production is replaced.

A sequence of add-directives adding to the same nonterminal can be abbreviated slightly, so that

$$\begin{array}{rcl} \langle\textsf{nonterm}\rangle[n_1] & \rightarrow & \varphi_1 \\ \langle\textsf{nonterm}\rangle[n_2] & \rightarrow & \varphi_2 \\ & \vdots & \end{array}$$

can be written as

$$\begin{array}{rcl} \langle\textsf{nonterm}\rangle[n_1] & \rightarrow & \varphi_1 \\ [n_2] & | & \varphi_2 \\ & \vdots & \end{array}$$

Besides add-directives, another kind of directive can occur in a delta: the **drop** directive. It can be used to remove productions from a nonterminal:

$$\textsf{\textbf{drop}}\ \langle\textsf{nonterm}\rangle\ n_1\ n_2\ \ldots\ n_k$$

This specifies that the when the delta is applied, the productions of nonterminal $\langle\textsf{nonterm}\rangle$ with names $n_1, \ldots, n_k$ should be removed from the grammar it is applied to.

3.10.3. Delta composition

We have already seen one operation that can be performed on a delta: it can be applied to a grammar to produce a new, extended grammar. Deltas can also be composed, to form a new delta. If A and B are deltas, then we can form the composition A $\oplus$ B. The composition of A and B is a delta that when applied corresponds to first applying A and then B. Because productions can override each other, A $\oplus$ B may not be the same as B $\oplus$ A if both A and B override the same production.

Delta composition can be useful if A and B are separate modules of the same grammar, so that A refers to nonterminals defined in B and vice versa. In this situation the result of applying either A or B directly to a grammar would not be well-formed, because it would be missing the definitions from the other. Instead, A and B have to first be composed and then applied to a grammar.

3.10.4. Parsers and "extensible" parsers

Tedir claims to produce "dynamically extensible parsers", which could be taken to mean that you can take a generated parser and then somehow modify it to be able to parse according to an extended grammar. The model used here is somewhat stronger than that, and to illustrate why, I will first explain why a model that actively modifies a parser is weak.

Consider the example from the introduction, where a language had a construct for enabling an extension locally. The example was that only a small part of the program needed to use regular expressions, so the constructs for regular expressions should just be enabled for a small part of the code:

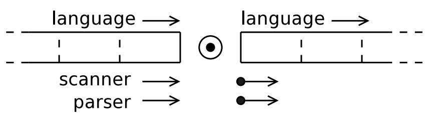
```
syntax (java.util.regex.Regex) {
    if (line ~= /[0-9]+:[0-9]+/) {
        /* ... whatever ... */
    }
}
```

In a model where the parser was extended destructively, this construct would be parsed by first extending the parser, at the beginning of the body of the **syntax** statement. Then, after parsing the body, the extension should be "turned off", which means that the extension should be undone and the parser changed back to the way it was before. If the regular expression syntax is enabled more than once, we have to do all the work of extending the parser and undoing the changes, using the same extension, each time the extension is used.

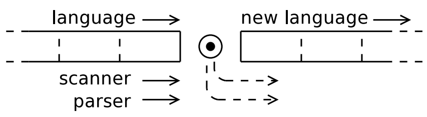
In Tedir, a parser is immutable and cannot be extended. Instead of extending the parser, the library allows a new and extended parser to be produced efficiently. Constructing the extended parser can share much of the structure of the existing parser, and is much more efficient than generating a parser from scratch. This library allows the extended parser to be substituted for the current parser, while the input is being parsed. This way, an extension must only be generated once and can then be reused. On top of this, the operation of replacing the current parser is very efficient, which means that once an extended parser has been generated, using it to enable local syntax extensions is essentially free.

3.11. Dynamic extension

In the first examples of parser suspension, we used the suspend actions to invoke a parser for a different, embedded language. In particular, the parser itself was only suspended, not changed. This is because the suspend actions returned `continue`, which told the parser to just continue parsing as before:



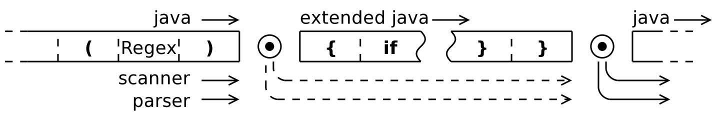
The Java language used before and after parsing the embedded XML, for instance, is the same. The alternative to is that the suspend action can specify that hereafter, a new scanner and parser should be used:



This allows the parser to be extended dynamically. Consider the example from before, with locally scoped language extensions:

```
syntax (java.util.regex.Regex) {
    if (line ~= /[0-9]+:[0-9]+/) {
        /* ... whatever ... */
    }
}
```

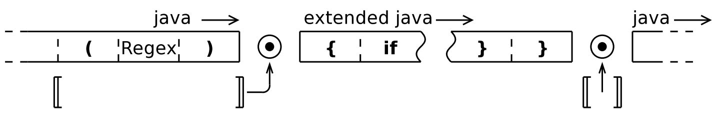
The **syntax** statement enables a named extension, in this case a Perl-like regular expression construct, in the body of the statement. To be able to parse this, the parser must be suspended after reading the name of the macro, extended with the specified syntax, parse a single statement, and then continue parsing with the original parser. This can be implemented using two suspend statements, the first enabling the syntax and the second disabling it again:



The definition of the **syntax** statement is this:

$$\langle\text{stmt}\rangle \quad \rightarrow \quad \underline{\textbf{syntax}}\ [\![\underline{\textbf{(}}\ \textbf{id}\ \underline{\textbf{)}}]\!]\ \langle\text{stmt}\rangle\ [\![\varepsilon]\!]$$

which causes the example above to be parsed like this:



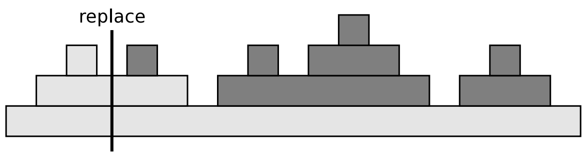


Figure 3.5.: Diagram of the effect of parser replacement

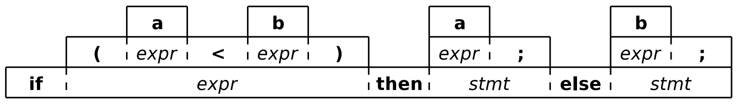
The production works as follows: the parser is first reads the name of the macro and then invokes the first suspend action, passing in the name. The suspend action can then use the name to obtain a delta describing the macro's syntax. By applying this delta to the grammar of the current parser, it obtains the parser to use for the body of the statement. Alternatively, if this extension has been used before, the action can look up an existing extended parser. Either way, the action can return replace, together with the extended parser and possibly also an extended scanner. The parser then continues to parse the ⟨stmt⟩ where it left off, but now using the extended parser and scanner that were returned by the suspend action. After the ⟨stmt⟩ has been parsed, using the extended parser, the parser is suspended again, so that the extension can be "turned off". The second suspend action simply returns replace, specifying that from that point on, the original parser and scanner should be used.

When replacing the current parser, the replacement must, either directly or indirectly, be a derived or base parser for the current parser.

3.11.1. Semantics of parser replacement

How exactly does replacing the current parser affect the way input is parsed? For instance, if we replace the parser in the middle of parsing a statement, what happens if the replacement has a different definition of that statement?

The easies way to explain the effect of parser replacement is probably to represent the parsing process as a level diagram. The following example shows the process of parsing the statement **if (a < b) then a; else b;**:

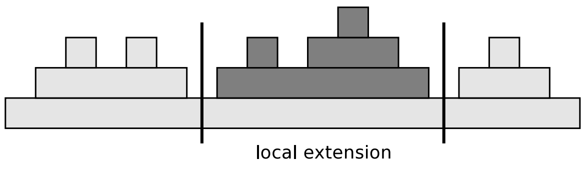


When scanning from left to right in this diagram, going up a level corresponds to starting to parse a nonterminal occurring as a subexpression of the current level.

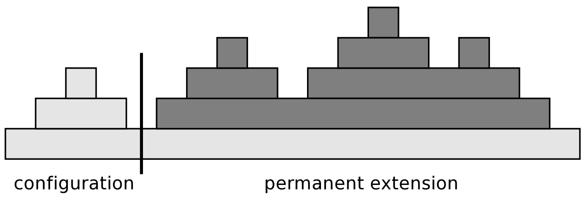
Figure 3.5 shows the effect of replacing the parser. The light levels are parsed using the original parser, and the dark levels are parsed using the new parser. If the parser has started parsing according to some nonterminal, replacing the parser will not affect how the rest of that nonterminal is parsed. In the example where the parser is replaced while it is in the middle of parsing a statement, and the replacement parser has a different definition of statements, the parser will simply continue to parse that statement as if nothing had happened. Only statements that start after the point where the parser was replaced are affected by the replacement.

This semantics is somewhat irregular. For instance, in the diagram, the replacement is made in some sub-term at level 2, but affects the way input is parsed non-locally, on level 1. The problem there is not, however, the replacement mechanism itself, but more the way it is used. The following two examples demonstrate how the extension mechanism can be used to get a more regular replacement semantics.

One way to use the extension mechanism is to enable it locally, within a single subterm. This corresponds to the local macro example above. In this case, we have to make sure that the extension is disabled after the subterm has been parsed, or else the extension will "leak out" and affect the way the rest of the input is parsed:



Another example is the grammar library example from the introduction where the parser first reads a specification of how the syntax should be extended, and the extends the syntax permanently:



Here, we must make sure that the configuration part and the body of the input are represented as two separate nonterminals, so that we are sure that the extension affects the whole body, and that there are no terms after the body that can be affected.

To finish off this chapter, we have some unfinished business. Until now, I have tried to justify all the design decisions made and explain their consequences. The one design decision I have not yet discussed is the most important one: the choice of the underlying algorithm. That is the subject of the last section.

3.12. Top-down, breadth first parsing

Until now, I have treated the fact that the parsing algorithm was based on top-down breadth-first parsing as if it was a given. This section motivates that choice.

3.12.1. Top-down vs. bottom-up

In section 2 we saw the difference between top-down and bottom-up parsing. Top-down parsing works by guessing a production and then parsing according to it. The hard part there is to select the right production – the production itself can be arbitrarily complex (within reason, of course). For instance, the productions can contain actions that must be executed as the productions are being parsed.

Bottom-up parsing works by first reading the input and then afterwords recognizing which production it parsed. This puts a restriction on how the productions can be written, since it must be possible to recognize them after they have been read, which for instance means that using EBNF directly with a bottom-up parser is impossible. It also makes it impossible to execute actions while the productions are being parsed, because the parser might not know which production it is parsing until after it has reached the end of it. In essence, top-down parsers are much easier to generalize and extend than bottom-up parsers.

Top-down parsers also have another advantage over bottom-up parsers: the internal structure of a top-down parser is very similar to its grammar, which means that a local change to the grammar is likely to only require a relatively local change to the parser. That is something that will become important when we have to efficiently generate parsers for locally extended grammars.

As mentioned, it is always possible to translate a grammar in EBNF into a grammar in BNF. This means that it would, in theory, be possible to allow EBNF with a bottom-up parser, by having a preprocessing step that translated an EBNF grammar into BNF. One problem with this is that this would make it much harder to report errors in terms of the original grammar. To be meaningful to the user, errors should be reported in terms of the EBNF grammar, not the BNF grammar used internally by the parser. This means that errors would have to be translated "backwards" from the BNF representation into EBNF. And using a translation scheme makes the structure of the parser even more dissimilar from the specified grammar, which is something we will generally try to avoid.

3.12.2. Breadth-first vs. depth-first

The second choice is between using a depth-first or breadth-first top-down parser. When a depth-first parser reaches a local ambiguity, is simply continues parsing according to one of the possible choices. Later, if that fails, it backtracks and tries the next choice. This model is simpler, because there is only a single thread of execution. It can also be more efficient because there is a chance that it will actually make the right choice, in which case it will never get to parse the other alternatives.

Tedir uses breadth-first parsing, where a local ambiguity causes the parser to try to parse the input in several different ways in parallel. There are two reasons for using breadth-first instead of depth-first parsing. The first reason is that just finding *some* way to parse the input is not good enough; here, we require that is exactly one way to parse the input. This means that the parser must explore all alternatives, because it must make sure that there is a unique parse tree. If there is more than one way to parse a given input, this is typically a symptom of a bug in the grammar, and something that should be noticed and reported to the programmer.

The second reason is that production actions should only be executed when all ambiguities have been resolved. Actions can have side-effects, and if an action is executed in a thread that turns out not to parse the input, there is no way to “backtrack” the execution of that action. With breadth-first parsing, it is easy to see when there is several ways to parse the input, because then there will be more than one thread. This means that the parser can notice as soon as there is only a single thread left and then execute the production actions. In depth-first mode, there is always just one thread, and before all the other threads have been executed, there is no way of knowing if this thread represents the only way of parsing the input.

This concludes the description of the basic design of the library. By now, I hope it is clear that this a library implementing this design can be used to solve the problems from the introduction. The next chapter describes a concrete implementation of this design, in Java.

4. The Java Library

The previous chapter gave an abstract, language neutral, description of the library. Back then, some issues were dismissed as being language specific and irrelevant to the underlying design. This chapter describes a concrete implementation of the library, in the Java programming language. A proof-of-concept prototype implementation of this design has been developed and is described in appendix A on page 115. The design makes use of many of the features added to the 1.5 version of the Java language. These features will be described as they are used.

The first sections of this chapter describe the Java library, followed by an evaluation of the library as a whole. The last section discusses typing issues in relation with a library implementation.

4.1. Specifying deltas

As described in section 3.10.2, a delta is a sequence of directives, adding or dropping named productions. In the Java library, a delta is specified by instantiating a Delta object and invoking add and drop methods. For instance, the statement grammar,

$$\begin{array}{lcl}
\langle\mathsf{stmt}\rangle & \rightarrow & \textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \langle\mathsf{stmt}\rangle \\
 & \mid & \textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \langle\mathsf{stmt}\rangle\ \textbf{else}\ \langle\mathsf{stmt}\rangle \\
 & \mid & \textbf{while}\ \langle\mathsf{expr}\rangle\ \textbf{do}\ \langle\mathsf{stmt}\rangle \\
 & \mid & \{\ \langle\mathsf{stmt}\rangle^*\ \} \\
 & \mid & \langle\mathsf{expr}\rangle\ \textbf{;}
\end{array}$$

would be expressed through calls to addProd:

```
Delta delta = new Delta();
delta.addProd("stmt", term("if"), nonterm("expr"), term("then"), nonterm("stmt"));
delta.addProd("stmt", term("if"), nonterm("expr"), term("then"), nonterm("stmt"),
    term("else"), nonterm("stmt"));
delta.addProd("stmt", term("while"), nonterm("expr"), term("do") nonterm("stmt"));
delta.addProd("stmt", term("{"), star(nonterm("stmt")), term("}"));
delta.addProd("stmt", nonterm("expr"), term(";"));
```

There are a number of different addProd methods available, for adding a production either with or without names and with or without tags. Production tags will be described

Syntax (φ)	**Java** (t)	**Description**
term	value(*name*)	*relevant terminal symbol*
<u>term</u>	term(*name*)	*irrelevant terminal symbol*
$\langle$nonterm$\rangle$	nonterm(*name*)	*nonterminal*
ε	empty()	*empty*
$\varphi_1\,\varphi_2\,\cdots\,\varphi_n$	seq($t_1, t_2, \ldots, t_n$)	*sequencing*
$\varphi_1 \mid \varphi_2 \mid \cdots \mid \varphi_n$	choice($t_1, t_2, \ldots, t_n$)	*choice*
φ^*	star(t)	*zero or more repetitions*
φ^+	plus(t)	*one or more repetitions*
$[\varphi]$	option(t)	*optional*
$\{\varphi_1 : \varphi_2\}^*$	starInfix(t_1, t_2)	*zero or more infix repetitions*
$\{\varphi_1 : \varphi_2\}^+$	plusInfix(t_1, t_2)	*one or more infix repetitions*
$\underline{\varphi}$	ignore(t)	*ignore*
$[\![\varphi]\!]$	suspend(t, ...)	*suspend*

Figure 4.1.: Expressions in the Java binding of Tedir

in section 4.4. In the delta constructed above, none of the productions have a name or a tag.

In the example, the name of the nonterminal is specified as a string. In general, any object can be used as the name of a nonterminal. The body of a production is specified as the last arguments to the method. Expressions are constructed by calling constructor methods; for instance, the expression $\langle$stmt$\rangle^*$ is constructed as star(nonterm("stmt")). The list of constructor methods is shown in figure 4.1.

If more than one expression is given at the end of an addProd, as in the example, those expressions are considered to be a sequence, so

```
delta.addProd("stmt", term("{"), star(nonterm("stmt")), term("}"));
```

means the same as

```
delta.addProd("stmt", seq(term("{"), star(nonterm("stmt")), term("}")));
```

The possibility of calling a method with a variable number of arguments is well-known from C and C++ but is a new feature in Java.

Note that all the terminals used in the delta are specified using the term(...) constructor, which marks them as irrelevant. Because of this shorthand, and the fact that irrelevance propagates upwards through composite expressions, it is hardly ever necessary to explicitly mark expressions as irrelevant using the ignore(...) constructor.

4.2. Operations on deltas

The two operations that can be performed on a delta, application and composition, were described in section 3.10.1. The Java implementation has three operations because the application of a delta to the empty grammar is considered a separate operation.

Deltas can be composed with the add method, which produces a new delta:

```
Delta fullSyntax = baseSyntax.add(oracleOuterJoinSyntax).add(postgresqlLimitOffsetSyntax);
```

A delta can be applied to the empty grammar by using the compile method. The compile method takes a single argument, a *token manager*, which has to do with the parser's interface with the scanner and will not be described until later. It returns an Grammar object:

```
TokenManager tokens = ...;
Grammar javaGrammar = javaDelta.compile(tokens);
Parser javaParser = javaGrammar.getParser();
```

The parser for the grammar can be obtained by calling the grammar's getParser method. A delta can be applied to a grammar by using the apply method:

```
Grammar extendedJavaGrammar = forEach.apply(javaGrammar, tokens);
```

The generated Parser can be invoked by calling the parse method. Besides the name of the start nonterminal and a scanner, this method accepts three other arguments that define the behavior of the parser: a *data constructor*, a *parse event handler* and a *suspend handler*. These will all be described in the following sections.

4.3. Expression values

In the description of expression values given in section 3.5, three constructors were used when forming the value of an expression: the value of repetition expression was constructed as a list, the value of a sequence was (in some cases) constructed as a tuple, and the value of an irrelevant expression was taken to be a special null value. In the Java implementation, all these data constructors can be configured by the programmer. They are specified through the *data constructor* object, which, as mentioned, is passed to the parser when it is invoked. A data constructor is an object that implements the DataConstructor interface:

```
public interface DataConstructor {
    public Object nothing();
    public Object emptyList();
    public Object append(Object list, Object value);
    public Object tuple(Object[] vals);
}
```

Whenever the parser needs the null value, it will call the nothing method on the data constructor. The emptyList and append methods define how lists are constructed, and the tuple method can be used to specify how tuples are constructed. This way, the user has full control over which data types are used to represent the parsed expressions.

For convenience, the library provides a default data constructor, ListConstructor, that uses Java's null as the null value, represents tuples as arrays, and can use any class that implements java.util.List to construct lists. For instance, a ListConstructor that represent lists as LinkedLists would be constructed like this:

```
DataConstructor dc = new ListConstructor(LinkedList.class);
```

The purpose of all this is to make the representation of the parsed expression as convenient as possible for the production actions. Production actions are the subject of the next section.

4.4. Actions

In the Java library, actions are associated with productions through two mechanisms: a *parse event handler* and a *production tags*. Each production can have an associated tag, which can be any object. Whenever a production has been parsed, the parse event handler is notified, and passed the tag of the production that has been recognized. The parse event handler is a callback object implementing the ParseEventHandler interface that is passed to the parser when it is invoked.

```
public interface ParseEventHandler {
    public Object parseEvent(Object nonterm, Object tag, Object data);
}
```

When a production has been recognized, the parseEvent method is invoked on the parse event handler, and given three arguments: the name of the nonterminal of the production, the tag of the production, and the value of the parsed expression. Note that the tag and the name of the production are two different things, and the name of the production is not automatically available to the event handler.

It is also possible to not specify a tag. In that case, the parse event handler is not invoked, and the value of the nonterminal is taken to be the value of the parsed expression. This can sometimes be convenient, especially with simple productions.

This interface is inspired by the SAX (*simple API for XML*) interface to XML parsing. As a SAX parser parses an XML document, it notifies the user of *parse events*, for instance the start or end of an element, by calling methods on a DocumentHandler object. There are two main differences between the two. First of all, in SAX, there is a fixed number of possible parse events, and the DocumentHandler interface has a method for each kind of event. In Tedir, each production introduces a new parse event, so Tedir only has a single event handling method, but adds tags to the production. This is not only a consequence of the extensibility, but simply because the parsers are constructed at runtime, so there is no way to statically determine the set of productions in a grammar.

The second difference is that in Tedir, the event handler returns a value to the parser, which is carried around by the parser and can be used by it to construct the value of other expressions. In SAX, it is up to the user to keep track of constructed values, for instance using a stack. In SAX, automatically constructing the parse tree of an XML document can be a bad idea because the user will, in many cases, only be interested in a small part of a large body of XML, and building a representation of the whole XML document is expensive or infeasible. In Tedir, the user will typically be interested in the value of the whole document, so it is reasonable that the parser supports the construction of that value.

This interface might look a bit convoluted and harder to use than the model described earlier, where each production had an associated action, a function, that was invoked when the production had been parsed. Here we only have a single "function", the event handler, and what is associated with each production is just dumb data. But a more convenient model can easily be built on top of this model. The following sections give three examples of how this interface can be used. All three examples parse and construct abstract syntax trees for a simple λ-calculus syntax:

$$
\begin{array}{rcl}
\langle\mathsf{expr}\rangle & \to & \textbf{id} \\
 & | & \underline{\boldsymbol{\lambda}}\ \textbf{id}\ \underline{\textbf{.}}\ \langle\mathsf{expr}\rangle \\
 & | & \underline{\textbf{(}}\ \langle\mathsf{expr}\rangle\ \langle\mathsf{expr}\rangle\ \underline{\textbf{)}}
\end{array}
$$

where, in accordance with tradition, the $\boldsymbol{\lambda}$ is represented by a \.

4.4.1. Enumerated tags

The simplest model is to let the tag of each production be the value of some enumeration that can be distinguished in the event handler. An example of this is shown in figure 4.2 on the next page, using the new enumeration construct from Java 1.5.

Besides the inconvenience of having to do so many casts go get access to the value of the parsed expressions, the main weakness of this approach is that the grammar cannot be extended, because the LambdaHandler is statically defined to only handle these three cases. The main advantage of this approach is that it is so straightforward, especially when there is only a few productions.

In order to make the parser extensible, the action should be associated directly with the production, not specified separately in the handler. That is the approach used in the next example.

```
enum LambdaTag { VAR, LAM, APP };

Delta delta = new Delta();
delta.addTagProd("expr", VAR, value("id"));
delta.addTagProd("expr", LAM, term("\"), value("id"), term("."), nonterm("expr"));
delta.addTagProd("expr", APP, term("("), nonterm("expr"), nonterm("expr"), term(")"));

public class LambdaHandler implements ParseEventHandler {
    public Object parseEvent(Object nonterm, Object tag, Object data) {
        switch ((LambdaTag) tag) {
            case VAR:
                return new Variable((String) data);
            case LAM: {
                Object[] tuple = (Object[]) data;
                return new Lambda((String) tuple[0], (Expr) tuple[1]);
            }
            case APP: {
                Object[] tuple = (Object[]) data;
                return new Application((Expr) tuple[0], (Expr) tuple[1]);
            }
        }
    }
}
```

Figure 4.2.: λ-calculus parser using enumerated tags

4.4.2. Callback tags

This example uses callback objects as tags to give a behavior similar to the function-as-action scheme described in the previous chapter. In this model, the tag associated with each production implements the ProductionAction interface:

```
public interface ProductionAction {
    public Object invoke(Object data);
}
```

The parse event handler then simply invokes the tag, passing it the value of the parsed input:

```
public class ProductionActionParseEventHandler {
    public Object parseEvent(Object nonterm, Object tag, Object data) {
        return ((ProductionAction) tag).invoke(data);
    }
}
```

An implementation of the λ-calculus example using this approach is shown in figure 4.3 on the following page. This approach has the advantage the appropriate action is specified in the production tag itself, which makes it easy to add new productions that carry their own actions with them. The main disadvantage is that this approach is very verbose, and that the actions still have to do a lot of casts to get access to the expression values. That is, however, only a problem when specifying a parser "by hand". In the situations the library was mainly designed to handle, where the grammar and actions are generated at runtime, this interface is preferable to the previous and the next one. Their advantages are mostly related to them being convenient to use when specifying parser that do not change dynamically.

4.4.3. Reflection

The last approach gives an interface similar to SAX, where the parse events cause different methods to be invoked on a single object. It also allows a limited form of extension through ordinary class inheritance.

Java's reflective API makes it possible to inspect the layout of an object at runtime. For instance, it can be used to obtain a list of all the object's methods. A method is represented by an object of type java.lang.reflect.Method, and this object can be used to invoke the corresponding method on an object. That means that if the production tags are Method objects, the parse event handler can reflectively invoke different methods on a particular object.

In order to do this, we need to associate a method with a particular production. This is done by using *annotation types*. An annotation can for instance be used to associate

```
Delta delta = new Delta;

ProductionAction var = new ProductionAction() {
    public Object invoke(Object data) {
        return new Variable((String) data);
    }
}

ProductionAction lam = new ProductionAction() {
    public Object invoke(Object data) {
        Object[] tuple = (Object[]) data;
        return new Lambda((String) data[0], (Expr) data[1]);
    }
}

ProductionAction app = new ProductionAction() {
    public Object invoke(Object data) {
        Object[] tuple = (Object[]) data;
        return new Application((Expr) data[0], (Expr) data[1]);
    }
}

delta.addTagProd("expr", var, value("id"));
delta.addTagProd("expr", lam, term("\"), value("id"), term("."), nonterm("expr"));
delta.addTagProd("expr", app, term("("), nonterm("expr"), nonterm("expr"), term(")"));
```

Figure 4.3.: λ-calculus parser using production actions

```
class LambdaGrammar {
    @Prod("<expr> -> $id")
    Object variable(String var) {
        return new Variable(var);
    }
    @Prod("<expr> -> '\' $id '.'  <expr>")
    Object lambda(String var, Expr body) {
        return new Lambda(var, body);
    }
    @Prod("<expr> -> '(' <expr> <expr> ')'")
    Object application(Expr rator, Expr rand) {
        return new Application(rator, rand);
    }
}

Delta delta = makeDelta(LambdaGrammar.class);
```

Figure 4.4.: λ-calculus parser using reflection

information with methods and classes. Figure 4.4 shows how the @Prod annotation is used to associate strings, in this case representing productions, with the methods of a class.

Using the reflective API, a delta can be constructed from a class such as Lambda-Grammar by iterating through the list of its methods. The parse event handler and the method that constructs the delta are given in appendix B on page 119. For each method, the string representation of the associated production can be accessed through the @Prod annotation. This representation is parsed, using the library itself, and added to a delta, with the Method object as tag. When the parser is invoked, an *action object* is specified, and each time a parse event occurs, the parse event handler reflectively invokes the corresponding method on the action object, passing it value of the body of the production.

This interface has the advantage that the productions can be written straightforwardly, as strings, and that we get rid of all the casts from the two first examples. Instead of casting a value of type Object to get access to the components of the expression's value, the type of the parsed input can be encoded in the types of the arguments of the associated method. The casting in handled automatically by the reflective method invocation.

Another advantage is that we can use ordinary inheritance to extend the parser. This works because the parser generation process is based on enumerating the methods of an object, and a subclass inherits the methods of its superclass. An example is shown in figure 4.5 on the next page.

```
class BooleanLambdaGrammar extends LambdaGrammar {
    @Prod("<expr> -> 'true'")
    Object trueExpr() {
        return new Lambda("t", new Lambda("f", new Variable("t")));
    }
    @Prod("<expr> -> 'false'")
    Object falseExpr() {
        return new Lambda("t", new Lambda("f", new Variable("f")));
    }
    @Prod("<expr> -> 'if' <expr> 'then' <expr> 'else' <expr>")
    Object ifThenElse(Expr cond, Expr thenPart, Expr elsePart) {
        return new Application(new Application(cond, thenPart), elsePart);
    }
}

Delta delta = makeDelta(BooleanLambdaGrammar.class);
```

Figure 4.5.: λ-calculus parser extended with boolean syntax

4.5. Scanner interface

The description of the parser's interface to the scanner given in section 3.6 stated that the parser could work together with any scanner, if it conformed to a simple interface. In the Java library, this interface is split into two: the Scanner interface and the TokenManager interface.

4.5.1. Interfaces

The parser can parse input produced by any object that implements the Scanner interface:

```
public interface Scanner {
    public int getNextToken();
    public int getCurrentToken();
    public Object getTokenValue();
}
```

As this interface shows, tokens are represented as (non-negative) integers. The examples above of how deltas were constructed all referred to tokens by their names, as strings, not integer values. The token manager is the mapping used to get from the name of a token to its value:

```
public interface TokenManager {
    public int getTokenIndex(Object name);
    public int getEOF();
}
```

Whenever the library needs to know which index corresponds to a given name, it will call the getTokenIndex method on the TokenManager. If the scanner has been constructed in advance, it will probably have a table that gives, to the name of each of its tokens, the index of the token. Another possibility is that this can be used to extend the scanner, or generate a new extended scanner. Each time the parser requests the index of a token, the token manager inspects the scanner to see if a token with that name exists. If not, it extends the scanner, for instance with a new keyword.

Besides mapping token names to integers, the token manager also specifies which token is the end-of-file marker.

The division between the scanner and the token manager has a simple reason: they are needed in two different contexts. The token manager is used when a delta is applied to a grammar, whereas the scanner is used when the generated parser is invoked.

4.5.2. Integer tokens

The choice to represent tokens as integers is purely a question of efficiency. The implementation of the parser needs to be able to use tokens as indices in tables, and the most efficient lookup mechanism in Java is to use arrays with integer indices.

It is important to note that this does not mean that the programmer using the library is forced to represent tokens as integers. If it is convenient or if speed matters, this representation makes it possible to get maximum performance out of the parser. If it is more convenient to use a scanner that represents tokens as objects, a wrapper class can be used to adapt the interface of the scanner.

In essence, the integer interface makes it possible to get maximum performance, while leaving the user free to build higher levels of abstraction if that is convenient.

4.6. Suspending the parser

The interface to suspending the parser is very similar to the production action interface. It has two elements: a *suspend handler* and *suspend tags*. As with the parse event handler, the suspend handler is passed to the parser when it is invoked. A suspend handler is an object that implements the SuspendHandler interface:

```
public interface SuspendHandler {
    public SuspendResult suspend(Object tag, Object data);
}
```

Associated with each suspend expression is a tag (in figure 4.1 the tag was hidden behind the "..."). When the parser is suspended, the suspend handler is invoked and passed the tag of the suspend expression and the value constructed for the body.

When the suspend handler returns, it must return a SuspendResult. There are two methods for constructing SuspendResults, corresponding to the two modes of returning, continue and replace:

```
public static SuspendResult proceed(Object value);
public static SuspendResult replace(Object value, Parser parser, Scanner scanner);
```

Unfortunately, the word continue is a reserved word in Java, so the name proceed is used for the continue method.

4.7. Evaluation

Is this a convenient interface for specifying parsers? This question corresponds to the question of "are abstract syntax tree constructors a convenient interface for specifying programs?". The answer is that it depends on the application.

When writing a program, it is much more convenient to be able to say

$$\textbf{if a} < \textbf{b then a else b}$$

than using abstract syntax constructors

```
conditional(binop(LT, var(a), var(b)), var(a), var(b))
```

On the other hand, when manipulating the program, for instance in a compiler, specifying programs through abstract syntax is much more convenient than concrete syntax. It is the same with this library. The library is designed to be useful for manipulating grammars and constructing parsers. For that purpose, which is by far the most important, since that is really the core functionality of the library, I believe that the design is appropriate and convenient. In particular, the interface enables the user to build more convenient abstractions using the base functionality of the library.

Some parts of the library are directed towards making it convenient to specify parsers "by hand", specifically the production-as-annotation interface described in section 4.4.3. Still, to take full advantage of the extensibility of the library when specifying parser by hand, the callback-tag interface should probably be used, and while using this interface might be convenient when manipulating parsers dynamically, using directly is inconvenient at best. To make the library easier to use for specifying grammars by hand, it is probably necessary to build another layer of functionality on top of the base layer described here. In this respect, the library has an important advantage over the interfaces used by most other tools: since the abstraction layers are built on top of the library, rather than being part of the core functionality, these interfaces can be developed freely and independently. This means that it is quite possible to not only build a convenient interface, but to build several different interfaces that will be compatible at the base level.

4.8. Typing issues

An interesting issue that has not been considered so far in this description is the question of typing. There are two aspects to this: what can be expressed about the library in Java's type system, and what can be said about the types of expression values.

4.8.1. Generifying the library

The library, as described here, only makes limited use of Java's type system. Many aspects of the library can actually be expressed in the type system and type checked statically. For instance, if we know that all production tags used in a parser are of type ProductionAction, then we should be able to use this information in the definition of ParseEventHandler. This can be expressed using the genericity mechanism that has been added to Java in version 1.5:

```
public class Parser⟨Tag⟩ {
    public Object parse(Object start, ParseEventHandler⟨Tag⟩ handler, ...);
    ...
}

public interface ParseEventHandler⟨Tag⟩ {
    public Object parseEvent(Object nonterm, Tag tag, Object data);
}
```

We can go further than this. If we know that all the names used for nonterminals are strings, we know that the name of the start nonterminal, which is given to the parser when it is invoked, must be a string. We also know that the first argument given to the parse event handler, the name of the recognized nonterminal, is a string. That makes for another type parameter:

```
public class Parser⟨Nonterm, Tag⟩ {
    public Object parse(Nonterm start, ParseEventHandler⟨Nonterm, Tag⟩ handler, ...);
    ...
}

public interface ParseEventHandler⟨Nonterm, Tag⟩ {
    public Object parseEvent(Nonterm nonterm, Tag tag, Object data);
}
```

When the data constructor, the suspend actions and the tokens values are brought into this, there many aspects of the library can be expressed through Java's type system. But trying to do so means that every class end up having at least five or six different type parameters, which quickly becomes a major burden. Finding the right balance here is hard, and is probably something that requires more practical experience with using the library.

4.8.2. Expression types

The rules used to construct expressions during parsing are essentially untyped. The choice rule, for instance, says that the value of a choice between expressions will be one of the expressions. This means that the most precise type we can give for a choice expression is the most specific common supertype of all the subexpressions. In Java all object types have a common supertype, `Object`, but in ML for instance, there can be situations where the types of the subterms of the choice expression are incompatible. In that situation, choices must be restricted to only being between expressions of the same type.

In the Java implementation, this issue has been completely ignored – all expressions are considered to be of type `Object`. This also has to do with the fact that the grammars can be defined at runtime, which means that static typing of the expressions will in general be impossible.

This concludes the description of the library's interface. In the following sections, I will focus on the algorithms that are used to implement the interfaces described in this and the previous chapter.

5. Compilation

The previous two chapters described the design of the library interface. This and the next chapter are concerned with the implementation of that design. This first chapter describes how a parser is generated from a specification, and the next chapter describes how the generated parser is executed.

5.1. Introduction

As described in chapter 3, a parser is produced by applying a delta to a grammar, the *base* grammar, to obtain a *derived* grammar and a parser for that grammar. The compilation process is the process of constructing the derived grammar and parser.

Incremental parser generation algorithms exist for LR parsers, but these algorithms only work for limited updates (adding productions but not modifying or removing them) and extend a parser by actively modifying it. The algorithm presented here is, as far as I have been able to determine, the first that produces LL parsers, and the first algorithm to produce a new extended parser, rather than modifying an existing parser.

The immediate goal of the compilation process is to produce a structure that can be used to efficiently parse input. Also, since compilation is something that happens while the client program, the program using the parser, is running, the efficiency of the compilation process is very much an issue. In the following subsections, I will discuss representation and efficiency issues.

5.1.1. Parser representation

The parser must be represented in a way that makes it possible to efficiently parse input. The parsing process is not only a matter of matching the input to the grammar, but also of constructing the values of different kinds of expressions and executing the user-supplied actions. Because of this, the parsing process ends up being quite similar, at least in principle, to regular program execution. This also means that a convenient representation for a parser turns out to be a format often used for programs: bytecode.

For each nonterminal in the grammar, a bytecode program is generated that specifies how to parse that nonterminal. Just to give an impression of the format, the bytecode representation of the expression $(\mathbf{a}|\mathbf{b}|\mathbf{c})^*$ is shown in figure 5.1 on the following page, and the corresponding graph representation is shown in figure 5.2 on the next page. The program shown contains instructions concerned with parsing the input (the `term`

```
    nil
1:  choice
2:  term: a
3:  cons
    goto: 1
4:  term: b
    goto 3
5:  term: c
    goto 3
6:  end
```

Figure 5.1.: Bytecode representation of $(\mathbf{a}|\mathbf{b}|\mathbf{c})^*$

instructions), instructions that construct the value of the parsed input (the nil and cons instructions) and instructions directing the control of the parser (the choice, goto and end instructions). The details of what the instructions mean will be described in abundant detail in section 5.3.

The bytecode shown in figure 5.1 represents the same as the graph in figure 5.2 but the bytecode representation much harder to understand, especially with respect to control flow. The graph is easier to read because the control flow of the code has been made explicit as edges between the nodes, which correspond to instructions. In the following, all bytecode will be represented as graphs.

5.1.2. Choice representation

A very important aspect of how the input is parsed is how the parser makes choices. An overview of how this happens was given in section 3.7. When making a choice, the parser first uses the lookahead token to rule out some of the alternatives. If there is only one alternative left, that alternative is chosen. Otherwise, there is a local ambiguity, which is resolved by using tentative parsing.

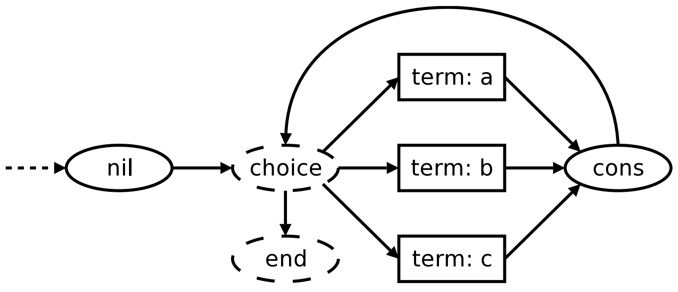


Figure 5.2.: Graph representation of $(\mathbf{a}|\mathbf{b}|\mathbf{c})^*$

To be able to do this, the parser must, when it reaches a choice, have some way of finding out which of the alternatives are applicable, based on the current lookahead token. This choice is made using a *choice table*. Each choice has an associated choice table that gives, to the current lookahead, the applicable alternatives. The bulk of the work done during compilation has to do with gathering the information necessary to build the choice tables.

What information do we need to be able to build a choice table? Consider the statement grammar, here with names added:

$$\begin{array}{rcl}
\langle\mathsf{stmt}\rangle[\mathsf{shortIf}] & \rightarrow & \textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \langle\mathsf{stmt}\rangle \\
[\mathsf{longIf}] & | & \textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \langle\mathsf{stmt}\rangle\ \textbf{else}\ \langle\mathsf{stmt}\rangle \\
[\mathsf{while}] & | & \textbf{while}\ \langle\mathsf{expr}\rangle\ \textbf{do}\ \langle\mathsf{stmt}\rangle \\
[\mathsf{block}] & | & \{\ \langle\mathsf{stmt}\rangle^*\ \} \\
[\mathsf{expr}] & | & \langle\mathsf{expr}\rangle\ \textbf{;}
\end{array}$$

The choice table generated for the choice between these five productions is this:

Token	**Production**
if	{longIf, shortIf}
while	while
{	block
(	expr
int	expr
id	expr

This table gives, to a token, the applicable productions. What we need to build this table is the set of tokens that can occur first in each production. For the first four production that is easy, because they all start with a terminal. But since the last production starts with a ⟨expr⟩, the tokens that can occur first in that production is the tokens that can occur first in an ⟨expr⟩. In this example, that was taken to be **(**, **int** and **id**. The set of tokens that can occur first in a nonterminal is called the *first set* of that nonterminal. Determining the first set of all nonterminals is one of the steps in compiling a grammar.

In some cases, however, we need to know more than the first sets to generate the choice table. Consider this small grammar for field declarations shown in figure 5.3 on the following page. This grammar produces declarations like

public int x;

and

byte c;

$$
\begin{array}{rcl}
\langle\textsf{field-decl}\rangle & \rightarrow & \langle\textsf{modifier}\rangle\ \langle\textsf{type}\rangle\ \textbf{id ;} \\
\\
\langle\textsf{modifier}\rangle[\textsf{public}] & \rightarrow & \textbf{public} \\
[\textsf{private}] & | & \textbf{private} \\
[\textsf{none}] & | & \varepsilon \\
\\
\langle\textsf{type}\rangle & \rightarrow & \textbf{int} \mid \textbf{byte} \mid \textbf{float} \mid \cdots
\end{array}
$$

Figure 5.3.: A simple grammar for field declarations

When parsing a field declaration, the parser must first parse the ⟨modifier⟩, which means that it must make the choice between ⟨modifier⟩'s three productions, based on the first token of the sentence. In the first example above, the choice is easy: the first token is **public**, which means that the right production must be the one called public. In the second example, the first token of the input is **byte**, and looking at the grammar, we can see that this means that there is no modifier. In general, if the lookahead token is **int**, **byte**, or any other type, then there is no modifier, so the none production should be taken. The choice table for ⟨modifier⟩ is this:

Token	Production
public	public
private	private
int	none
byte	none
float	none
⋮	⋮

How do we determine the set of tokens that imply that there is no ⟨modifier⟩? Well, if there is no modifier, then that means that the current token was not generated by ⟨modifier⟩, but by something following it. In this case, it is followed by a ⟨type⟩, so any token that can occur first in a type implies that there is no modifier. In general, if a production can produce the empty string, then we have to take the tokens that can follow it into account. This is why we didn't need to consider this in the ⟨stmt⟩ example: none of the productions can produce the empty string. The set of tokens that can occur after a nonterminal is called the *follow set* of that nonterminal. As this example demonstrates, we need to find the follow set of the nonterminals to construct the choice table.

On top of having to analyze the grammar to find these properties, it is also important that the compilation process is fast. The next section explains why that is, and describes the approach used to achieve this.

5.1.3. Compilation efficiency

With parser generators, the parser used by a client program is produced at the same time as the client program itself is compiled, and the resulting parser is linked into the client program. This means that the cost of compiling the parser can be paid once and for all, and the result can then just be loaded and used directly.

Here, parsers are generated at runtime, which means that the time and space cost of compilation must be paid by the program that will use the parser, each time it is run. If the compilation process is expensive, it will be felt directly in the client program, especially if the client program needs to construct many different parsers. Allowing this is one of the goals of the library. If the compilation process is too inefficient, the library becomes much less useful for those applications.

The form of the parser generation process, applying a delta to an existing grammar, gives a great potential for speeding up the compilation process. If you're making a limited update of an existing grammar, it is probably not necessary to completely recompile the derived grammar. Instead, it should be possible to reuse much of the analysis and structure of the base grammar. In other words, the compilation process should be *incremental.*

The problem is that neither the first set or follow set of a nonterminal can be calculated independently of the rest of the grammar. For instance, consider the following nonterminal:

$$\begin{array}{lcl} \langle\mathsf{stmt}\rangle & \rightarrow & \textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \langle\mathsf{stmt}\rangle\ \textbf{else}\ \langle\mathsf{stmt}\rangle \\ & | & \langle\mathsf{expr}\rangle\ \textbf{;} \end{array}$$

As mentioned before, we need to calculate the first set of all nonterminals to construct the choice tables. In this case, the first set of the ⟨stmt⟩ nonterminal depends directly on the first set of ⟨expr⟩, which means that even if an extension of a grammar does not change ⟨stmt⟩ directly, its first set may still have to be recalculated because of changes made to ⟨expr⟩. The compilation process handles this by carefully recording the dependencies that exist between the the nonterminals so that, in the above example, it is recorded that changes to the first set of ⟨expr⟩ affect ⟨stmt⟩ and should cause the first set of ⟨stmt⟩ to be recalculated. This means that, beyond the immediate goal of calculating properties of the nonterminals, the compilation process must identify the dependencies among the nonterminals and maintain records of those dependencies.

Up until now, I have talked about applying a delta to a grammar without describing how a grammar is represented. A grammar is represented by a *grammar descriptor* that contains, apart from just a representation of the grammar, information about properties of the grammar and dependency information. Exactly what information and which dependencies will be described in the following.

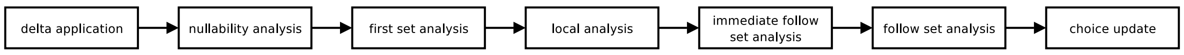


Figure 5.4.: Overview of the compilation process

5.1.4. The language treated

In the following, I will only treat the compilation of a subset of the full definition language described in chapter 3. Generalizing the compilation process to the full language is straightforward, except for the bytecode generation phase, where the a larger subset of the language will be treated.

Also, instead of using the syntax introduced in chapter 3, I will use an abstract syntax in the following. The subset used is this:

Concrete	**Abstract**
term	term(*term*)
⟨nonterm⟩	nonterm(*nonterm*)
ε	empty
$\varphi_1\,\varphi_2\,\cdots\,\varphi_n$	seq($\varphi_1, \varphi_2, \ldots, \varphi_n$)
$\varphi_1 \mid \varphi_2 \mid \cdots \mid \varphi_n$	choice($\varphi_1, \varphi_2, \ldots, \varphi_n$)
$\{\varphi_1 : \varphi_2\}$	infix(φ_1, φ_2)

5.1.5. Compilation overview

The compilation process consists of a sequence of phases, as shown in figure 5.4. The first phase is concerned with constructing a representation of the grammar that results from applying the delta, and generating bytecode. The second phase, the *nullability analysis* is an analysis that finds the set of nonterminals that can generate the empty string, which is something we need to know for the next two phases. The following phase is the *first set analysis*, in which the first set of all nonterminals is found. The *local analysis* phase uses the results of the nullability and first set phases to calculate some properties about locations within the bodies of the nonterminals. The last two analysis phases are the *immediate follow set analysis*, which is a “preprocessing phase”, and the *follow set analysis*, which uses the previous analysis to find the follow set of each nonterminal. Finally, the choice tables can be constructed using the results of the local analysis and follow set analysis. The description given here is relatively detailed, but does gloss over some aspects, especially in relation to exactly how the dependencies are tracked. A full account of the algorithm in all its glory is given in pseudocode in appendix C on page 121.

As mentioned, the bulk of the work done when compiling is the analysis necessary to generate the choice tables. The overall structure of each of the five analysis phases are relatively similar.

1. The previous phases will have identified a set of nonterminals that may have been affected by the update. For instance, if a nonterminal has been changed directly it must have its first set recalculated.

2. The full set of nonterminals that must have the property recalculated is found. The set consists of the set of nonterminals found in the first step and possibly a number of nonterminals that depend on them. An example is the one above, where the first set of ⟨stmt⟩ depended on the first set of ⟨expr⟩. In this case, if ⟨expr⟩ was found to have been affected in step 1, the dependency would cause ⟨stmt⟩ to be marked in step 2. In the case of first sets, this will again cause the nonterminals depending on ⟨stmt⟩ to be marked, but that is not always the case.

3. The property is recalculated for all the nonterminals identified in steps 1 and 2. If a nonterminal has not been marked, it means that the property cannot possibly have changed, and so it is safe to reuse the information stored in the base grammar.

4. When the property has been recalculated for a nonterminal, it is inspected to see whether the property actually did change. If it did, this information is stored and may be used in step 1 of a following phase. For instance, if the first set of a nonterminal changes, any nonterminal containing a reference to that nonterminal must be reanalyzed during the local analysis phase.

5.2. Delta application

The starting point of the compilation process is that we are given a delta and a grammar descriptor. The first phase is concerned with applying the delta to the given base grammar to obtain a representation of the derived grammar. In section 3.10.2, a delta was described as being simply a sequence of directives that either add or remove a production. The delta is applied by sequentially applying the directives to the base grammar.

A grammar is represented as a mapping from nonterminal names to the productions of that nonterminal. The productions of a nonterminal are represented by another mapping, from the name of a production to the production itself. Productions that have no name are given distinct dummy names.

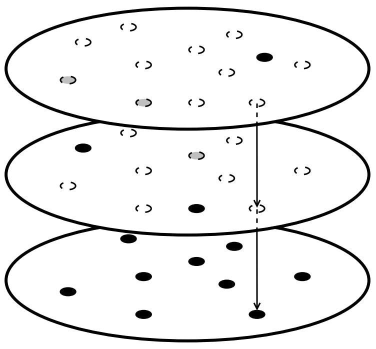


Figure 5.5.: A layered map

Before, I said that the changes are applied to the base grammar. That is not completely true, because the result of applying the delta should really be a new grammar, and the base grammar should remain unchanged. To implement this, the mapping from the name of a nonterminal to the body is represented as a a layered structure similar to the one shown on the right. The map of the derived grammar is initialized to be an empty layer on top of the base grammar. Looking up an entry is done by first looking up the entry in the current layer. If it is not there, the lookup continues recursively in the layer below. Changes are always made in the current layer, so if we need to make a change to a nonterminal in the layer below, the nonterminal is first copied up into the current layer and changed there. This way, the representation of the derived grammar can share the structure of the base grammar for all the nonterminals that are unchanged.

An add-directive of this form:

$$\langle \mathsf{nonterm} \rangle [n] \quad \rightarrow \quad \varphi$$

is executed by first looking up ⟨nonterm⟩ in the grammar – if it does not already exist, it is added with an empty body. The production is then added to it.

A **drop** directive,

$$\textbf{drop}\ \langle \mathsf{nonterm} \rangle\ n_1\ n_2\ \ldots\ n_k$$

is executed by looking up ⟨nonterm⟩ and removing the productions with the specified names.

When all directives have been applied, the grammar descriptor now has a description of the derived grammar. At this point, the *body* of each changed nonterminal is constructed. The body of a nonterminal is the single regular expression consisting of a choice between all the productions of the nonterminal. For instance, if a nonterminal has n productions, $\{p_1, p_2, \ldots, p_n\}$, the body of the nonterminal is $(p_1|p_2|\cdots|p_n)$. The body of a nonterminal is used in the first two analysis phases.

In 3.3, we saw that a parser can only be generated for a grammar if the grammar is well-formed. Because of this, it would be natural to expect that the grammar was checked for well-formedness at some point during the compilation. While undefined nonterminals and illegal repetition expressions are detected during the compilation, it turns out that an explicit check must be made for left recursion. While this is possible and probably not very complex, I have not included such a check in the compilation algorithm.

The final part of this phase is concerned with generating bytecode for the nonterminal. The translation process is described in the next section.

5.3. Bytecode translation

The bytecode generated for a nonterminal expresses two things: how the nonterminal should be parsed, and how the value of the parsed expression should be constructed.

The expressions to be translated are very similar to regular expressions. The process of going from a regular expression to a finite automaton is well-understood (Kozen 1997), and a state minimization algorithm exists for such finite automata. Unfortunately, those

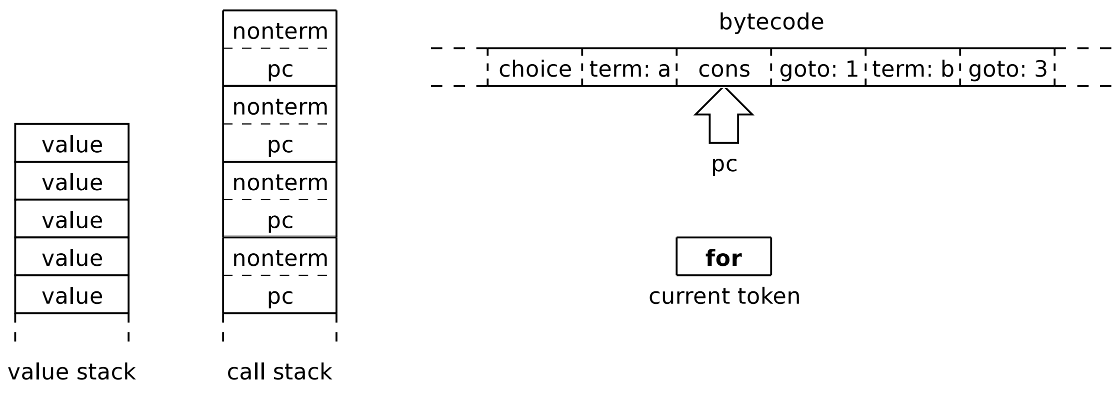


Figure 5.6.: Diagram of the parser engine

algorithms do not apply to this case. What makes this case different from ordinary regular expressions is the second requirement: the bytecode must also describe how the value of the expression is constructed. So while the procedure for the graph construction used here is very similar to the one used with ordinary regular expressions, we cannot apply the other algorithms, such as the state minimization algorithm.

Before going into the actual translation, I will give a short overview of the engine that will eventually execute the bytecode. This will not be the whole story but it will do until chapter 6 where I will give a complete description.

5.3.1. Parser engine overview

The bytecode of a parser is executed by the parser engine as if it were a program. Figure 5.6 gives a cartoon overview of the parser engine. The parser engine sequentially reads and executes the bytecode instructions. The state of the engine consists of three components: the value stack, the call stack and the current token.

The value stack is used to construct the value of the parsed input. All the value-constructing instructions operate in terms of manipulating values on the value stack. The call stack is used to manage the control of the parser. Whenever the parser has to recursively parse a nonterminal, a *nonterminal call*, the current nonterminal and the current program counter are pushed onto the stack, and restored when the parser is done parsing the call.

Virtual machines for general programming languages also use value and call stacks, but they are usually merged into a single stack that serves both purposes. Here, the call stack is used in a way that makes is much more convenient to keep two separate stacks.

The last component is a single register containing the current token, which is used to control some of the instructions.

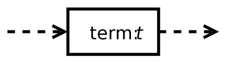
Now, we are ready to finally get to the actual description of how expressions are translated.

5.3.2. Translating simple expressions

To start off gently, I will first show the translation of some simple expressions, without considering whether or not the expressions are considered relevant (in the sense of relevance and irrelevance introduced in 3.5.1).

Terminal expressions

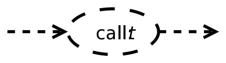
A terminal expression, term(t), is translated into a term node:



When executed, the term instruction does two things: first, it sees whether or not the current token is t. If it is not then parsing fails. If it is, it reads the value of the token and pushes it onto the value stack. Then it reads the next token from the scanner and stores it as the current token. It is an invariant of the translation that whenever the code corresponding to a relevant expression has been executed, a single value, the value of the expression, will be left on the value stack.

Nonterminal expressions

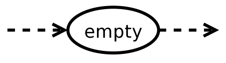
A nonterminal expression, nonterm(t), is translated into a call node:



As described, call instruction causes the current nonterminal structure and the current program counter to be pushed onto the call stack. The nonterminal structure for t is then fetched from the current parser and execution continues from the first instruction of the called nonterminal's bytecode. Later, after that nonterminal has been parsed, control will be passed back to the instruction following the call instruction.

Empty expressions

The empty expression, empty, is translated into an empty node:

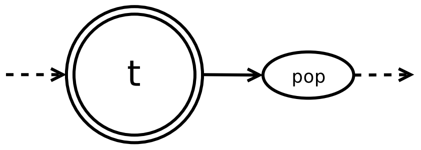


The empty instruction causes the null value to be pushed onto the value stack.

5.3.3. Translation modes

Above, we assumed that we were actually interested in the values of the expressions. But to match the semantics of expressions described before, we have to generate different code when the expression is relevant and when it is irrelevant. The code for the relevant expression should cause a value to be pushed onto the value stack, as above, and their irrelevant counterparts should match the input but not cause any values to be constructed.

In the case of terminals and nonterminals, generating the code for the corresponding irrelevant expressions is straightforward: just generate code for the relevant expression and then add a pop instruction at the end to pop the generated value off the stack again. If t is a term(x) or nonterm(x) expression, ignore(t) can be translated into this:



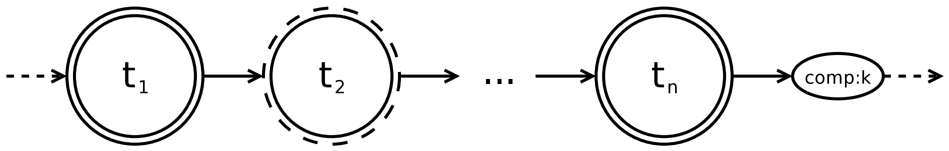
A note on the graphical notation: in the following, a large double circle with a solid outline, such as the one above will mean "whatever t was translated into, as a relevant expression". A double circle with a dashed outline will mean the same, only with t being translated as an irrelevant expression.

The simple translation scheme, translating an irrelevant expression into a relevant one and then just popping the value, will only work for the simple expressions and in the following, relevance has to be considered more carefully.

5.3.4. Sequences

As described in section 3.5, the rule for constructing the value of a sequence expression, seq($t_1, t_2, \ldots, t_n$), had three special cases. If there were no relevant subexpressions, the whole sequence was considered irrelevant and the value of the whole expression should be the empty value. If exactly one subexpression was relevant, the value of the whole sequence should be the value of that subexpression. If there was more than one relevant subexpressions, say k, the value of the whole expression should be a k-tuple containing the values of those k subexpressions.

This figure shows how the last case is translated:



The subexpressions are chained together so that t_1 is first executed, then t_2 and so forth. The relevant expressions (here t_1 and t_n) are translated as relevant, and the irrelevant ones (here t_2) are translated as irrelevant. After t_n has been executed, the k relevant expressions will have caused k values to be pushed onto the stack. The last instruction, comp:k, pops those values off the stack and stores them in a k-tuple, which is then pushed back onto the value stack.

When there is a single relevant subexpression, the sequence is translated in the same way, except that no comp instruction is used. When the code for all the subexpressions has been executed, the value of the relevant subexpression will automatically be left on the value stack. If there are no relevant subexpressions then all the subexpressions will have been translated as irrelevant, which means that after parsing them, no values will be left on the stack. In this case, the value of the whole expression should be the null value which we will then have to explicitly push onto the stack with a empty instruction at the end.

If a sequence has been marked to be ignored, it is translated the same way for all special cases: the subexpressions are translated as a chain, and all translated as irrelevant.

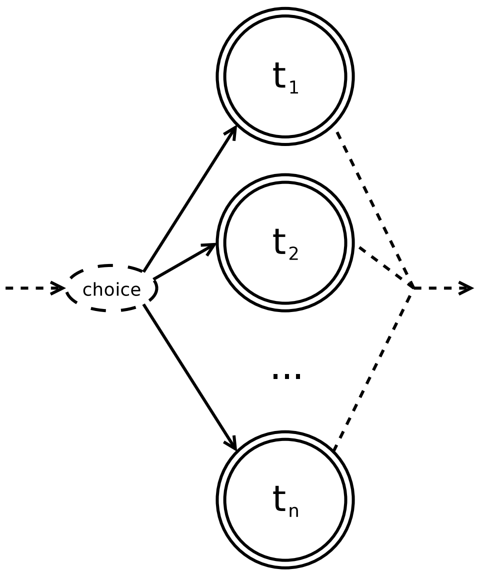
5.3.5. Choices

So far, the translation rules have been completely straightforward. We have not, for instance, tried to be clever and notice special cases where more efficient code could be generated. Spending some time on optimizing the code is usually considered a good investment, and doing so is certainly a possibility, but it is not something I will discuss here.

The library is, however, required to optimize the way choices are made, as described in 3.7.3. Before showing how this optimization is performed, I will first give the straightforward translation of a choice expression, which we will later improve into the full, optimized translation.

Naive translation

The straightforward translation of a relevant choice, choice($t_1, t_2, \ldots, t_n$), is this:



The choice instruction is responsible for making the choice between the alternatives. A detailed overview of how this is done, using tentative parsing, has already been given. After the code has been executed, one of the alternatives will have left a value on the value stack, which will be the value of the whole choice expression. Note that this is exactly the point where we might need to get the value of an insignificant expression, because the code for all the subexpressions must leave a value on the stack, even the insignificant ones.

In section 3.5, it was specified that a choice, choice($t_1, t_2, \ldots, t_n$), was considered insignificant exactly if it was ignored or all the t_i were insignificant. An insignificant choice can be translated the same way as a relevant one, except that all the subexpressions are translated as insignificant.

But as mentioned in section 3.7.3, the language actually specifies that choices should be made in a certain optimized way. To do this, we have to adopt a more clever translation scheme.

Optimizing translation

The guarantee given was that in a choice between a set of sequences, the choice between any sequences with a common prefix would be postponed until after that common prefix had been parsed. That means that writing this:

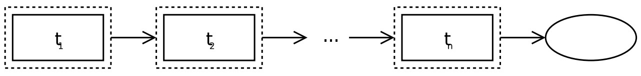
$$\begin{array}{lcl} \langle\text{stmt}\rangle & \rightarrow & \textbf{if}\ \langle\text{expr}\rangle\ \textbf{then}\ \langle\text{stmt}\rangle\ \textbf{else}\ \langle\text{stmt}\rangle \\ & | & \textbf{if}\ \langle\text{expr}\rangle\ \textbf{then}\ \langle\text{stmt}\rangle \end{array}$$

is guaranteed to be just as efficient as this:

$$\langle\text{stmt}\rangle \quad \rightarrow \quad \textbf{if}\ \langle\text{expr}\rangle\ \textbf{then}\ \langle\text{stmt}\rangle\ [\textbf{else}\ \langle\text{stmt}\rangle]$$

The straightforward translation scheme described above doesn't work simply because it cannot give that guarantee. The solution is to use a smarter translation that first "unfolds" the subexpressions and then "weaves" them together again, making choices only where it is necessary. The starting point is that we are given a choice expression, choice($t_1, t_2, \ldots, t_n$). For simplicity, we can consider all the t_i to be sequence expressions because all other expressions can be thought of as a sequence with only one element.

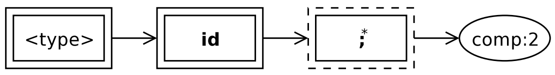
The first thing to do is to unfold all the sequences of the choice. The result of unfolding a sequence is a list of the sequence's subexpressions, each with an indication of whether or not the expression is relevant. Figuratively, the expression seq($t_1, t_2, \ldots, t_n$) is translated into this:



where each t_i is translated according to whether or not they are relevant. The final, unmarked, instruction represents the instruction that should follow the sequence (either comp:k, empty or none). To give a concrete example, the sequence

$$\langle\text{type}\rangle\ \textbf{id}\ \underline{\textbf{;}}^{*}$$

is unfolded into this



When all sequences have been unfolded, they are weaved together. An abstract example of the weaving process is shown in figure 5.7 on the facing page. For simplicity, the terms of the unfolded sequence are shown as numbers.

Given a number of unfolded sequences, shown as step 0, we first identify those sequences that have the same first term. Two terms are the same if they represent the same expression and have the same relevance. In step 1 in the figure, two such sequences have been identified. If a number of sequences have the same first term then those sequences are "joined" by merging their first term and recursively weaving them from the second term on. A choice is then inserted between all the first terms, except in the case where all sequences have the same first term. A special case of this is when there is only a single sequence to choose from. Finally, when all the sequences have been weaved together, the terms of the sequences can be translated.

This algorithm optimizes the code the way we guaranteed – in fact, it *exactly* optimizes the code that way and makes no attempt to optimize beyond the guarantee. The argument is as follows: because identical terms are always merged before a choice is inserted, the algorithm cannot insert a choice that chooses between identical terms. This gives us that if two sequences have the same prefix, then no choice can be inserted until after the common prefix, because any choice inserted before would be between identical terms.

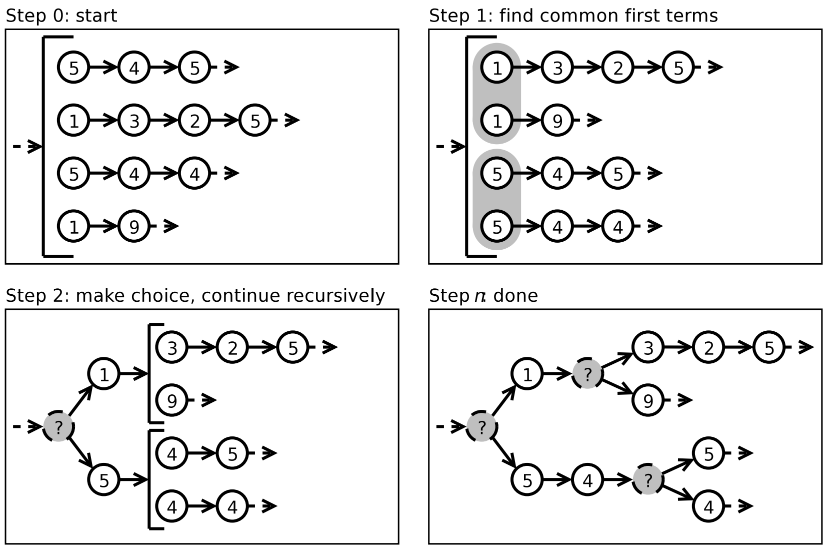
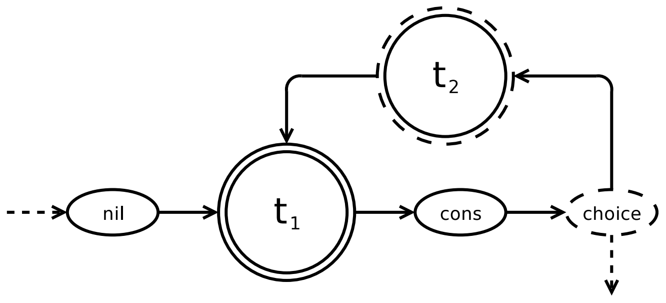

Step 0: start
Step 1: find common first terms
Step 2: make choice, continue recursively
Step n done

Figure 5.7.: Overview of the weaving procedure

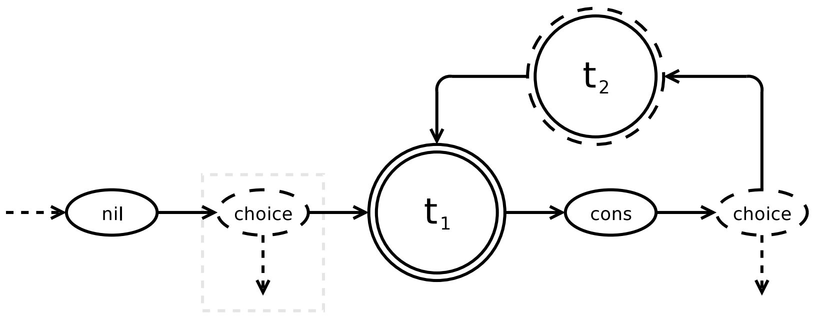
5.3.6. Infix repetition expressions

A relevant non-empty infix repetition expressions, infix(t_1, t_2), is translated into this:



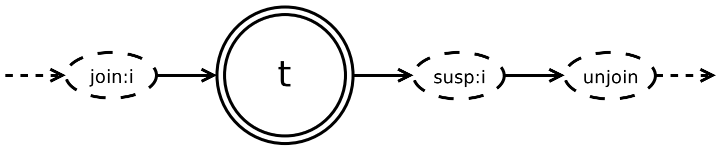
The nil instruction pushes a empty list onto the value stack. After the t_1 has been parsed, the cons instruction pops the two top elements, the list and the value, off the stack, appends the value to the list, and pushes the result back onto the stack. Then there is a choice between parsing a t_2 and then another t_1, or stopping. When the parsing is done, the top element on the value stack will be the list of t_1's parsed. An irrelevant infix plus expression is translated into the same basic structure but with t_1 translated as insignificant and without the list-constructing instructions.

A relevant infix star expression is translated the same way except that a choice to not parse anything at all is inserted in the beginning:



5.3.7. Suspend expressions

A suspend expression, suspend(t), is translated into this:



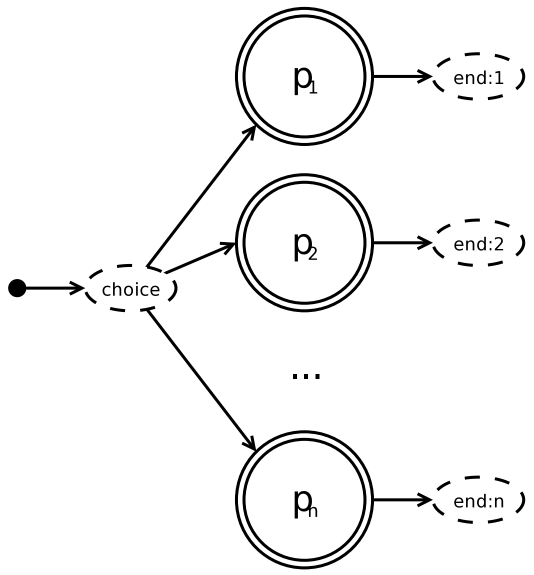
The semantics of suspend expressions was discussed in section 3.8, and these instructions work together to implement that. If, in tentative mode, one thread executes a join instruction then it is suspended. If all threads end up executing the instruction then they are joined together and a new, single, thread parses the body of the expression. The susp instruction causes the thread to pop off the value left on the value stack by the body of the expression and invoke the associated suspend action. Depending on the value returned from the suspend action, the parser may then be changed. Then, the unjoin instruction causes the control to be returned to the suspended threads and parsing continues. During reparsing, the join node causes the value returned by the suspend action to be pushed onto the stack, and then continues execution right after the unjoin instruction.

If the parser is not parsing tentatively, the join and unjoin operations have no effect.

When unparsing, the parser will need to be able to skip over a whole suspend expression. To do this, we have to record, for each join instruction, the location immediately following the corresponding unjoin instruction.

5.3.8. Whole nonterminal translation

We have now been through the translation rules for all the different expressions, except the expressions that can be trivially derived from the ones described. The last thing we need is the top level translation for a nonterminal. A nonterminal with productions $\{p_1, p_2, \ldots, p_n\}$ is translated into something similar to this:



This is actually not completely accurate because, since we are making a choice, we have to use the "unfold and weave" procedure from before and not just make the choice directly. Also, if there is only one production then there is no need to insert a choice in the first place.

The black dot marks the starting point, which is the instruction at location 0 in the bytecode. After the body of each production is an end node which pops the value of the production off the value stack, invokes the action associated with the production and pushes the returned value back onto the value stack. It then pops the call stack and returns to the specified location so that parsing can continue from where the nonterminal was originally called.

Having generated the nonterminal bytecode, the rest of the compilation process is concerned with analyzing the grammar and constructing a choice table for each choice instruction in the bytecode.

5.4. Nullability Analysis

The first analysis phase is concerned with determining which nonterminals are nullable, that is, can produce the empty string. This property is not very interesting in itself, but is used heavily in the two following phases. This analysis is not overly complex, but it takes a lot of explanation to justify that what happens is reasonable. In this first section especially, I will be careful with explaining and justifying the way this phase works, because much of what is discussed here applies in the following phases.

$$\begin{array}{lcl}
\text{nullable}(\mathsf{term}(t)) & = & \mathsf{false} \\
\text{nullable}(\mathsf{nonterm}(n)) & = & \mathit{nullability}(n) \\
\text{nullable}(\mathsf{empty}) & = & \mathsf{true} \\
\text{nullable}(\mathsf{seq}(t_1, \ldots, t_n) & = & \text{nullable}(t_1) \curlywedge \cdots \curlywedge \text{nullable}(t_n) \\
\text{nullable}(\mathsf{choice}(t_1, \ldots, t_n) & = & \text{nullable}(t_1) \curlyvee \cdots \curlyvee \text{nullable}(t_n) \\
\text{nullable}(\mathsf{infix}(t_1, t_2)) & = & \text{nullable}(t_1)
\end{array}$$

Figure 5.8.: The definition of nullability

5.4.1. Definition of nullability

The nullability of a nonterminal can be calculated as a function of the body of the nonterminal. The definition of nullability is shown in figure 5.8. This definition uses the $\curlywedge$ ("lazy and") and $\curlyvee$ ("lazy or") operators, rather than $\wedge$ and $\vee$. These operators are similar to $\wedge$ and $\vee$, except that they are lazy and evaluate from left to right. That means that where $a \wedge b$ requires both a and b to be well-defined, $a \curlywedge b$ can be well-defined even though b is not, if a is false.

Why is it necessary to consider laziness and order of evaluation? The problem is the nullability of a nonterminal is recursive, through the use of *nullability*(n). Consider, for instance, the nonterminal shown in figure 5.9. If the nullability had been defined using eager logic operators, the nullability of $\langle \mathsf{a} \rangle$ would be defined as

$$\mathit{nullability}(\langle \mathsf{a} \rangle) = (\mathsf{false} \wedge \mathit{nullability}(\langle \mathsf{a} \rangle) \wedge \mathsf{false}) \vee (\mathsf{false} \wedge \mathit{nullability}(\langle \mathsf{b} \rangle) \wedge \mathsf{false})$$

This definition is not just a function, it is a recursive equation. While this equation can be solved[1], a simpler solution is to use the lazy logic operators. Lazy operators save the day exactly because of the otherwise unfortunate business we discussed in section 3.3: the absence of left recursion in grammars. Back then, we defined a left recursive nonterminal to be a nonterminal that could produce a sentence whose first term is the nonterminal itself. If such sentences are disallowed, it means that if we always go from left to right when evaluating the nullability of a nonterminal x, we must necessarily meet something that is not nullable before meeting a reference to x itself. Remember that because of

[1]These equations can be solved iteratively, using the fix-point algorithm. This hinges on the fact that the nullability can be seen as a monotone function. Actually, since both the nullability, first set and follow set can be expressed as monotone recursive equations, this is how they are traditionally calculated (Appel 1998).

$$\begin{array}{lcl}
\langle \mathsf{a} \rangle & \to & \textbf{[}\ \langle \mathsf{a} \rangle\ \textbf{]} \\
 & | & \textbf{(}\ \langle \mathsf{b} \rangle\ \textbf{)}
\end{array}$$

Figure 5.9.: A simple recursive nonterminal

the laziness of the $\curlywedge$ operator, reaching a term that is not nullable means that we can stop evaluating, and hence we never reach the x.

Note that the arguments above only use properties of the $\curlywedge$ operator, not $\curlyvee$. Using the $\curlyvee$ operator is actually completely unnecessary, we could have just as well used $\vee$. It is used only for symmetry reasons.

5.4.2. Nullability function

Now that the nullability function has been shown to be well-defined, the time has almost come to use it. First, we need to take care of one other issue. The definition of nullability given in the previous section is sufficient to calculate the nullability of a nonterminal in a grammar, but it is insufficient in one important regard: tracking dependencies. Because we aim to only recalculate the parts of the grammar that may be affected by an update, it is important to be able to tell which parts they are. This means that besides just knowing the nullability of a nonterminal n, we have to know which nonterminals can, if changed, affect the nullability of n.

When does the nullability of a nonterminal depend on the nullability of another nonterminal? One possibility is to say that if nonterminal n_1 occurs in nonterminal n_2, then n_2's nullability depends on n_1. If we consider the definition of nonterminal $\langle \mathsf{a} \rangle$ in figure 5.9, that would mean that if a change occurred in $\langle \mathsf{b} \rangle$, then $\langle \mathsf{a} \rangle$ should have its nullability recalculated, because $\langle \mathsf{b} \rangle$ occurs in $\langle \mathsf{a} \rangle$. By inspecting $\langle \mathsf{a} \rangle$, however, it becomes clear that no matter if $\langle \mathsf{b} \rangle$ is nullable or not, $\langle \mathsf{a} \rangle$ can never be nullable. So even though it might seem that $\langle \mathsf{a} \rangle$ depends on $\langle \mathsf{b} \rangle$, it really doesn't. On the other hand, consider this example:

$$\begin{array}{rcl} \langle \mathsf{c} \rangle & \rightarrow & \langle \mathsf{d} \rangle \\ & | & \langle \mathsf{e} \rangle \end{array}$$

Here, the nullability of $\langle \mathsf{c} \rangle$ really does depend on the nullability of $\langle \mathsf{d} \rangle$ and $\langle \mathsf{e} \rangle$: $\langle \mathsf{c} \rangle$ is nullable if either $\langle \mathsf{d} \rangle$ or $\langle \mathsf{e} \rangle$ is. These two examples show that determining the genuine nullability dependencies in a grammar is not completely trivial, and that just the occurrence of a nonterminal in the body of another does not imply that there is a dependency. The solution to this is to not use the definition directly, but instead use it to generate a *nullability function* for the nonterminal.

The nullability function of a nonterminal is really just the application of the definition of nullability to the body of the nonterminal, only written out in full. For instance, the nullability function of $\langle \mathsf{a} \rangle$ from section 5.4.1 would be

$$f_{\langle \mathsf{a} \rangle}(\bar{n}) = (\mathsf{false} \curlywedge n_{\langle \mathsf{a} \rangle} \curlywedge \mathsf{false}) \curlyvee (\mathsf{false} \curlywedge n_{\langle \mathsf{b} \rangle} \curlywedge \mathsf{false})$$

Given a mapping from nonterminals to their nullability, here called $\bar{n}$, this function computes the nullability of $\langle \mathsf{a} \rangle$. The main advantage of actually constructing the nullability function, rather than just applying the definition, is that the nullability function can be simplified, using simple equivalences such as these:

$$\begin{array}{rcl}
\mathsf{false} \curlywedge \phi & \equiv & \mathsf{false} \\
\mathsf{true} \curlywedge \phi & \equiv & \phi \\
\phi \curlyvee \mathsf{true} & \equiv & \mathsf{true} \\
& \vdots &
\end{array}$$

Using these rules, the nullability function shown above for $\langle\mathsf{a}\rangle$ can be simplified to just

$$f_{\langle\mathsf{a}\rangle}(\bar{n}) = \mathsf{false}$$

Note that $n_{\langle\mathsf{b}\rangle}$ does not occur in the simplified nullability function, which reflects the fact that $\langle\mathsf{a}\rangle$'s dependency on $\langle\mathsf{b}\rangle$ was only superficial. On the other hand, the nullability function[2] for $\langle\mathsf{c}\rangle$ is

$$f_{\langle\mathsf{c}\rangle}(\bar{n}) = n_{\langle\mathsf{d}\rangle} \curlyvee n_{\langle\mathsf{e}\rangle}$$

In this case, $\langle\mathsf{c}\rangle$'s dependency on $\langle\mathsf{d}\rangle$ and $\langle\mathsf{e}\rangle$ is reflected by the fact that $n_{\langle\mathsf{d}\rangle}$ and $n_{\langle\mathsf{e}\rangle}$ occur in the simplified nullability function. This gives a way to determine the genuine dependencies of a nonterminal: the nullability of a nonterminal k depends on the nullability of k' if $n_{k'}$ occurs in the simplified nullability function of k.

We are now done with the preliminaries and ready to go ahead and calculate the nullability. The analysis proceeds as follows: first we recalculate the nullability function for the nonterminals where it might have changed (which includes new nonterminals). We then use the nullability functions to find out which nonterminals might have changed their nullability. Finally, we recalculate the nullability of all nonterminals where it might have changed.

5.4.3. Recalculating the nullability function

Which nonterminals should have their nullability function recalculated? The nullability function of a nonterminal only depends on the body of the nonterminal. This means that it should be recalculated for each nonterminal whose body has been directly changed as part of the update. The nullability function for all other nonterminals can be reused from the base grammar.

Once the nullability function has been found, we can find the genuine dependencies and use them to update the dependencies between the nullability of nonterminals of the grammar.

5.4.4. Recalculating nullability

Having recalculated the nullability functions, we can now determine which nonterminals should have their nullability recalculated. The nullability of a nonterminal of course

[2]Hereafter, the term "nullability function" will always refer to the simplified nullability function.

depends on the nonterminal's nullability function, so all nonterminals whose nullability function changed should have their nullability recalculated.

If the nullability of a nonterminal x depends on a nonterminal y, whose nullability might have changed, the nullability of x might also changed. This means that the nullability should be recalculated for all nonterminals that depend, directly or indirectly, on a nonterminal whose nullability function has changed.

Finally, when we have identified all the nonterminals whose nullability might have changed, it can be recalculated using the nullability function. Having done that, we can construct the set of all the nonterminals whose nullability actually did change. We need to know that set in the next two phases.

5.5. First set analysis

The first set of a nonterminal is the set of tokens that may occur as the first token of a string generated by that nonterminal. The procedure for calculating the first sets is very similar to the one calculating the nullability.

As with the nullability, the first set of a nonterminal can be calculated as a function of the nonterminal's body. The definition is shown in figure 5.10. This definition is used to construct a *first set function* for each nonterminal. If the nonterminal ⟨expr⟩ is nullable, the first set function for the example ⟨stmt⟩ grammar from before,

$$\begin{array}{lcl} \langle\mathsf{stmt}\rangle & \rightarrow & \textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \langle\mathsf{stmt}\rangle\ \textbf{else}\ \langle\mathsf{stmt}\rangle \\ & | & \langle\mathsf{expr}\rangle\ \textbf{;} \end{array}$$

would be

$$f(\bar{s}) = \{\mathbf{if}\} \cup (s_{\mathsf{expr}} \cup \{\mathbf{;}\})$$

The expressions that make up first the first set function are empty sets, singleton sets, variables and unions of such expressions. Since the only "connective" used is set union, such an expression can be simplified into just two components: the set of tokens and the set of variables occurring in the expression. For instance, the expression above could be simplified into this:

$$\begin{array}{lcl} \text{first}(\mathsf{term}(t)) & = & \{t\} \\ \text{first}(\mathsf{nonterm}(n)) & = & \mathit{first}(n) \\ \text{first}(\mathsf{empty}) & = & \{\} \\ \text{first}(\mathsf{seq}(t_1, \ldots, t_n)) & = & \bigcup_{i=1}^{k} \text{first}(t_i),\ k = \min\{i | \neg\text{nullable}(t_i)\} \\ \text{first}(\mathsf{choice}(t_1, \ldots, t_n)) & = & \bigcup_{i=1}^{n} \text{first}(t_i) \\ \text{first}(\mathsf{infix}(t_1, t_2)) & = & \text{first}(\mathsf{seq}(t_1, t_2)) \end{array}$$

Figure 5.10.: The definition of the first set

$$f(\bar{s}) = \{\mathbf{if}, \mathbf{;}\} \cup (s_{\mathsf{expr}})$$

This is not as significant a simplification as with the nullability function, it only rearranges the terms of the function, but does make for a much simpler representation.

To calculate the first sets, we have to go through the same steps as we did in the nullability analysis. First, the first set function should be recalculated for all the nonterminals where it might have changed. Then, we have to identify the set of nonterminals whose first set might have changed. Finally, we can recalculate the first set for the nonterminals we found before.

5.5.1. Recalculating first set functions

With the nullability, the set of nonterminals whose nullability function might have changed was simply the nonterminals that had been changed directly. In this case, a nonterminal's first set function might have changed even though the nonterminal has not been changed directly. This is because of the rule for calculating the first set of a sequence, which depends on the nullability of some of the subterms:

$$\text{first}(\mathsf{seq}(t_1, \ldots, t_n)) = \bigcup_{i=0}^{k} \text{first}(t_i),\ k = \min\{i | \neg\text{nullable}(t_i)\}$$

This means that if the nullability of one of the first k subexpressions of a sequence changes, the first set function may change and should consequently be recalculated.

The set of nonterminals whose nullability may affect an expression can be found, as we saw in the last section, as the set of variables occurring in the nullability function of that expression. This means that when the first set function of a nonterminal x is constructed, we can find the set of nonterminals whose nullability may affect it by finding the nullability functions of the subexpressions when using the rule for sequences.

That gives the rule for determining which nonterminals to recalculate: a nonterminal must have its first set function recalculated if it has been directly changed by the update, or if its first set function depends on the nullability of a nonterminal whose nullability has changed. Hence, at the beginning of the first set phase, we can use the set of nonterminals whose nullability has changed, which was found in the previous phase, to find the set of nonterminals that should have their first set function recalculated.

5.5.2. First set

Having found the first set function for all nonterminals, we can find the set of nonterminals whose first set might have changed. First off, if the first set function of a nonterminal has changed then the first set of that nonterminal might have changed. Furthermore, if the first set of a nonterminal x depends on the first set of a nonterminal y, and the first set of y might have changed, then the first set of x might have changed. This means that, as with the nullability, we need to take the transitive closure over the "depends on" relation to get the full set of nonterminals whose first set should be recalculated.

Finally, the first set can be calculated for all the nonterminals found above, and we can then identify the nonterminals whose first set really did change.

5.6. Local analysis

The first two analysis phases each determined a single property of each nonterminal. This phase is concerned with determining similar properties but this time about locations within the code of each nonterminal. It contains two sub-analyses: the reducibility analysis and the local first set analysis.

The result of the local analysis will be used for two things. One part of the analysis will be used directly to construct the choice tables, and another part will be used in the two following analyses. Before going on, I will just review where we are in the compilation process.

5.6.1. Follow set

As shown in the ⟨modifier⟩ example in the introduction, it is necessary to find the set of tokens that can follow each nonterminal. Finding this set is the most complex part of the compilation because it is not as much a property of the nonterminals themselves, but of the surrounding grammar that uses the nonterminals. Because of this, the follow set of a nonterminal is very sensitive to changes in the rest of the grammar.

A token can end up in the follow set of a nonterminal in two ways. The one way is if it occurs immediately after a call to the nonterminal. Consider a variant of our running example, the ⟨stmt⟩ nonterminal, that has a **repeat** ... **until** loop rather than a **while** ... **do** loop:

$$
\begin{array}{rcl}
\langle\mathsf{stmt}\rangle & \rightarrow & \textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \langle\mathsf{stmt}\rangle \\
& | & \textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \langle\mathsf{stmt}\rangle\ \textbf{else}\ \langle\mathsf{stmt}\rangle \\
& | & \textbf{repeat}\ \langle\mathsf{stmt}\rangle\ \textbf{until}\ \langle\mathsf{expr}\rangle \\
& | & \{\ \langle\mathsf{stmt}\rangle^*\ \} \\
& | & \langle\mathsf{expr}\rangle\ \textbf{;}
\end{array}
$$

Looking at this grammar, we can see two tokens that can follow a ⟨expr⟩: the **then** token, from the two if-statements, and the **;** token, from the last expression statement. One of the things that will be computed for each nonterminal during the local analysis is a map that gives, to a nonterminal x, the set of tokens that can occur immediately after a call to x in that nonterminal. For ⟨stmt⟩, the result would be this:

Nonterminal	Tokens
⟨stmt⟩	{**else**, **}**, **until**}
⟨expr⟩	{**then**, **;**}

This table gives to each nonterminal the *local follow sets* of that nonterminal, which is the set of tokens that can occur immediately after the nonterminal in the body of $\langle\mathsf{stmt}\rangle$.

A token can also occur after a nonterminal even if it does not occur immediately after a call to the nonterminal. If a token can follow a nonterminal y and x occurs at the end of y, then that token can follow x. An concrete example is demonstrated by this short derivation:

$$\begin{array}{rcl} \langle\mathsf{stmt}\rangle & \rightarrow & \textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \langle\mathsf{stmt}\rangle\ \textbf{else}\ \langle\mathsf{stmt}\rangle \\ & \rightarrow & \textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \lfloor \textbf{repeat}\ \langle\mathsf{stmt}\rangle\ \textbf{until}\ \langle\mathsf{expr}\rangle \rfloor\ \textbf{else}\ \langle\mathsf{stmt}\rangle \end{array}$$

Because $\langle\mathsf{expr}\rangle$ occurs at the end of the **repeat** ... **until** statement, and the token **else** can follow $\langle\mathsf{stmt}\rangle$, **else** can also follow $\langle\mathsf{expr}\rangle$. That is something we won't deal with until later, but we do have to prepare for it now. In particular, we need to know, for each nonterminal x, the set of nonterminal calls that occur at the end of x. In the $\langle\mathsf{stmt}\rangle$ example, two nonterminal can occur at the end of a statement: $\langle\mathsf{expr}\rangle$, as we have seen, and $\langle\mathsf{stmt}\rangle$ itself.

This is half of what the local analysis must calculate for each nonterminal: the local follow sets and the set of nonterminals that occur last in that nonterminal.

5.6.2. Partial calculation of choice tables

In the introduction we saw that as long as none of the alternatives of a choice can produce the empty string, we don't need to have the follow set of the nonterminal to construct the choice table. As an example of this, we saw two examples: the $\langle\mathsf{stmt}\rangle$ grammar, where no production could produce the empty string, and the field declaration grammar, where $\langle\mathsf{modifier}\rangle$ had an empty production, which meant that we had to calculate the follow set. In this phase we will calculate two things related to this. First of all, for each choice, we will calculate the tokens that can occur first in each of the alternatives, but without considering the follow set. Instead, we will just identify which alternatives can produce the empty string so that, when we have calculated the follow sets, we can go back and fill in the rest of the choice table.

In the field declaration example from the introduction,

$$\begin{array}{rcl} \langle\mathsf{modifier}\rangle[\mathsf{public}] & \rightarrow & \textbf{public} \\ [\mathsf{private}] & | & \textbf{private} \\ [\mathsf{none}] & | & \varepsilon \end{array}$$

that means that we should calculate something similar to this table:

Branch	**Local first**	**Reducible**
public	{**public**}	no
private	{**private**}	no
none	{}	yes

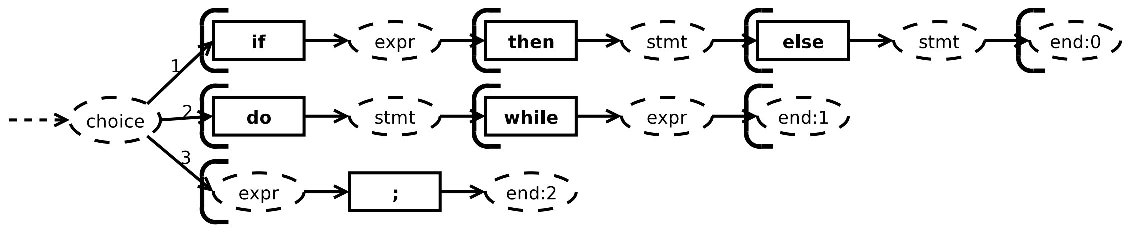


Figure 5.11.: Example graph

This totals to four different properties we have to calculate for each nonterminal: the local follow sets of the nonterminals called, the set of nonterminals that occur last in the nonterminal, the set of tokens that can occur first in each choice branch and finally, the branch choices that can produce the empty string.

These four properties actually boil down to calculating two properties: the reducibility, and the local first set of selected nodes in the graphs.

5.6.3. Reducibility

The reducibility analysis is used to calculate the set of nonterminals occurring last in the nonterminal, and the choice branches that can produce the empty grammar. A node is reducible if an end node can be reached, starting from just before that node, without passing through a node that will consume tokens. The reducibility must be calculated for each node following a choice node and each node following a nonterminal call. These nodes have been marked in the graph shown in figure 5.11, which is a simplified version of the graph of the following grammar:

$$\begin{array}{lcl} \langle\text{stmt}\rangle & \rightarrow & \textbf{if}\ \langle\text{expr}\rangle\ \textbf{then}\ \langle\text{stmt}\rangle\ \textbf{else}\ \langle\text{stmt}\rangle \\ & | & \textbf{do}\ \langle\text{stmt}\rangle\ \textbf{while}\ \langle\text{expr}\rangle \\ & | & \langle\text{expr}\rangle\ \textbf{;} \end{array}$$

If a node is reducible then it is possible to reduce, that is, finish parsing, the nonterminal without consuming any more input. For a node following a choice node, this corresponds to the possibility of producing the empty string. Similarly, if a node following a nonterminal call is reducible, that corresponds to the call occurring last in the nonterminal.

Determining the reducibility is a simple matter of graph reachability: can an end node be reached without passing through a term node or a call node calling a nonterminal that is not nullable. The definition of reducibility is shown in figure 5.12 on the following page. This definition uses a flattened notation for graph nodes where *action* represents any value-constructing node. The reducibility is essentially the same as the nullability, only at the level of nodes, rather than whole nonterminals. In particular, a nonterminal is nullable exactly if the initial node is reducible. This leads us to the question of whether

$$
\begin{array}{lcl}
\text{reducible}(\mathsf{term}(t)\!\rightarrow \psi) & = & \mathsf{false} \\
\text{reducible}(\mathsf{call}(n)\!\rightarrow \psi) & = & \begin{cases} \text{reducible}(\psi) & \text{if nullable}(n) \\ \mathsf{false} & \text{otherwise} \end{cases} \\
\text{reducible}(\mathsf{choice}\!\rightarrow \{\psi_1, \ldots, \psi_n\}) & = & \bigvee_{i=1}^{n} \text{reducible}(\psi_i) \\
\text{reducible}(\mathsf{end}(i)) & = & \mathsf{true} \\
\text{reducible}(\mathit{action}\!\rightarrow \psi) & = & \text{reducible}(\psi)
\end{array}
$$

Figure 5.12.: Definition of reducibility

the the reducibility is even well-defined. The graph of a nonterminal can contain loops, if it uses repetition expressions, and since the reducibility of a node can depend on the following nonterminal, are we sure that the reducibility of a node does not end up depending on itself? The key to this is one of the conditions for a grammar being well-defined: in a repetition expression, the body of the separator are not allowed to both be nullable. This means that we cannot go around in a loop without going through a part of the graph that is not nullable. Since the calculation stops when it meets something that is not nullable, there is no way to get from a node, round a loop, and back to the node again. Hence, the reducibility is well-defined.

Once the reducibility has been calculated for all nodes, we can collect the set of nonterminals where the next node is reducible, and for each choice node, the branches that can produce the empty string.

5.6.4. Local first set

The same way the reducibility corresponds to the nullability of a nonterminal, the local first set corresponds to the first set of a nonterminal. The local first set of a node in the graph of a nonterminal is the set of tokens that can be produced, starting from immediately before that node. The definition of the local first set is shown in figure 5.13.

The local first set must be determined for the same nodes as the reducibility. The local first set of a node following a choice node is the set of tokens that can be produced by choosing that branch. The local first set of a nonterminal call is the immediate follow set of that call.

The reducibility analysis and the local first set analysis have been presented as separate

$$
\begin{array}{lcl}
\text{local-first}(\mathsf{term}(t)\!\rightarrow \psi) & = & \{t\} \\
\text{local-first}(\mathsf{call}(n)\!\rightarrow \psi) & = & \begin{cases} \text{first}(n) \cup \text{local-first}(\psi) & \text{if nullable}(n) \\ \text{first}(n) & \text{otherwise} \end{cases} \\
\text{reducible}(\mathsf{choice}\!\rightarrow \{\psi_1, \ldots, \psi_n\}) & = & \bigcup_{i=1}^{n} \text{local-first}(\psi_i) \\
\text{reducible}(\mathsf{end}(i)) & = & \{\} \\
\text{reducible}(\mathit{action}\!\rightarrow \psi) & = & \text{local-first}(\psi)
\end{array}
$$

Figure 5.13.: Definition of local first set

things, but are so similar that they can easily be performed as one. The reducibility analysis can, for instance, be folded into the local first set analysis by representing the reducibility of a node by whether or not the local first set contains a special “end” token.

5.6.5. Detecting changes

When the local analysis is done, we have calculated four properties of the the nonterminal. At this point, we have to compare the result of the analysis we just made of the derived with the analysis performed in the the base grammar, to detect changes.

First of all, if a nonterminal occurred at the end of a nonterminal in the base grammar, but not in the derived grammar, or vice versa, it is added to the set of nonterminals whose reducibility has changed. Similarly, the set of nonterminals whose local first set changed is identified. These two sets will be used to determine the set of nonterminals that must have their immediate follow set and follow set recalculated.

Similarly with the choices: if a branch could produce the empty string in the base grammar but not in the derived grammar or vice versa, the choice is added to a set of choices whose branches have changed. The same thing happens if the local first set of a branch has changed. We will need that information in the very last phase, when we determine the set of choice nodes that must have their choice table recalculated.

5.6.6. Recalculation

As in the previous phases, we only recalculate the local analysis for the nonterminals where it might have changed. The reducibility and local first set of a graph only depends on the shape of the graph and the nullability and first set of the nonterminals called from the graph. This means that if we perform the local analysis on all nonterminals that have been changed directly, or which call nonterminals whose nullability or first set has changed, we are on the safe side.

At this point, we are done with the results of the nullability and first set analyses. These two properties were calculated specifically to be used in the local analysis, and the following analysis will use results of the local analysis directly. The results of those analyses are not discarded, but will be stored in the grammar descriptor that is produced by this analysis, because they may be useful if the grammar produced here is ever extended.

5.7. Immediate follow set analysis

The immediate follow set analysis is an auxiliary analysis used by the follow set analysis. Its goal is to determine, for each nonterminal, what is the set of tokens that can occur *immediately* after the nonterminal. As described in the last section, a token can occur immediately after a nonterminal if it is in the local first set of a node immediately following a call to that nonterminal.

Consider this grammar

$$\begin{array}{lcl} \langle \mathsf{a} \rangle & \rightarrow & \mathbf{x}\ \langle \mathsf{b} \rangle\ \mathbf{y} \\ \langle \mathsf{b} \rangle & \rightarrow & \mathbf{z}\ \langle \mathsf{c} \rangle \\ \langle \mathsf{c} \rangle & \rightarrow & \mathbf{w} \end{array}$$

In this grammar, the immediate follow set of ⟨b⟩ is $\{\mathbf{y}\}$, because a **y** occurs immediately after an occurrence of ⟨b⟩. The immediate follow set of ⟨c⟩ is $\{\}$, because the only occurrence of ⟨c⟩ is at the very end of ⟨b⟩. This shows the difference between the follow set and the immediate follow set: while ⟨c⟩ may actually be followed by a **y**, through ⟨b⟩, **y** must occur immediately after the ⟨c⟩ to be included in the immediate follow set.

In the last phase we constructed, for each nonterminal x, a mapping that to a nonterminal y gave the set of tokens that can occur immediately after a call to y in x. If we take the union over all nonterminals that call y of the local follow set of y, we get the set of tokens that can occur immediately after a call to y somewhere in the grammar. In other words, a token t is in the immediate follow set of y if there is some nonterminal x where a call to y can be followed immediately by t. But doing this actually only gives us something that is *almost* the immediate follow set; the only thing missing is the end-of-file marker.

In traditional parsers a start symbol must be specified together with the grammar. The specified start symbol is not actually used as the starting point; instead, the real start symbol is constructed as this:

$$\langle \mathsf{real\text{-}start} \rangle \rightarrow \langle \mathsf{start} \rangle\ \mathbf{\$}$$

where ⟨start⟩ is the specified start symbol, and **$** represents the end of file marker.

As we will see later, this is very similar to what happens here, the main difference being that in Tedir, the start symbol is not specified when the parser is constructed, but rather when the parser is invoked. This means that any nonterminal could potentially be used as the start symbol, which again means that any nonterminal could potentially be followed immediately by the end of file marker. Hence, the immediate follow set of all nonterminals must contain the end of file marker.

5.7.1. Recalculation

When should it be recalculated? The immediate follow set of a nonterminal x depends on two things: which nonterminals call x, and what is the local follow set of x in those nonterminals. In the local first set analysis, we made sure to notice when there was a change in the local follow set of a nonterminal, including when there were changes in the set of nonterminals that called x. That set exactly gives us the set of nonterminals to recalculate.

$$\begin{aligned}\text{follow}(n) &\supseteq \text{immediate-follow}(n)\\ \text{follow}(n) &\supseteq \bigcup_{k \in R(n)} \text{follow}(k)\end{aligned}$$

where $R(n)$ is the set of nonterminals in which n occurs in a reducible position

Figure 5.14.: Definition of the full follow set.

Note that unless a nonterminal calls itself, this property depends exclusively on properties of the other nonterminals, not the nonterminal itself. That means that, unlike the previous analyses, this property is not necessarily recalculated for a nonterminal when it is changed. That is no problem for nonterminals that existed in the base grammar, because the immediate follow set calculated for there still be correct. For new nonterminals, on the other hand, there is no existing value to use if they are not recalculated. This means that we have to notice new nonterminals that are not recalculated, and explicitly give them the trivial immediate follow set $\{\mathbf{\$}\}$.

After the immediate follow set has been recalculated for all nonterminals for which it might have changed, we can find the set of nonterminals for which it actually *did* change. And then we are ready to calculate the full follow set.

5.8. Follow set analysis

The full follow set of a nonterminal can be expressed in terms of the immediate follow set as shown in figure 5.14. The first inclusion is trivial: at token can occur after a nonterminal if it can occur immediately after that nonterminal. The second clause states that if a nonterminal x occurs at the end of y (in a reducible position) then x can be followed by any token that can follow y. An example of this is section 5.7 on page 85, where the follow set of $\langle\mathsf{c}\rangle$ contains **y**, because **y** is in the follow set of $\langle\mathsf{b}\rangle$ and $\langle\mathsf{c}\rangle$ occurs at the end of $\langle\mathsf{b}\rangle$.

5.8.1. Which nonterminals to recalculate

For which nonterminals should the follow set be recalculated? Looking at the definition, we see that three things affect the follow set of a nonterminal. The first thing is the immediate follow set of the nonterminal – if that changes, then the full follow set should certainly be recalculated. Finding the set of such nonterminals was the last thing we did in the immediate follow analysis. The second thing is the set of nonterminals in which the nonterminal occurs in a reducible position, $R(n)$. The set of such nonterminals was found during the reducibility analysis.

Finally, the follow set of a nonterminal x depends on the follow set of all nonterminals in $R(n)$. This means that if y is marked to have its follow set recalculated and x is called from it in a reducible position in y, then x should also has to have its follow set recalculated. This is the transitive closure over the first two sets of the "occurs reducible in" relation.

5.8.2. Recalculating reducibility

Having found the set of nonterminals whose follow set should be recalculated, we now have to actually do the recalculation.

In the nullability and first set phases, the properties were (potentially) recursively defined, but since the grammars were free of left-recursion, no nonterminal ended up actually depending on itself. Unfortunately, that does not hold for the follow set: the definition is recursive, and it is quite possible for the follow set of a nonterminal to depend directly or indirectly on itself.

The traditional way to calculate the follow is iterative. Using the definition if figure 5.14, the algorithm goes as follows: initializing the follow set of all nonterminals to the empty set. Pick a nonterminal n and calculate $f(n)$, where

$$f(n) = \text{immediate-follow}(n) \cup \bigcup_{k \in R(n)} \text{follow}(k)$$

Then set the follow set of n to be $f(n)$. Do this iteratively until no further changes occur, no matter which n is picked. At this point, the follow set of all nonterminals has been found.

The algorithm used here is based on this approach, with a few improvements. The first improvement is that only some of the nonterminals have to be recalculated. For all the nonterminals not marked to be recalculated in the first step, the follow set can just be reused from the base grammar. The second improvement is that we can initialize the follow set of a nonterminal n to be the immediate follow set of n, rather than the empty set. Finally, when the follow set of a nonterminal n has been updated, we can explicitly find the set of nonterminals that might have been affected by the change and should be recalculated in the next iteration: that is the set of nonterminals occurring in n in a reducible position.

After applying this iterative algorithm, the follow set will have been found for all nonterminals. Now we are *finally* done analyzing the grammar and can get on with the last phase: updating the choice structures.

5.9. Choice update

Most of the work on constructing the choice tables was actually done during the local analysis: there, we found, for each choice node, the set of tokens that can occur first in each branch, and the branches where we had to take the follow set into account. We have now found the follow set of all nonterminals, so constructing the choice structure is straightforward. First, fetch the table that gives, to each branch, the set of token that can occur first. Then, fetch the follow set of the nonterminal where the choice occurs. For each branch that can produce the empty string, add the follow set to the set of tokens that can occur first. The result is a table that, to each branch, gives the set of tokens that indicates that this branch should be taken. Make the "inverse" table that gives, to each token, the set of branches that are applicable when that is the next token. This is the first set.

To give a concrete example, take the $\langle\mathsf{modifier}\rangle$ grammar:

$$\begin{array}{rcl} \langle\mathsf{modifier}\rangle[\mathsf{public}] & \rightarrow & \textsf{\textbf{public}} \\ [\mathsf{private}] & | & \textsf{\textbf{private}} \\ [\mathsf{none}] & | & \varepsilon \end{array}$$

Here we first take the table:

Branch	Local first	Reducible
public	{**public**}	no
private	{**private**}	no
none	{}	yes

then we add the follow set of ⟨modifier⟩ to the entries that can produce the empty string:

Branch	Tokens
public	{**public**}
private	{**private**}
none	{**byte**, **int**, **float**, ...}

and finally invert the table

Token	Branches
public	{public}
private	{private}
byte	{none}
int	{none}
float	{none}
⋮	⋮

For the last time, we get to consider the question of when the choice table should be recalculated. The choice table depends on three things: the local first sets of the branches, the reducibility of the branches and, if some of the branches can produce the empty string, the follow set of the nonterminal in which the choice occurs.

In the local analysis, we were careful to record the choices where the local first sets and the reducibility of the branches changed, so all the choices recorded there must be recalculated. In the local analysis we also recorded, for each nonterminal, the set of choices whose branches could produce the empty string. For each nonterminal whose follow set changed, all those choices should have their choice table recalculated.

Once all the choice tables have been calculated, the nonterminal bytecode, the choice tables, and all the other structures generated for the nonterminals are packaged as a single structure, the *nonterminal structure*. For all nonterminals where no choice tables were changed, the nonterminal structures from the base grammar can be reused. Finally, a map is constructed from the name of a nonterminal to the corresponding nonterminal structure. This structure is the parser, which is returned, together with the constructed grammar descriptor.

6. Parser Engine

The last section described how the parser bytecode and choice tables were generated; this section describes how that bytecode is executed.

6.1. Introduction

The parser engine is really an interpreter, executing the parser as if it were a program, and the nonterminals were functions. In some respects, the parser engine is simpler than ordinary interpreters – for instance, even though parsing a nonterminal corresponds to calling a function, nonterminals do not have local variables. In other respects, for instance with choice instructions, the execution model is more complex because it has to implement something similar to making nondeterministic choices. Also, the engine must handle the possibility of the current parser being replaced during execution.

This algorithm presented here is essentially a generalized version of the LL algorithm presented in 2.2. Several implementation exist of generalized LR parsers, and the possibility of generalizing the LL parsing model is well known. However, the approach presented here appears to be the first practical implementation of a breadth-first generalized LL parser, and the three-mode model presented here is, as far as I have been able to determine, the first practical adaptation of the theoretical model.

The previous chapters have already revealed much about how the engine works: an overview of tentative parsing and reparsing was given in section 3.7.2, a description of how suspension plays together with tentative parsing was given in section 3.9 and an overview of how the engine interprets the generated bytecode was given in section 5.3.1. What is left is essentially to describe how these different aspects fit together and fill in the remaining blanks.

This description is divided into two parts. The first part describes the different modes of execution used by the engine, unambiguous, tentative and reparsing mode. The first part will ignore the suspension aspect, which is the subject of the second part.

6.2. Unambiguous mode

As described in section 3.7.2, the engine operates in three different modes: unambiguous mode which, as the name indicates, is used when there are no ambiguities, tentative mode and reparsing mode. The parser cycles between these three modes: first it parses

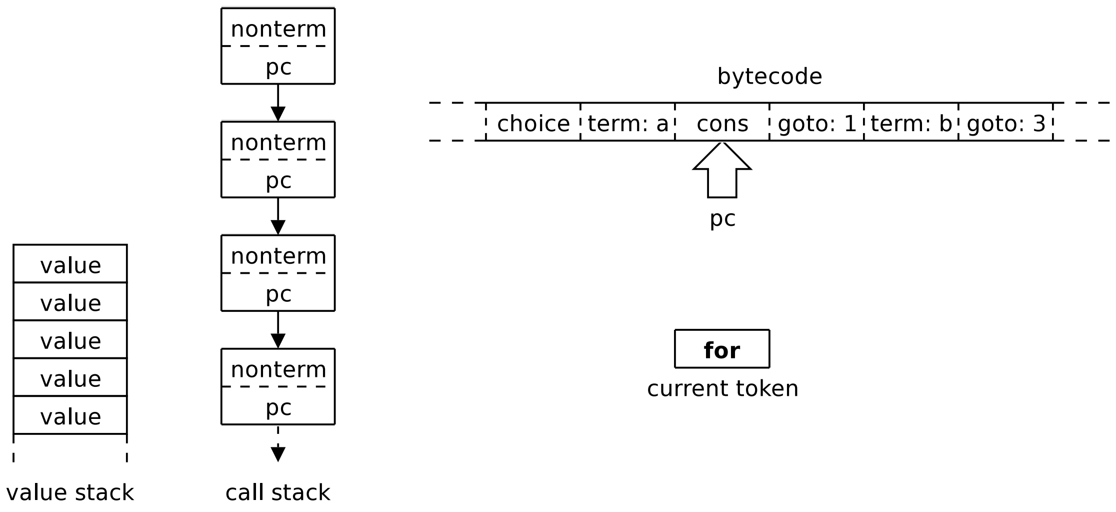


Figure 6.1.: Overview of a thread in unambiguous mode

in unambiguous mode until it reaches an ambiguity. Then it goes into tentative mode which reads forward until the ambiguity has been resolved, at which point is reparses the input that was parsed tentatively. After this, it goes back into unambiguous mode. This first section describes how unambiguous mode works, and the next section describes tentative and reparsing mode.

The description of the bytecode interpreter given in section 5.3.1 was said to describe a simplified model of how the engine works. Actually, what was given back then was a description of how unambiguous mode works. An overview of the data structures used by the engine in unambiguous mode was given in figure 5.6 on page 67 and is shown again in figure 6.1, with the one difference that here, the call stack is shown as linked. Why that is will be explained in the next section.

In unambiguous mode, the engine sequentially reads and executes bytecode instructions. It uses a call stack for nonterminal calls and returns, and builds expression values using a value stack. It keeps the current token in a register which is used when making choices. The only thing that was not covered back then was the details of how call/end and choice works.

When a call instruction is reached, the target of the call is looked up in the current parser. The parser gives, to the name of a nonterminal, the nonterminal structure of that nonterminal, which contains the bytecode of the nonterminal. To set up the return location, the current nonterminal structure and program counter are pushed onto the call stack. Then, execution can continue from the first instruction of the bytecode of the called nonterminal.

To return, an end instruction simply pops the top nonterminal structure and program counter off the call stack and continues executing from there. To get the correct semantics of parser replacement, it is important that it is a direct reference to the nonterminal structure that is pushed onto the call stack. Using the name of the nonterminal instead, so that the end instruction had to look up the corresponding nonterminal structure in the current parser, would break down if the current parser was replaced.

If a choice instruction is reached, and the choice cannot be resolved by looking at the lookahead token, it means that there is a local ambiguity, so the parser must go into tentative parsing mode to resolve the ambiguity. Because the parser has to go back later and reparse the input later, it must first record the state of the current thread: the current nonterminal structure, program counter and the state of the call stack. Then, it can start parsing the input tentatively.

6.3. Tentative parsing and reparsing

The basics of tentative parsing and reparsing was described in section 3.7.2. In tentative parsing mode, several threads try to parse the input in different ways. The threads are executed in lock-step, controlled by a main scheduler. When only a single thread is left, the ambiguity has been resolved. The parser then goes back to where the ambiguity was discovered and parses the input again, using the path that was found during tentative parsing. The first section describes the scheduler, the second describes how a single thread is executed, and the last section describes reparsing mode.

6.3.1. The scheduler

In tentative mode, there are a number of threads, whose execution is managed by a main scheduler. The scheduler gives control to each thread in turn, and a thread executes until it reaches a situation that prompts it to yield the control. There are three such situations. One possibility is that the thread dies. This happens if a thread fails to parse the input or if it reaches an ambiguity and is split up into a number of new threads; the latter situation will be described in a moment. A dead thread just stops executing and quietly disappears.

If a thread reaches a term instruction and this does not cause the thread to fail, it yields control to the scheduler. This is because, in tentative mode, the scanner is a resource shared between all the threads so all threads must be advanced, using the current token, before the next token can be read from the scanner. Once all threads have been advanced, the scheduler records the current token and token value, reads the next token from the scanner, and begins another step of advancing all threads. The scheduler makes a record of all the tokens and token values that are read, which will be used during reparsing mode.

The third situation that prompts a thread to yield control is when it reaches a join instruction and must prepare to be suspended. That situation will be described in section 6.4.1.

When the scheduler has made a step and all threads have been advanced, it checks to see how many threads are still alive. If all threads are dead it aborts and reports a syntax error. If there is more than one, that means that the ambiguity has not been resolved yet, and the threads must be advanced another step. If there is exactly one, a unique path has been found, and the parser can go into reparsing mode. Before describing reparsing mode, the next section describes how a single tentative thread is executed.

6.3.2. Tentative thread execution

Figure 6.2 gives an overview of the data structures used when executing a single tentative thread. Besides the absence of the value stack, there are a number of other differences from the threads used in unambiguous mode.

The call stack is now linked. This is because it allows the different threads to freely push and pop activations on and off the stack without interfering with each other. In a model using a vector- or array-based stack, that would be much harder to do efficiently. As mentioned before, the call stack was also linked in unambiguous mode, but back then, it didn't really make a difference.

A new structure has been added to each thread: the *path record*. The path record is a linked list that records which choices have been made, so that once only a single thread is left, the reparsing phase knows which path that thread represents.

A single tentative thread is executed the same way a an unambiguous thread except that no value stack operations are performed and no production actions are invoked. If a local ambiguity occurs when executing a choice instruction, the thread spawns a child thread for each of the possible paths. Each child thread is given a path record whose head records which choice the child represents, and whose tail is the path record of the parent thread. The child threads are then added to the end of the schedule, so that they are executed before the next token is read. Finally, the parent dies.

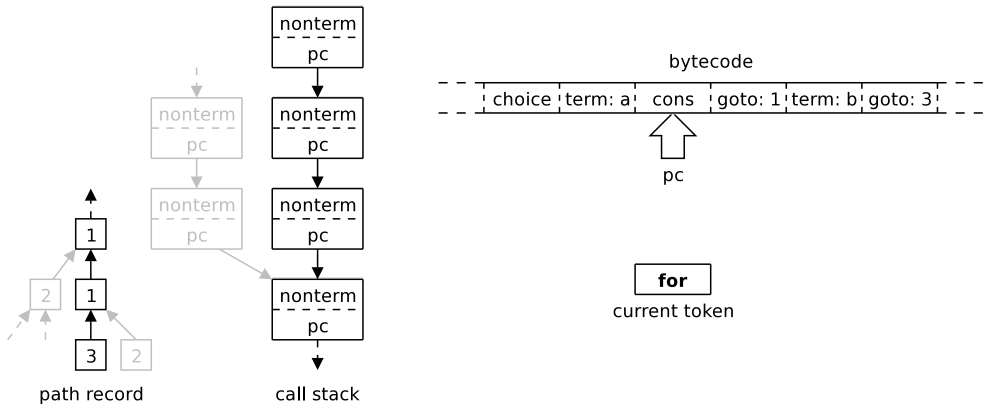


Figure 6.2.: Overview of a thread in tentative parsing mode

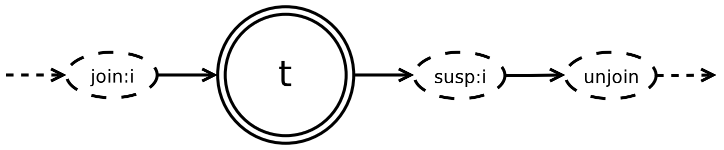
6.3.3. Reparsing

Just before going into tentative parsing mode, the state of the parser was stored. That is where reparsing mode starts. Reparsing mode works almost exactly the same way as unambiguous mode, except that is does not read tokens and values from the scanner, it reads them from the record made by the scheduler during tentative parsing mode. Also, whenever the thread reaches a choice instruction with more than one applicable path, it can choose the right path by using the path record stored in the surviving thread. Once it reaches the end of the token record, it has caught up with what was parsed while in tentative mode. This means that the parser can go back to parsing in unambiguous mode, and the cycle can start again.

So far, all the descriptions of the parser engine have been overviews and "not the whole story", and stopped short of giving all the details. So has the description above, because it has not described how suspension fits into all this. In the next section, I will give the last piece of the puzzle: a description of how suspend works.

6.4. Suspending the parser

Just as a reminder, this is how a suspend(t) expression was translated in section 5.3:



Together with this, a table was constructed that gives, to each join instruction, the address of the instruction following immediately after the corresponding unjoin instruction. This is used when the parser needs to skip over the whole expression.

The parser can only be suspended in unambiguous mode. This means that if the parser must be suspended during tentative parsing, all the tentative threads must first be joined into a single one which can be suspended. This first section describes how the parser is suspended in unambiguous mode, and the next section describes the extra work that must be done to join the threads in tentative parsing mode.

Suspending the parser in unambiguous mode is reasonably straightforward. The parser first parses the body of the suspend expression, which leaves a value on the value stack. It then pops that value off the stack and passes it to the suspend action. If parsing the body causes an ambiguity that has not been resolved when the susp instruction is reached, an error is signaled.

The suspend action either returns an object and signals that the parsing should continue as before, or returns an object, a scanner and a parser and signals that the current parser should be replaced before continuing. If it returns continue then the returned object is pushed onto the value stack as the value of the whole expression, the current

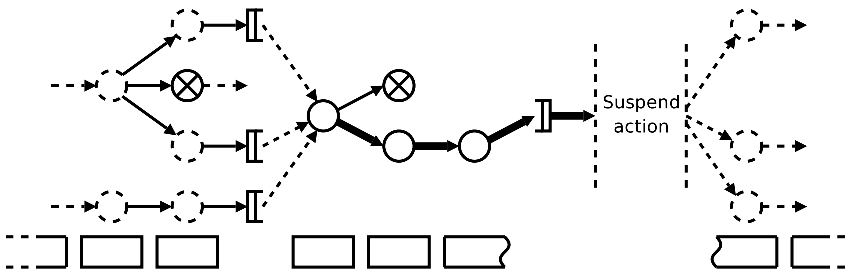


Figure 6.3.: Suspending the parser during tentative parsing

token is updated by rereading the current token from the scanner, and then parsing can continue as before. The current token must be reread from the scanner because the suspend action could have manipulated the input stream. If the action returns replace, the same thing happens except that before continuing, the current parser and scanner are replaced with the parser and scanner returned by the action. This means that the operation of extending the parser is very cheap: it only takes two assignments.

Why does that give the semantics of replacement described in section 3.11.1? Two things defined the semantics: replacement does not affect the nonterminals that have already been partially parsed, and all new calls to nonterminals are parsed using the new parser. The key to this is that the nonterminal structure for a nonterminal is read from the current parser exactly once: at the point where the nonterminal is called. After that, the nonterminal structure is always used directly. That means that after replacing the parser, when the parser returns to a nonterminal that is in the middle of being parsed, the definition used will be the one stored in the stack, from when the nonterminal was originally called, not the new definition. On the other hand, all new calls are executed by fetching the called nonterminal from the current parser, which will be the new parser.

6.4.1. Suspending during tentative parsing

Section 3.9 gave an overview of how suspension plays together with tentative parsing. The two overview figures from that section are shown again in figure 6.3 and figure 6.4 on the facing page. What happens is essentially that the tentative threads are joined into one, which is used to parse the body of the suspend expression (figure 6.3). Then the threads are unjoined and can continue parsing as before. When reparsing the input, the suspend expression is simply skipped, since it has already been parsed (figure 6.4).

Whenever a thread reaches a join instruction, it yields control to the scheduler. When all threads have been advanced, the scheduler checks to see if they have all reached the same join instruction. Two threads have reached the same join instruction if they are executing the same bytecode program and have the same program counter, which points to a join instruction. If not, an error is raised. Otherwise, the parser creates a single new thread for parsing the suspend expression, and executes that thread in unambiguous

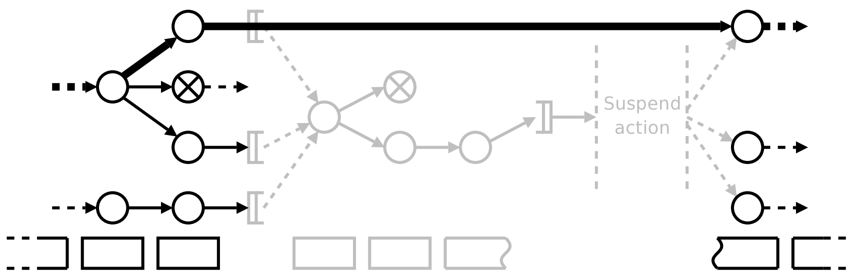


Figure 6.4.: Reparsing a suspend expression after disambiguation

mode, starting with an empty call and value stack. The thread executes as usual until it has executed the suspend action and reached the unjoin instruction. At that point, the value stack contains a single value, the value returned by the suspend action. As before, if an ambiguity occurs while parsing the body of the suspend expression, that is not resolved before reaching the unjoin instruction, an error is signaled.

Before the parser can continue to execute the tentative threads, the scheduler must make a record of the suspension in the *suspend record*. For each time the parser is suspended, the scheduler stores the effects of the suspension in this record. This will be used during reparsing, where the parser has to replay the parsing process, including the suspensions. Specifically, when the parser is replaced, this must be recorded so that the parser can also be replaced during reparsing. There is no need to record that the scanner was replaced, since the token record is used during reparsing, so tokens are not read directly from the scanner.

Two things are recorded: the value of the suspend expression returned by the unambiguous thread, and the parser that was the current parser after the suspension. Having recorded this, the scheduler updates the current token and continues parsing the threads as before.

After tentative parsing mode, the parser reparses the input. The interpreter must first reset the current parser to be the parser used when the ambiguity was found. Whenever a join instruction is reached, the suspend record gives the value of, and the parser to use after, the suspend expression. The value is pushed onto the value stack, and the current parser is replaced by the specified new parser (which is a no-op if the parser was not actually replaced). Finally, the next token is read from the list of tokens and reparsing can continue, starting from immediately after the unjoin instruction.

6.5. Extensions

The model presented here is very open to extensions. One important extension is allowing productions to be given relative priorities. Production priorities can, for instance, be convenient for resolving the dangling **else** ambiguity:

$$\begin{array}{lcl} \langle\mathsf{stmt}\rangle & \xrightarrow{2} & \textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \langle\mathsf{stmt}\rangle\ \textbf{else}\ \langle\mathsf{stmt}\rangle \\ \langle\mathsf{stmt}\rangle & \xrightarrow{1} & \textbf{if}\ \langle\mathsf{expr}\rangle\ \textbf{then}\ \langle\mathsf{stmt}\rangle \end{array}$$

This way, if it turns out that both rules apply, the one with the highest priority wins. This can be implemented by associating, with each thread, a list of threads with lower priorities. If the thread reaches the end of the ⟨stmt⟩ production, it has succeeded, and can go through the list and kill all the threads that have lower priority that are still alive. It sounds evil, I know, but is relatively straightforward to implement. Many other extensions are possible, and the open structure of the parser means that many of them can be implemented without major changes to the basic algorithm.

This concludes the description of the algorithms implementing the library. The next chapter investigates the efficiency of the prototype implementation of the library and the algorithms.

7. Experimental Results

This chapter presents the results of the performance experiments run with the prototype implementation of the library. There are two interesting aspects of this: how fast is the delta application process, and how fast is the generated parser. The performance results for each of these aspects are given in the next two sections.

All the grammars used in the experiments are based on the Java 1.4 grammar. There is a number of reasons for this. First of all, the Java grammar is relatively complex, but also a very clean context-free grammar. It does not, for instance, rely on indentation (like Python or Haskell) or depend on the occurrence of type declarations (like C and C++). It also has the advantage that huge amounts of Java 1.4 source code is freely available as test data. Finally, a grammar specification for Java 1.4 is available for most other parser generators for Java, which makes comparison much easier.

All performance tests were timed on a 1.5 GHz Intel Pentium M processor running Linux version 2.4.20-8. All Java files were compiled using the Java 1.5.0 beta 1 compiler and executed on the HotSpot 1.5.0 beta 1 virtual machine. The timing was done using Java's builtin timing function, System.currentTimeMillis().

7.1. Compilation algorithm

The performance of the compilation algorithm was tested by measuring the time of first compiling a basic Java 1.4 grammar, and then applying different extensions to the result.

Five different extensions were applied: a for each statement, like the one added in Java 1.5, the concise list syntax described in section 1.2.2 on page 4, and which is used in many functional languages, and a local function syntax that allows functions to be declared within methods. Finally, two extensions were borrowed from the Pizza Java dialect (Odersky & Wadler 1997): a syntax for anonymous functions and an extension allowing algebraic datatypes and pattern matching. The results, taken as an average over 1000 runs, are shown in figure 7.1 on the following page.

The results show that a large grammar, such as Java 1.4, can be compiled in less than 15 ms., and that a range of different extensions can each be applied in less than 1 ms. Two things can be concluded from this. The one thing is that for limited purposes, the compilation algorithm is probably fast enough that a simpler non-incremental version will be adequate. Especially since, without having to trace dependencies, the basic algorithm can be made to run even faster.

The other thing to conclude from this is that incremental compilation does actually

Syntax	Time (ms.)
Java 1.4	14.3
For each	0.65
List syntax	0.84
Local function	0.45
Algebraic datatypes	0.76
Anonymous functions	0.76

Figure 7.1.: Execution times for grammar compilation

give a considerable speedup. In this case, compiling the extensions is over 20 faster, on average, than compiling the grammar from scratch. In applications that make heavy use of syntax extension, for instance with local macros, incremental compilation can speed up the compilation process enough that, with extensions such as the ones described above, the library can generate parsers for well beyond a thousand dialects per second.

7.2. Parser algorithm

Being able to generate parsers quickly is all well and good but not much use if the parsers generated do not parse input efficiently. The worst-case time complexity of the parsing algorithm presented her is exponential, so it is both interesting to see how fast the parses are, and how the execution time behaves as a function of the size of the input files. This section compares the performance of a Java parser generated by Tedir with similar parsers generated by two widely used parser generators: JavaCC (version 3.2) and ANTLR (version 2.7.3). Both of these are LL parsers.

7.2.1. Test setup

The parser that was constructed by JavaCC was generated from the Java 1.4 grammar specification available from the JavaCC grammar repository. The ANTLR parser was generated from the Java 1.4 grammar that is distributed with the tool, where all the code for generating ASTs had been manually removed. In both cases, the scanner used was the one generated by the tool itself.

The grammar used by Tedir was more or less taken directly from the definition given in the Java language specification (Gosling, Joy, Steele & Bracha 2002). The main difference is that some idioms used in the specification, for instance for specifying repetitions with infix separators, were replaced with the appropriate operators in Tedir. A conscious effort was made to make the grammar as straightforward as possible and not optimize for efficiency. The scanner used was generated by the JFlex scanner generator.

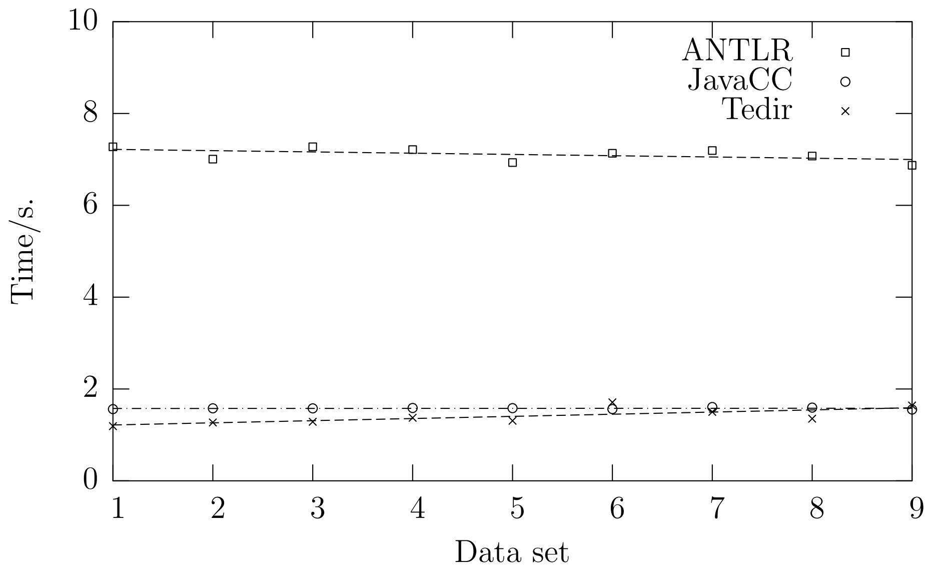


Figure 7.2.: Parsing speed of Tedir compared with JavaCC and ANTLR

7.2.2. Data

The test data was around 60 MB of Java source files, taken from the source of Eclipse, which is an open-source, Java-based IDE. A few files were left out, either because JavaCC and ANTLR would not accept them, or because they contained dangling else ambiguities that Tedir could not handle.

The test data was divided into groups of equal size, according to the size of the files, so that all the files in group n were smaller than the files in group $n + 1$. The performance was timed by parsing each group 15 times and taking the average over the last 10 times – the first few runs always take longer because the files are not yet cached in memory. The result of the comparison is shown in figure 7.2.

From the graph, it is clear that JavaCC and Tedir are the fastest, ANTLR being around five times slower. The graph in figure 7.3 on the following page shows only JavaCC and Tedir.

Figure 7.3 on the next page shows that while JavaCC is completely unaffected by the file size, the throughput in Tedir slowly decreases as the input files become larger. Figure 7.4 on the following page shows the same data as figure 7.3 on the next page, but with the execution time as a function of the average size of the files in each group. Luckily, very large source files are actually relatively rare in practice. Out of the total data set, which contains almost 7000 source files, only around 100 are larger than 40 Kb. Of course, since those files are larger, those few files account for around 10% of the total size of the data set. And the decrease in throughput is so slow that the possibility of exponential time complexity not a problem in practice.

The average throughput, as measured over the whole data set, is shown in figure 7.5.

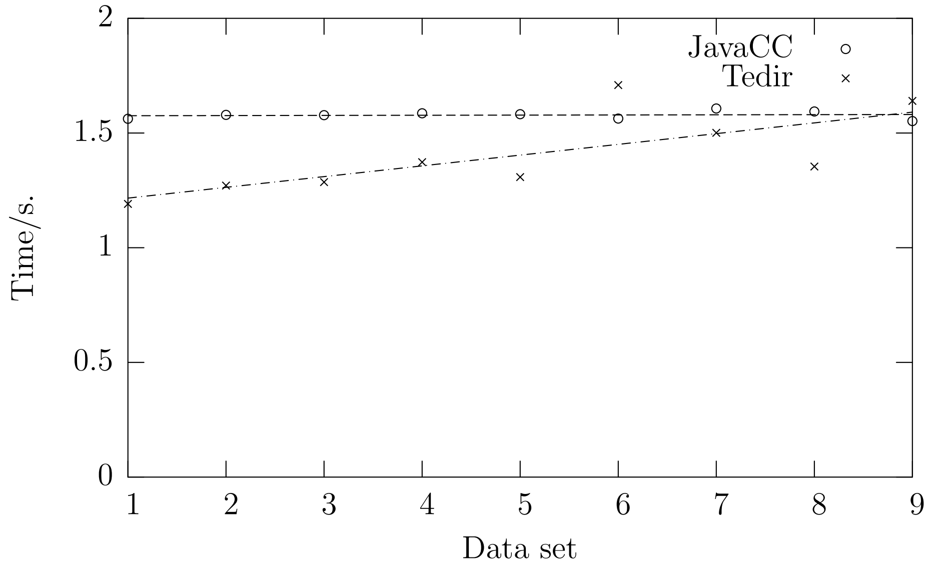


Figure 7.3.: Parsing speed of Tedir compared with JavaCC

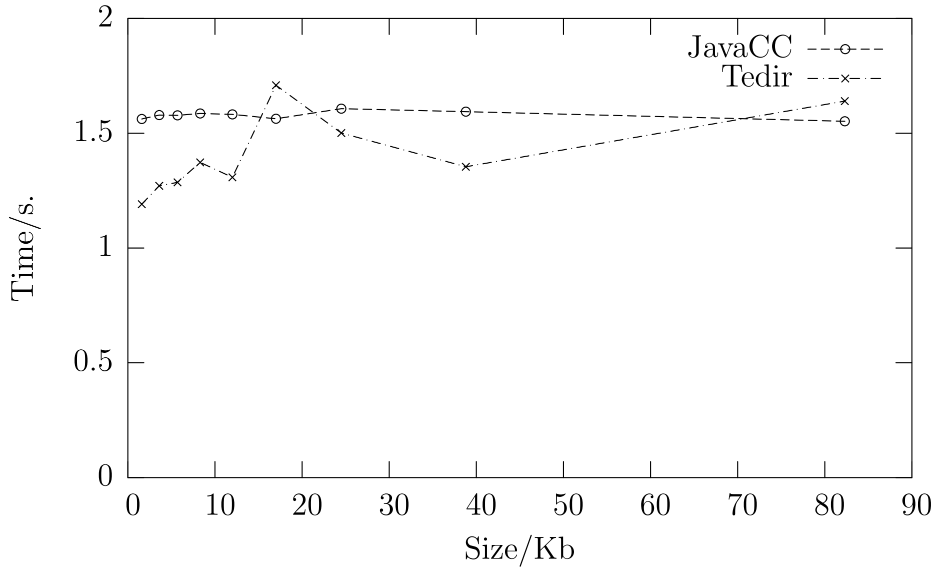


Figure 7.4.: Parsing speed as a function of the average file size

Parser	Throughput (Kb/s)
ANTLR	930
JavaCC	4,188
Tedir	4,708

Figure 7.5.: Average throughput of each parser

Over all, in these measurements, Tedir is just over 10% faster than JavaCC. Note, however, that this should not be taken to mean that Tedir is generally faster than JavaCC, because some factors that affect the throughput, for instance the fact that the three parsers all use different scanners, have not been taken into account.

Two conclusions can be drawn from this. First of all, there is no reason to use a static parser for efficiency reasons. The parsers generated by Tedir can clearly be at least as efficient as the statically generated parsers. Secondly, and most importantly, even with a completely straightforward and un-optimized grammar, Tedir is so fast that making further optimizations to the grammar simply are not worthwhile.

7.2.3. Execution characteristics

This section discusses some characteristics of the execution of the parser. This doesn't say anything about the efficiency of the parser, but gives an insight into how the parser engine behaves dynamically.

Figure 7.6 shows the number of bytecode instructions executed by the parser engine in the course of parsing the whole data set, divided according to the mode in which the instructions were executed. In total, it takes around 97 million instructions (MI) to parse the 60 MB of Java files.

The instructions executed in tentative mode are only executed to disambiguate the input, not to actually parse it; that is handled during reparsing mode. So if we subtract the number of intructions executed during tentative parsing, we get that it takes around 61 MI to actually parse the input without considering disambiguation. Of these instructions, 47.7 MI were executed in unambiguous mode and 13.2 MI were executed in reparsing mode. This means that a little less than 80% of the input could be parsed unambiguously, and disambiguation was neccesary for just over 20% of the input. Dur-

Mode	Instructions (MI)
Unambiguous	47.7
Tentative	35.9
Reparsing	13.2

Figure 7.6.: Instruction count over the whole data set.

ing these 20%, 35.9 MI were executed to do disambiguation, and since it takes a single thread 13.2 MI to parse this input in reparsing mode, that corresponds to having an average of 2.7 threads during tentative parsing.

To sum up, this tells us that with a straightforwardly written grammar 80% of the input can be parsed unambiguously. Of the remaining input, an average of 2.7 threads were used in tentative parsing mode.

8. Related Work

Almost all the issues discussed in this thesis have been the subject of prior work. Usually, however, this prior work deals with the issues in a different context, or with different goals. Each of the following sections review an area in which work related to this thesis has been done.

8.1. Parser libraries

Most of the current academic research into the generation of parsers using run-time libraries is concerned with *parser combinators* (Hutton 1992). Ordinary parsers, including Tedir, only allow parser to be described using a fixed set of operators, such as φ^* and $[\varphi]$. With parser combinators, operators are represented as higher-order functions that can be freely combined to define new operators. An example of the specification of a nonterminal is this function definition (adapted from Hutton (1992)) in the Miranda language:

```
expr = (((term) $then (literal '+') $xthen (term)) $using plus)
  $alt (((term) $then (literal '-') $xthen (term)) $using minus)
  $alt term

term = ...
```

This function defines a parser this production:

$$\begin{array}{lcl} \langle\text{expr}\rangle & \rightarrow & \langle\text{term}\rangle + \langle\text{term}\rangle \\ & | & \langle\text{term}\rangle - \langle\text{term}\rangle \\ & | & \langle\text{term}\rangle \end{array}$$

The parser is invoked by simply applying the `expr` function to a string.

The operators used (`then`, `alt`, `xthen` and `using`) are all simple (one- or two-line) higher-order functions. The `$` notation is used to make functions into right-associative infix operators. `then` and `xthen` are sequencing operators, `alt` is the alternation operator and `using` is used to specify a function to be applied to the result of parsing an expression. In this case, the `plus` and `minus` simply add two integers.

As in Tedir, the value of an expression is "packaged" during parsing, by the combinators, and then passed to a function, which means that it is convenient to be able to discard irrelevant values. In (Hutton 1992), two alternative sequencing operators are defined, `xthen` and `thenx`, which discard the left and right values respectively. This way, using `$xthen` in the example above essentially marks the left-hand term, the plus or minus operator, as being irrelevant.

Even though parser combinators can be, and have been, implemented in non-functional languages (Dijkstra & Swierstra 2001), they are predominantly used in functional languages. Much of the work on parser combinators has been theoretical, but practical libraries exist (Leijen & Meijer 2001). Unfortunately, parser combinators tend to be slow, and are hardly used in practice.

Besides combinator-based libraries, there are a number of libraries implementing traditional parsing algorithms. The most popular library seems to be the Parse::RecDescent module for the Perl language (Conway 2003). Parse::RecDescent accepts a flat text string containing the description of a grammar and returns a parser for that grammar (given that the description was valid). This example shows how a parser for the grammar shown above would be defined (the `q { ... }` form is a string-quoting operator):

```
my $grammar = q {
  expr : term '+' term
       { $return = $item[1] + $item[3]; }
       | term '-' term
       { $return = $item[1] - $item[3]; }
       | term
       { $return = $item[1]; }
  term : ...
};
my $parser = new Parse::RecDescent($grammar);
```

The specification contains both the grammar definition and the text of the production actions. Defining grammars this way is convenient, especially when prototyping, but has all the weaknesses inherent in using plain text to define structured data in a running program. This is not a fundamental problem, however, and building an interface similar to Tedir's on top of the plain text interface would be straightforward. The main application for Parse::RecDescent is probably prototyping, especially because the generated parsers tend to be relatively slow.

8.2. General context-free parsers

Some tools allow parsers to be constructed for general context-free grammars, rather than a fixed subset like LL(1) or LALR(1). Like Tedir, those parsers must deal with grammars where ambiguities cannot be resolved locally, and if parsing involves executing user-supplied actions, they must have a suitable execution model for that.

There are many algorithms for general context-free parsing, but the most popular approach is generalized LR or *GLR* parsing (Tomita 1986). Roughly, GLR parsing is to LR parsing what the parsing algorithm presented here is to LL parsing. It resembles the LR parsing algorithm (see section 2.3 on page 13), except that in cases where an LR parser cannot decide whether to shift or reduce, or has more than one way to reduce, a GLR parser will try out all the possibilities in parallel.

The ASF+SDF system (van den Brand, Scheerder, Vinju & Visser 2002) uses a scannerless GLR algorithm, which handles both lexical analysis and parsing. It deals with ambiguities by keeping a forest of possible parse trees. During parsing, this forest is pruned using user-specified *disambiguation filters*. Disambiguation filters can, for instance, specify relative preferences, so that some trees are preferred over others, or specify that trees of certain shapes should always be rejected. The hope is that at the end, there will only be a single tree left. The ASF+SDF parser is an integrated part of the ASF+SDF meta-environment and cannot be used independently. This means that it does not have to deal with the issue of how user actions are executed.

The Bison parser generator (Donnelly & Stallman 2002) has recently been extended to allow it to generate GLR parsers. It handles the execution of user actions while resolving ambiguities in essentially the same way as Tedir: as long as there are unresolved ambiguities, the parser records the actions to be performed. No actions are executed until the ambiguities have been resolved.

Tedir is somewhat unusual in that it uses a top-down parser that allows ambiguous grammars. As mentioned above, the goal of most such parsers is to be able to parse general context-free grammars, and because of the left-recursion restrictions on parsers based on the top-down approach, top-down parsers are unsuitable for that purpose. Grune & Jacobs (1990) report that no research on generalized top-down parsers had been reported in the literature at that point, and that seems to still be true.

8.3. Grammar extension

Some tools allow grammars to be specified "by difference", by defining one grammar to be an extension of another.

The popular ANTLR parser generator (Parr 2004) for Java, C++ and C#, allows grammars to be extended using *grammar inheritance*, which is very similar to ordinary object-oriented inheritance. When specifying a grammar, you can specify that the grammar should inherit all the productions of another grammar. This is used as a mechanism for collecting the common parts of several grammars in a single module in a way that allows changes in the base grammar to be automatically propagated into the derived grammars. It is implemented as a preprocessing phase where the derived grammar is constructed by copying the base grammar and applying the specified changes. A parser can then be generated for the resulting grammar as usual.

The `metafront` system (Brabrand, Schwartzbach & Vanggaard 2003) is a tool for making transformations between context-free grammars. This can be used to make translations between completely different languages, for instance from Java to HTML, or from an extended language to the base language, in which case it essentially works as a macro processor. As in ANTLR, a grammar can be declared to extend another grammar, in which case the extended grammar inherits all productions and actions from the base grammar.

Whereas the inheritance mechanism in ANTLR is mainly useful for managing a well-known set of similar grammars, in `metafront`, there is much more focus on the extensions themselves. For one thing, extensions can be separately checked for consistency, before being used. Also, to allow grammars to be written as straightforwardly as possible, and to allow extensions to be developed separately, `metafront` uses *specificity parsing*, which is a variant of LL parsing. This approach makes it possible to parse according to grammars that are written straightforwardly, and to give error messages in terms of the individual productions of the extensions.

8.4. Incremental parser generation

In the parsers described in the previous section, the extension mechanism was implemented as a preprocessing step, and the algorithms for constructing a parser for the extended grammar were completely unaware of whether a grammar was an extension of an existing grammar or not. Some work has been done to make the parser construction process incremental so that the generation of a parser for an extended grammar does not require the whole grammar to be recompiled. The motivation for making the generation of parsers incremental is simple: it is faster than generating the parser from scratch. Much of the early work on this was concerned with making traditional parser generators faster, but interest in this subject seems to have decreased as the the speed of processors increased – the effort that goes into making traditional parser generators incremental just doesn't seem worthwhile anymore.

The ILALR (Horspool 1990) parser generator is an LALR(1) parser generator, designed to assist in *grammar debugging*. Using ILALR, a user can interactively specify a grammar and the corresponding parser will be updated whenever a production is added. This way the user can see, each time a production is added, if there are shift/reduce or reduce/reduce conflicts in the grammar, and interactively fix the problem if there are. Unlike Tedir, there is no reason to keep the original parser after an update so the algorithm used in ILALR updates the parser destructively.

Even though the ILALR algorithm cannot be applied to the compilation process in Tedir because ILALR constructs LR parsers, the process of constructing an LL parser and an LALR(1) parser have much in common. In particular, they both have to incrementally calculate the nullability, first and follow sets. In ILALR, they are first recalculated for the parts of the grammar that might have been affected, using a fast but approximate algorithm. The approximation may err by spuriously detecting conflicts, so whenever a conflict is found, an accurate algorithm is applied to that part of the grammar. If the conflict is still there after the accurate analysis, it is reported to the user.

This way, a working parser can for the most part be generated cheaply by the approximate algorithm, and the accurate analysis is only applied when there is an indication that the approximation is too inaccurate. Unfortunately, the algorithm only performs efficiently when new productions are added, not when existing productions are changed or removed – in those cases, it suggests that the nullability, first and follow sets should be recalculated from scratch. Adapser, which is described in the next section, takes a similar approach, but allows productions to be both added and removed.

Horspool (1990) reports that some work has been done in incremental generation of LL parsers, but that the publications have not been widely distributed. In particular, he mentions a PhD dissertation (Vidart 1974) about incremental generation of LL(1) parsers, but I have been unable to find any further information about this.

8.5. Dynamically extensible parsers

The algorithm described in the previous section is a static extension algorithm in the sense that extensions take place before the parser is used. It does not seem that much work has been done in dynamically extensible parsers. Two dynamically extensible parser libraries exist, but there is very little information available about how they are implemented.

Parse::RecDescent, which was described in section 8.1, allows a running parse to be extended with a set of new productions. A set of productions can be added in one of two ways. Either the productions can be added directly, or they can be added after all existing production have been removed from the affected nonterminals. This example adds two new productions to the parser defined in section 8.1:

```
my $extension = q {
  expr : '-' term
       { $return = -$item[2]; }
  term : factor '%' factor
       { $return = $item[1] % $item[3]; }
};
$parser->Extend($extension)
```

The parser is updated by simply doing a complete recalculation. Apparently, parsing can continue as before after the parser has been replaced, the same way as in Tedir. If so, it is likely that the semantics of an update is also the same as in Tedir. This is all guesswork, however, because the only source of information about the details of this is the source code of the library, and I am far from fluent in Perl.

Adapser (Carmi 2002) is an adaptive LALR(1) parser for C++, which appears to share many of the goals of Tedir. For instance, one of its goals is to be able to parse languages that provide constructs for changing their own syntax. It also allows parsers to be extended during parsing but, like ILALR, does so by destructively updating the current parser.

The system uses a lazy approach similar to the one used in ILALR, but unlike ILALR, the parser is generated on demand during parsing. This means that if some part of the grammar is not used in the program that is being parsed, the part of the parser that handles this part of the grammar is never even constructed. The paper cited above reports that an interpreted, strongly typed, C-like language has been implemented using only grammar modifications, without using symbol-tables or a runtime stack. Unfortunately, besides the short paper cited above, no information seems to exist about Adapser. In particular, no detailed information is available about the algorithms used in the implementation.

8.6. Object-oriented parsers

One area that gives some interesting solutions to some of the issues we have seen here are the object-oriented parsers. Object-oriented parsers map a grammar definition into an inheritance hierarchy, with a class for each nonterminal in the grammar.

One example is the Mjølner BETA metaprogramming system (Mjølner 1996), which is a metaprogramming system for the BETA programming language. Since there is a direct correspondence between nonterminals and classes, the nonterminals are restricted to have a certain shape that can be expressed in terms of classes. An example of a grammar specification using this *structured context-free grammar* form is shown in figure 8.1 on the facing page, here using a slightly different notation than the one used in the BETA metaprogramming system. The corresponding class hierarchy is shown in figure 8.2 on the next page.

The example uses three of the forms that can be used in the definition of a structured context-free grammar. A nonterminal can be declared as a choice between a number of other nonterminals, as ⟨Stmt⟩ in the example. This imposes a subclass relation between the nonterminals, so the classes corresponding to the nonterminals on the right become subclasses of the class corresponding to the nonterminal on the left. Nonterminals can also be declared as a sequence of terminals and nonterminals, as the individual statements in the example. This defines a single class with an attribute for each named component of the left hand side. The last nonterminal ⟨StmtList⟩ is declared as a possibly empty list of nonterminals, here ⟨Stmt⟩. The class constructed corresponding to ⟨StmtList⟩ is a subclass of the List class. Besides the three forms used in the example, nonterminals can also be declared as a non-empty list of nonterminals ($\xrightarrow{+}$) or an optional nonterminal ($\xrightarrow{?}$).

This gives an interesting solution to the problem of representing the result of parsing an expression. Whenever a nonterminal is parsed, an object of the corresponding class is constructed, except for the nonterminals that represent choices between other nonter-

$$
\begin{array}{rcl}
\langle\text{Stmt}\rangle & \rightarrow & \langle\text{ShortIfStmt}\rangle \\
 & \mid & \langle\text{LongIfStmt}\rangle \\
 & \mid & \langle\text{WhileStmt}\rangle \\
 & \mid & \langle\text{BlockStmt}\rangle \\
 & \mid & \langle\text{ExprStmt}\rangle \\
 & & \\
\langle\text{ShortIfStmt}\rangle & \rightarrow & \textbf{if}\ \langle\text{cond: Expr}\rangle\ \textbf{then}\ \langle\text{thenPart: Stmt}\rangle \\
\langle\text{LongIfStmt}\rangle & \rightarrow & \textbf{if}\ \langle\text{cond: Expr}\rangle\ \textbf{then}\ \langle\text{thenPart: Stmt}\rangle\ \textbf{else}\ \langle\text{elsePart: Stmt}\rangle \\
\langle\text{WhileStmt}\rangle & \rightarrow & \textbf{while}\ \langle\text{cond: Expr}\rangle\ \textbf{do}\ \langle\text{body: Stmt}\rangle \\
\langle\text{BlockStmt}\rangle & \rightarrow & \{\ \langle\text{stmts: StmtList}\rangle\ \} \\
\langle\text{ExprStmt}\rangle & \rightarrow & \langle\text{expr: Expr}\rangle\ \textbf{;} \\
 & & \\
\langle\text{StmtList}\rangle & \overset{*}{\rightarrow} & \langle\text{Stmt}\rangle
\end{array}
$$

Figure 8.1.: A structured context-free grammar for statements

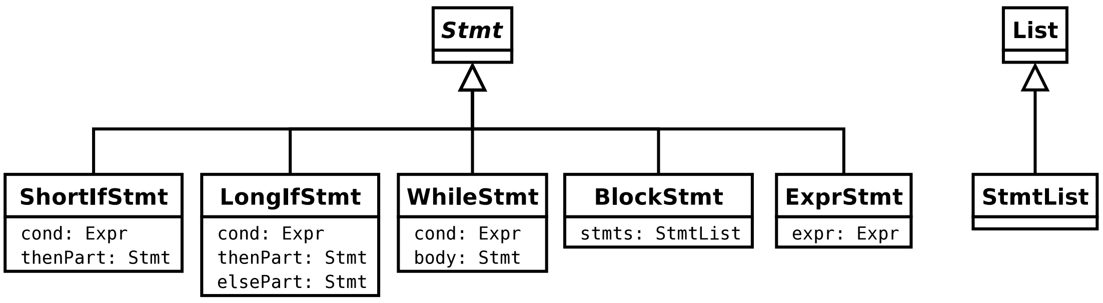


Figure 8.2.: Class hierarchy constructed for the statement grammar.

minals. When parsing a ⟨IfStmt⟩, there is no need to construct a ⟨Stmt⟩ object, because the information that an ⟨IfStmt⟩ is a ⟨Stmt⟩ is implicit in the inheritance hierarchy.

Koskimies & Vihavainen (1992) use the representation of nonterminals as classes to make it possible to develop grammar modules independently. In this model, the classes that represent nonterminals can be developed independently, in the sense that changing the definition of one nonterminal does not require a recompilation of the classes representing the nonterminals of the surrounding grammar. Note that these are changes to the definition of the grammar, before it is used, not dynamic changes. This separate compilation of nonterminals is accomplished by using *lazy recursive descent parsing*, where the structures that are necessary for doing the actual parsing are constructed on demand when the parser is used.

9. Conclusion

In this thesis, I have presented the design of a dynamically extensible parser library. An important part of the motivations for building such a library was to make syntax extension possible for users. Because of this, there has been a strong emphasis in the design on making the library useful for non-experts. I have argued that the result is appropriate for this purpose.

Of course, a really good library design requires continuous use and refinement. What I have presented here is only my *idea* of how a library should be designed to be practically useful; it is unlikely that Tedir, in version 1.0, will be the library to end all parser libraries. But I believe that library is not only highly useful at it is, but that it provides an excellent starting point for further refinement. In particular, it should be clear that the library allows a clean implementation of the examples given in the introduction.

To implement the library, I have developed two new algorithms: an algorithm for incremental generation of LL parsers and a practical algorithm for generalized breadth-first top-down parsing. These algorithms have been developed as a long evolutionary process, and can be evolved even further. The main direction for future work on these algorithms is to implement features such as production priorities, better error reports and error repair mechanisms. The parser execution model is very open, and many of these features can be implemented relatively straightforwardly as extensions without requiring changes to the basic model.

Finally, I have implemented a prototype of this library and demonstrated that these algorithms are efficient. In particular, I have shown that the parsing algorithm parses input at least as efficiently as traditional, statically generated parsers. This has also demonstrated the it is in fact possible to efficiently parse according to grammars that are not LL(k) or LALR(1), but have been written completely straightforwardly. The parsing algorithm presented here is not only applicable in this special case, but can also be used in the implementation of traditional statically generated parsers. This also means that many of the usability considerations that defined this library can be applied to traditional parser generators, without making the resulting parsers less efficient.

In the introduction, the need for extensible parsers was motivated by the increasing need for configurability in tools, and the limitations caused by current parser technology. In some cases, such as the SQL example in the introduction, the configurability is limited to being able to combine a few pre-defined building blocks. For applications like that, the library can be applied directly.

But the motivation for designing this library was the need for much more flexible methods of syntax extension, as in the other two examples given in the introduction.

Abstraction is power, and there is a continuing tendency in programming languages towards providing higher and higher levels of abstractions, and to allow the programmer to create new abstractions. But without extensible parsers, there is a hard, technically imposed, limitation on an important aspect of designing new abstractions, the syntax. The goal of this thesis has been to remove this limitation, and I believe that this goal has been reached. But the parser is only a tool, and developing the mechanisms to use it will be at least as much a challenge as developing the parser itself. I believe that the development of more flexible mechanisms for syntax extension is by far the most interesting direction for future work.

A. Library prototype

The CD-ROM enclosed with this thesis contains the prototype implementation of the Java library. The source code is available under the `src/` directory. It also contains two programs that demonstrate some of the features of the library: an extensible Java parser and a parser for a simple statement grammar that allows locally scoped grammar extensions.

Running the example programs requires the 1.5 version of the Java platform, which is available in a beta version from `http://java.sun.com/`. Under Unix, the examples can be run as shell scripts. Otherwise the corresponding `.jar` files must be executed manually. The jar files are available in the `jar/` directory.

A.1. Extensible Java parser

The extensible Java parser can be run using the `extend` shell script in the root directory of the CD-ROM or by running the `jar/extend.jar` file. This example can be used to experiment with parsing different extensions to the Java grammar. The program accepts three command-line arguments:

- The name of a file containing the Java grammar. A java grammar specification is available in `grammars/java.grm`.
- The name of a file containing an extension to the Java grammar. A number of example extensions are available under `grammars/`, for instance an extended for loop (`java-for.grm`) and the compact list syntax described in section 1.2.2 (`java-list.grm`).
- The name of a Java file to be parsed

The program will first read the Java specification and produce a parser for the specified grammar. It will then read the extension and apply it to the Java grammar. Finally, it will use the extended parser to parse the specified Java input. The following gives an example of how `extend` can be used.

The source file `Test.java` uses the extended list syntax:

```
class Test {
    List objs = ["a", "b", "c"];
}
```

Trying to parse this file according to the unextended Java grammar causes an error. The program requires that an extension is specified, so the empty extension, `grammar/java-none.grm`, is specified:

```
$ ./extend grammars/java.grm grammars/java-none.grm Test.java
Reading grammars/java.grm...
Reading grammars/java-none.grm...
Parsing Test.java...
Exception in thread "main" com.tedir.SyntaxError: 14
        at com.tedir.Interpreter.interpret(Interpreter.java:110)
        at com.tedir.Parser.parse(Parser.java:27)
        at com.tedir.Parser.parse(Parser.java:31)
        at com.test.Test.main(Test.java:35)
```

As this example shows, the error reporting facilities in the prototype library are less than impressive. The SyntaxError thrown here reports that the parser unexpectedly encountered a token 14; 14 is the index of the `[` token. If we specify that the grammar should be extended with the list extension in `grammars/java-list.grm`, the parser can now parse the file:

```
$ ./extend grammars/java.grm grammars/java-list.grm Test.java
Reading grammars/java.grm...
Reading grammars/java-list.grm...
Parsing Test.java...
--- Done ---
Unambiguous: 182
  Tentative: 27
 Reparseing: 23
```

If parsing succeeds, the program outputs the number of instructions that were executed during the parsing of the input.

A.2. Locally extensible statement parser

The second example. `stmt`, can be used to experiment with locally defined syntax extensions. This example can be run using either the `stmt` shell script that is also in the root directory of the CD-ROM, or executing `jar/stmt.jar`. The program only accepts a single argument, which is the input file.

The program parses input written in yet another variant of the simple statement syntax that has been a running example throughout this thesis, extended with a construct for defining local grammar extensions:

```
<expr> -> $ident
        | $num
;

<stmt> -> "if" <expr> "then" <stmt> "else" <stmt>
        | "while" <expr> "do" <stmt>
        | "{" <stmt>* "}"
        | <expr> ";"
        | "macro" [[ <grammar> ]] "in" <stmt> [[ . ]]
;
```

The first part of a **macro** statement must be the definition of a grammar, which can be used to specify a grammar extension that will be enabled within the body of the **macro** statement. For instance, this program is illegal, because the statement grammar does not define an **unless** construct:

```
if a then {
  unless x do {
    1; 2;
  }
} else {
  3; 4;
}
```

and trying to parse this program causes an error:

```
$ ./stmt stmts/unless-error.grm
Exception in thread "main" com.tedir.SyntaxError: 1
        at com.tedir.Interpreter.interpret(Interpreter.java:110)
        at com.tedir.Parser.parse(Parser.java:27)
        at com.test.StmtParser.main(StmtParser.java:123)
```

Using the **macro** statement, the grammar can be extended with the **unless** locally:

```
if a then {
  macro {
    <stmt> -> "unless" <expr> "do" <stmt> ;
  } in {
    unless x do {
      1; 2;
    }
  }
} else {
  3; 4;
}
```

Now that the **unless** statement is defined, the program can be parsed:

```
$ ./stmt stmts/unless-macro.grm
Done parsing.
```

This example demonstrates that the library allows parsers to be constructed on the fly, based on part of the input that is currently being parsed, and that the extended parsers can be enabled locally.

B. Reflective Production Actions

This appendix gives the implementation of the reflection-based interface to specifying a grammar.

The code that constructs the delta from an annotated class is shown in figure B.1. It iterates through all the methods of the specified class and for each method annotated with a production, it parses the production, tags it with the corresponding Method and adds it to the delta.

```
public static final Delta makeDelta(Class klass) {
    Delta delta = new Delta();
    for (Method method : klass.getMethods()) {
        Prod annot = method.getAnnotation(Prod.class);
        if (annot != null) {
            String prodString = annot.value();
            Production prod;
            try { prod = ProductionParser.parseProd(prodString); }
            catch (Exception e) { throw new RuntimeException(e); }
            delta.addTagProd(prod.name, method, prod.body);
        }
    }
    return delta;
}
```

Figure B.1.: Method for generating a delta from an annotated class

The declaration of the @Prod annotation is shown in figure B.2. Prod is declared to contain a single String and be retained at runtime.

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Prod {
    String value();
}
```

Figure B.2.: Annotation type for specifying productions

```
public static final class MethodHandler implements ParseEventHandler {
    private final Object obj;
    public MethodHandler(Object obj) {
        this.obj = obj;
    }
    public Object parseEvent(Object nonterm, Object tag, Object data) {
        Method method = (Method) tag;
        try {
            if (data instanceof Object[]) return method.invoke(obj, (Object[]) data);
            else return method.invoke(obj, new Object[] {data});
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Figure B.3.: Parse event handler for executing methods reflectively

Finally, the parse event handler shown in figure B.3 invokes the method corresponding to the parsed production on the action object, passing the value of the parsed expression as argument.

C. Compilation Algorithm Details

This appendix gives the details of the compilation algorithm that were not given in-chapter 5 . The pseudocode of the algorithm is given here without any description or comment; chapter 5 can be used as a reference for a more intuitive description of the algorithm. The algorithm is split into seven sub-algorithms, corresponding to the seven phases of the compilation algorithm. The parts of the algorithm that were fully described in chapter 5, like the algorithm for generating bytecode or calculating the first set functions, are not shown here.

The algorithm relies heavily on two kinds of data structures: layered maps and reversible maps. Layered maps were described in section 5.2, and are used to store properties of the nonterminals in a way that allows structure sharing without changing the grammars being extended. Reversible maps are used for tracking dependencies. They allow dependencies of the form "the key x depends on the keys $\{y_1, y_2, \ldots, y_n\}$" to be stored, and can then answer queries like "on which keys does x depend" and reverse queries like "which keys depend on y". A reversible map can also calculate the transitive closure over both of these relations. There are different strategies for implementing a reversible map. The prototype implementation of the library uses layered maps of bit-vectors to allow structure sharing between the reversible maps used in the base grammar and the derived grammar.

Algorithm C.1 Delta application

```
for $n \in$ *directly changed nonterminals* do
    let $c$ be the set of nonterminals called from $n$
    record the dependency $n \leftarrow c$ in *nonterminal calls*
```

Algorithm C.2 Nullability analysis

```
**for** $n \in$ *directly changed nonterminals* **do**
    **let** $f$ **be** the nullability function calculated for $n$
    map $n$ to $f$ in the *nullability functions*
    **let** $s$ **be** the set of free variables in $f$
    record the dependency $n \leftarrow s$ in *nullability dependencies*
    **if** the nullability function of $n$ has changed **then**
        add $n$ to *nullability function changed*
**let** *recalc* **be** the transitive closure of *nullability function changed*
    over *nullability dependencies*
**for** $n \in$ *recalc* **do**
    **let** $b$ **be** the nullability of $n$
    map $n$ to $b$ in *nullability*
    **if** the nullability of $n$ has changed **then**
        add $n$ to *nullability changed*
```

Algorithm C.3 First set analysis

```
**let** *recalc first set function* **be** *directly changed*
**for** $n \in$ *nullability changed* **do**
    **let** $d$ **be** the nonterminals depending on $n$ in *nullability first dependencies*
    add all members of $d$ to *recalc first set function*
**for** $n \in$ *recalc first set function* **do**
    **let** $f$ **be** the first set function calculated for $n$
    map $n$ to $f$ in *first functions*
    **let** $d$ **be** the nonterminals on whose nullability $f$ depends
    record the dependency $n \leftarrow d$ in *nullability first dependencies*
    **let** $s$ **be** the set of variables occurring in $f$
    record the dependency $n \leftarrow s$ in *first dependencies*
**let** *recalc* **be** the transitive closure of *first set function changed*
    over *first dependencies*
**for** $n \in$ *recalc* **do**
    **let** $s$ **be** the first set calculated for $n$
    map $n$ to $s$ in *first sets*
    **if** the first set of $n$ has changed **then**
        add $n$ to *first set changed*
```

Algorithm C.4 Local analysis

```
let recalc local be directly changed
for $n \in$ nullability changed $\cup$ first set changed do
    let $d$ be the nonterminals depending on $n$ in nonterm calls
    add all members of $d$ to recalc local
for $n \in$ recalc local do
    { NONTERM CALLS }
    let prev follow map be the previous follow map of $n$
    let reducible nonterms be {}
    for $m \in$ the set of nonterminals called from $n$ do
        let local follow be {}
        for $s \in$ the set of nodes following calls to $m$ do
            let $l$ be the local first set of $s$
            add $l$ to local follow
            let $r$ be the reducibility of $s$
            if $r$ then
                add $m$ to reducible nonterms
            map $m$ to local follow in local follow map
            if $m$ mapped to a different value in prev follow map then
                add $m$ to local follow changed
    record the dependency $n \leftarrow$ reducible nonterms in reducible nonterm calls
    for $d \in$ the symmetric difference between local follow map and prev follow map do
        add $d$ to local follow changed
    { CHOICE NODES }
    let reducible choices be {}
    let choice descriptors be the choice descriptor map of $n$
    for $c \in$ the set of choice nodes in the graph of $n$ do
        let choice descriptor be [follow: {}, redecible: {}]
        for $k \in$ the set of nodes following $c$ do
            let $f$ be the local first set of $k$
            map $k$ to $f$ in the choice descriptor.follow
            let $r$ be the reducibility of $k$
            map $k$ to $r$ in choice descriptor.reducible
            if $r$ then
                add $c$ to reducible choices
        map $c$ to choice descriptor in choice descriptors
        if $c$ has changed in choice descriptors then
            add $(n, c)$ to changed choices
    map $n$ to choice descriptors in choice descriptor map
    map $n$ to reducible choices in reducible choice map
```

Algorithm C.5 Immediate follow set analysis

```
for $n \in$ *directly changed nonterminals* $\cup$ *local follow changed* do
    let *immediate follow set* be {EOF}
    for $m \in$ the nonterminals depending on $n$ in *nonterm calls* do
        let $l$ be the mapping for $m$ om *local follow maps*
        let $f$ be the mapping for $n$ in $l$
        add $f$ to *immediate follow set*
    map $n$ to *immediate follow set* in *immediate follow sets*
    if $n$ has changed in *immediate follow sets* then
        add $n$ to *immediate follow set changed*
```

Algorithm C.6 Follow set analysis

```
let *recalc* be *directly changed*
let *changed root* be the union over *immediate follow changed*
    of *reducible nonterminal calls changed*
let *closure* be the transitive closure of *changed root* over *reducible calls*
add all members of *closure* to *recalc*
let *local follow map* be {}
for $n \in$ *recalc* do
    map $n$ to its immediate follow set in *local follow map*
let *recalced* be *recalc*
while *recalc* is non-empty do
    let *current recalc* be *recalc*
    *recalc* $\leftarrow$ {}
    for $n \in$ *current recalc* do
        let $o$ be $n$'s value in *local follow map*
        let $c$ be {}
        for $m \in$ the set of nonterminals in which $n$ occurs reducible do
            if $m$ has a mapping in *local follow map* then
                add the furrent follow set of $m$ to $c$
            else
                add $m$'s follow set to $c$
        if $o$ and $c$ are different then
            add all nonterminals occurring reducible in $n$ to *recalc*
for $n \in$ *recalced* do
    map $n$ to the current follow set of $n$ in *follow*
    if $n$ has changed in *follow* then
        add $n$ to *follow changed*
```

Algorithm C.7 Choice table update

```
for $n \in$ *follow changed* do
    for $c \in$ the set of *reducible choices* do
        recalculate the choice $c$ in $n$
for $(n, c) \in$ *changed choiced* do
    recalculate the $c$ choice in $n$
```

Bibliography

Appel, A. W. (1998), *Modern Compiler Implementation in C*, Cambridge University Press.

Brabrand, C., Schwartzbach, M. I. & Vanggaard, M. (2003), The metafront system: Extensible parsing and transformation, *in* B. Bryant & J. Saraiva, eds, 'Electronic Notes in Theoretical Computer Science', Vol. 82, Elsevier.

Bracha, G. & Cook, W. (1990), Mixin-based inheritance, *in* N. Meyrowitz, ed., 'Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming', ACM Press, Ottawa, Canada, pp. 303–311.

Carmi, A. (2002), 'Adapser: an lalr(1) adaptive parser'.

Cash, J. (1975), *Cash: the Autobiography*, HarperCollins.

Conway, D. (2003), *Parse::RecDescent documentation.*
URL: *http://search.cpan.org/dist/Parse-RecDescent/lib/Parse/RecDescent.pod*

Dijkstra, A. & Swierstra, D. S. (2001), Lazy functional parser combinators in java, Technical Report UU-CS-2001-18, Universiteit Utrecht.

Donnelly, C. & Stallman, R. (2002), *Bison: The YACC-compatible Parser Generator.*
URL: *http://www.gnu.org/software/bison/manual/ps/bison.ps.gz*

Gosling, J., Joy, B. & Steele, G. (1996), *The Java Language Specification, First Edition*, Addison-Wesley.
URL: *http://java.sun.com/docs/books/jls/*

Gosling, J., Joy, B., Steele, G. & Bracha, G. (2002), *The Java Language Specification, Second Edition*, Addison-Wesley.
URL: *http://java.sun.com/docs/books/jls/*

Grune, D. & Jacobs, C. J. H. (1990), *Parsing techniques a practical guide*, Ellis Horwood Limited.

Horspool, R. N. (1990), 'Incremental generation of LR parsers', *Computer Languages* **15**(4), 205–223.
URL: *citeseer.ist.psu.edu/horspool89incremental.html*

Hutton, G. (1992), 'Higher-order functions for parsing', *Journal of Functional Programming* **2**(3), 323–343.
URL: *citeseer.ist.psu.edu/hutton93higherorder.html*

ISO/IEC (1996), 'Extended bnf – generic base standard'.

Knuth, D. E. (1964), 'Backus normal form vs. backus naur form', *Communications of the ACM* **7**, 735–736.

Knuth, D. E. (1965), 'On the translation of languages from left to right', *Information and Control* **8**(12), 607–639.

Koskimies, K. & Vihavainen, J. (1992), Incremental parser construction with metaobjects, Technical Report A-1992-5, University of Tampere.

Kozen, D. C. (1997), *Automata and Computability*, Springer.

Leijen, D. & Meijer, E. (2001), 'Parsec: A practical parser library', *Electronic Notes in Theoretical Computer Science* **41**(1).

Mjølner (1996), *The Mjølner BETA System Metaprogramming System – Reference Manual.*

Odersky, M. & Wadler, P. (1997), Pizza into Java: Translating theory into practice, *in* 'Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France', ACM Press, New York (NY), USA, pp. 146–159.

Østerbye, K. (2002), Refill – a generative java dialect, *in* 'Proceedings of the ECCOP Workshop on Generative Programming, Malaga, Spain, 2002'.

Parr, T. (2004), *ANTLR Reference Manual.*
URL: *http://www.antlr.org/doc/index.html*

Tomita, M. (1986), *Efficient parsing for natural language*, Kluwer Academic Publishers.

van den Brand, M., Scheerder, J., Vinju, J. J. & Visser, E. (2002), Disambiguation filters for scannerless generalized LR parsers, *in* 'Computational Complexity', pp. 143–158.

Vidart, J. (1974), Extensions Syntactiques dans une Contexte LL(1), PhD thesis, University of Grenoble.

Waldhoff, R. (2003), 'Wanted: Modular/extensible parser generator'.
URL: *http://radio.weblogs.com/0122027/2003/07/07.html*