Syntax Design Considerations

Dick Grune
dick@dickgrune.com

October 26, 2014; DRAFT

For open issues, see Section 10.

1 Introduction

Since people can read and understand abstract algorithms, it can be assumed that their Platonic phrase-structure grammar is unambiguous. For a computer to read and parse an abstract algorithm as a program, however, a context-free grammar is required that is preferably deterministic but at least unambiguous.

CF grammars cannot express the long-distance relationships (context conditions) required to keep the produced programs unambiguous and meaningful.

The standard approach is to initially ignore the context conditions and modify the CF grammar so it becomes deterministic (LALR(1) or LL(1)). A postscan is then required to undo the modifications and exercise the context conditions to establish the meaning of the program. Both creating the deterministic grammar and constructing the postscan is difficult.

Since we have more freedom of operation than a compiler writer working from a fixed manual –but less than a language designer who designs his own language– we can try to solve the ambiguity problem in a prescan, reducing the task of the postscan to handling the context conditions only.

2 A grammar for Teckel

Abstract algorithms consist of a mix of expressions and syntactic constructs. The expressions are fairly standard, using operators like `\land` (logical and), `\union` (set union), etc., which are well represented in LaTeX, which fixes their representations. The syntactic structures are also fairly standard, featuring function and operator declarations, if- and while-statements, etc., but their representations are controversial, witness the large number of algorithm typesetting packages available under LaTeX. Many of these have idiosyncratic syntaxes, which Teckel would hate to impose on its users. The solution lies in defining simple LaTeX commands for the syntactic constructs, which

* are easy to parse for Teckel;
* can be redefined easily to achieve most print formats.

Example: the Teckel lines

```
\Tdefine{a, a \in V}{G(1)}
\Tassign[\cup]{V}{\{b\}}
\Tfunction{\xi}{a, b \in \{0..9\}}{a \times b \mod 10}
```

could print, for example, as

$a, a \in V \equiv G(1)$
$V \cup = \{b\}$
$\xi(a, b \in \{0..9\}) \equiv (a \times b \mod 10)$

or, for example, as

let $a, a \in V$ **be** $G(1)$
set V **to** $V \cup \{b\}$
function $\xi(a, b \in \{0..9\})$ **is** $a \times b \mod 10$

As expected the syntax construct part of the grammar was immediately deterministic (LALR(1)), but the expression grammar was ambiguous and had around 60 LALR(1) conflicts. Most of the problems were due to the impossibility to distinguish between operators, functions and identifiers.

2.1 Fundamental ambiguities

At least two features set Teckel syntactically apart from the average programming language: users can define their own operators, prefix, infix and postfix; and multiplication can and concatenation must be done with the Invisible Operator. The combination of these features causes at least three ambiguities:

1. Does '$\xi\ \phi \equiv ...$' define a prefix operator ξ with formal parameter ϕ, or a postfix operator ϕ with formal parameter ξ?
2. In the expression '$a\ \xi\ \phi\ b$', is ξ a postfix operator and ϕ an infix operator, or is ξ the infix operator and ϕ a prefix operator?
3. Is the expression '$P(a, b)$' a call of function P or the product of P and the tuple (a, b)?

If we want to have an unambiguous grammar of Teckel we need to get rid of these ambiguities before parsing starts.

Problem 1 is an essential ambiguity, and requires an arbitrary decision. Human readers would probably conclude that $\xi\ a$ or $\xi\ \alpha$ defines a prefix ξ and that $a\ \phi$ defines a postfix ϕ, since these are the more marked.

This intuition is formalized by decreeing that the formal operand to a postfix operator has to be a single lower-case letter (or a packed formal, where "packed" means "inside parentheses"), and that unsubscripted postfix operators cannot be single lower-case letters. So $\xi\ a$ and $\xi\ \alpha$ define a prefix ξ and $a\ \phi$ defines a postfix operator ϕ.

Problems 2 and 3 can be solved by a prescan which determines which names are prefix/infix, postfix and functions. It is not necessary to distinguish prefix and infix (or the + and −, which are both prefix and infix, would have been in trouble for centuries).

We do not identify constant identifiers, the fourth possibility in a `\Tdefine`, since they may also be used as formals; people write things like: "declare $F(a, b)$, and now try this for $a = 1$ and $b = 0$". (Nor is there any need to.)

There is another fundamental ambiguity in the notation itself (Section 10.3): The expression 1..n clearly means 1 2 ... $n-1$ n, and $X_1..X_n$ clearly means X_1 X_2 ... X_{n-1} X_n. But why doesn't it mean X_1 X_1+1 ... X_n-1 X_n?

????˚ ˚ introduce GLR somewhere here

2.2 Problems with the GLR grammar

˚ ˚ under construction

It is undecidable if a CF grammar is ambiguous, so the only way to make certain our grammar is unambiguous is by inspection and by extensive testing.

The CF grammar constructed on the basis of the tokens as supplied by the prescan was tested with complicated Teckel programs and Bas Basten's ambiguity detection program *AmbiDexter*, and was found to be ambiguous in the following places:

- The parameter list in

  ```
  \Tfunction{F_k}{a, b, a \in A}{\Treturn{1}}
  ```

  can be parsed both as `a, b(, a \in A)` and as `a(, b, a \in A)`, where the parenthesized segment is the `type_definition_list_addition`. Since parameter list must be checked for duplicate definitions anyway, we can just as well analyse the whole parameter list in the postscan.˚ ˚ postscan

- The expression `|U|V|W|` is ambiguous. It can be parsed several ways, depending on where we insert the invisible operator (multiplication or concatenation) or the equally invisible operator applicator. Two feasible interpretations would be `|U|.V.|W|` with `V` a numeric and the dots representing the invisible multiplication operator; and `| U|V|.W |` with `U` a monadic operator yielding a set from an integer and `W` a set; the whole expression would then yield the size of the product of the two sets.

  The problem with the `|` is that it is symmetrical: the open `|` is the same as the close `|`. But there is a difference: the open-`|` is always preceded by `{`, `(`, or an operator, and the close-`|` is always followed by an operator, `)`, or `}`. So all would be well if the invisible operator did not exist.

  This suggests modifying the grammar so the invisible operator cannot occur on the top level in the argument to the set size operator. It is actually pretty simple to do this, as follows: The first sensible non-terminal in the expression hierarchy to be an argument to the size operator is `arith_expression`; all higher ones yield Booleans. The invisible operator appears only five levels lower, in `factor`, so only the rules in between have to be duplicated. This leaves only the interpretation `(| U |).V.(| W |)`, which is suitable.

  However, the invisible operator is also used for string concatenation, and we want to be able to write `|aSa|` for the length of the string `aSa`. So the solution based on the elimination of the invisible operator is unacceptable.

  The next suggestion that comes to mind is forbidding size operators on the top level inside size operators. That is more difficult to do. One way is by duplicating the whole `expression` hierarchy. Another is by having the

prescan mark the |s alternatingly as open-| and close-|, but that is not easy either because of the use of | as a filter separator in generated sets. Defining the top level as the regions between commas in a generated set with the exclusion of enclosed sections, we may get several regions with an even number of |s, but there will be one region with an odd number of them. Consider the expression

$$\{(a, b), a \in |V|\rho|W|, b \in \tau|U| \mid \kappa a < |A|, b \neq |B|\}$$

which such a prescan would reduce to

$$\{, ||||, |||||, ||\}$$

for analysis. Now there seems to be no easy way to find out which of the five |s in the middle is the set generator separator.

Also it is questionable if suppressing set size operators within set size operators is a good idea aside from its implementability. The expression $|\tau|U||$ is perfectly OK for an operator τ which works on an integer and yields a set, and is unambiguous.

So perhaps the wisest thing to do is to leave the ambiguity in and give an error message in the extremely unlikely event that somebody stumbles into it.

The grammar seems ambiguous but isn't, in the following places:

- Multiplication and operator application have forms in common (= are ambiguous) due to the identity of the invisible operator and the operator applicator. `F a` is both the multiplication of `a` by `F` and the application of the prefix operator `F` to `a`. Solution: the prescan has already marked `F` as a prefix operator if it is one. The form `F a` may still be the product of two numericals or the concatenation of two atoms, but that depends on their types, which is a matter for the postscan.
- Enumerated sets and generated sets have forms in common due to `\in` occurring in both. The enumerated set

  ```
  \{ b, b \in G \}
  ```

  which is a set containing two Booleans, is identical to generated set

  ```
  \{ b, b \in G \}
  ```

  which is the not necessarily Boolean set `\{b, b \in G | \Ttrue \}`. A similar ambiguity occurs with a range type definition:

  ```
  \{ 0 < i < 10 \}
  ```

  is both a set containing one Boolean and the set of the number 1 to 9.

  However, a set of Booleans is a rather limited thing; its only values are $\{\}$, $\{\textbf{false}\}$, $\{\textbf{true}\}$, and $\{\textbf{true}, \textbf{false}\}$, however many members the enumerated or generated set may have had. So we disallow Boolean expressions

as the top level expressions in sets; this is easy to do since Boolean expressions are produced high in the expression tree, and it makes the ambiguity go away. Sets of Booleans can still be made, by putting parentheses around the expressions: `\{ b, (b \in G) \}`.

So, once the prescan is in place, the CF grammar (`syntax.g`) is almost unambiguous, and safe. The prescan is described in `PrescanDesignConsiderations.pdf`.

3 Expressions with Ellipses

There are three general forms of expressions with ...:

1. $A_1\ op_1\ ...\ op_{n-1}A_n$ (§3.1)
2. $A_1\ ...\ A_n$ (§3.2)
3. $E\ ...\ E$ (§3.3)

where A_1 and A_n are indexed elements and the Es are expressions.

Each of these forms has variants, which has made it difficult to find a simple syntax for these forms.

3.1 Operator shorthand

The operator shorthand

$$A_1\ op_1\ ...\ op_{n-1}A_n$$

is an expression and yields a single value. Examples are $\texttt{x}_1\texttt{+...+x}_5$ and $\texttt{b}_1\wedge\texttt{...}\wedge\texttt{b}_5$. The operators op_1 and op_n must be the same; they must by symmetrical and have types $(\alpha,\alpha)\Rightarrow$`Bool` or $(\alpha,\alpha)\Rightarrow\alpha$: forms like $\texttt{x}_1\texttt{-...-x}_5$ are not allowed. These restrictions are implemented in the postscan.° ° postscan

There is a modified version of this construct:

$$A_1\ op_1\ A_{1+s}\ op_2\ ...\ op_{n-1}A_n$$

which allows the user to specify the step s.

The constructs are implemented by accepting ... as a `primary_1` operand, which is clearly a kludge. It is very likely that the postscan° will have to restructure this form. It also has the disadvantage that it allows forms like $x_1 \times \cdots \times x_n + y$, or $a \wedge b_1 \wedge \cdots \wedge b_n \wedge c$, which we may not want. ° postscan

3.2 Sequence shorthand

The sequence shorthand

$$A_1\ ...\ A_n$$

can be (part of) an expression or of formal parameter.

3.2.1 Sequence expressions

The most general form of an expression using a sequence expression is someting like

$$ab^ncF(1,\xi_1)F(3,\xi_3)...F(n,\xi_n)\mathcal{Q}$$

in short, the concatenation of `primary_2`s. As the example shows, the range of the repetition and the arguments affected by the range can be complicated beyond any limit, and there is no way to express this in a CF grammar. The problem is again solved/postponed by accepting ... as a `primary_1` operand. The rest of the analysis has to be done by the postscan.

A disadvantage is that it allows forms like $\xi_1...\xi_i...\xi_n$, which will have to be rejected by the postscan since they have no meaning.

3.2.2 Sequence formals

As (part of) a formal parameter

$$A_1 \ ... \ A_n$$

matches a sequence of values. The matching yields two arguments: an integer n and an array A. The values are accessible as the array elements A_1 ... A_n. Unlike sequence expressions the step size is always 1 and cannot be specified.

Some algorithms "zip" two or more sequences: $x_0B_1x_1B_2x_2\cdots B_mx_m$ is a real-world example [1]. Since the step size cannot (and need not) be specified, this must be expressed as $x_0B_1\cdots B_mx_m$.

We want to be able to embed this in a "normal" sequence, for example $\alpha\beta\gamma_1\cdots\gamma_n\delta$ or $A \rightarrow x_0B_1\cdots B_mx_m$. Matching this to the actual parameter yields 4 arguments: an atom A, an integer m, and two arrays of atoms, $x[0..m]$ and $B[1..m]$.

Again, it seems next to impossible to express all these varieties in a CF grammar. So we settle for

```
formal_element_range:
    formal_element_sequence from_to_token formal_element_sequence
```

This, however, makes two consecutive `formal_element_range`s ambiguous: in

```
a b \xi_1 ... \xi_n \xi_1 a_1 ... \xi_n a_n \mathcal{Q}
```

where does the second `\xi_1` go? To allow the user to disambiguate, the `formal_element_range` may be parenthesized.

In the end the postscan˚ will have to sort all this out. ˚ postscan

3.3 Integer sets and index sequences

If enclosed in { and }, the form

$$E \ ... \ E$$

[1] This sequence is required to match the right-hand side of a syntax rule, with $x_i \in \Sigma^*$ and $B_i \in N$.

results in a set of integers: $\{1..5\}$ denotes $\{1, 2, 3, 4, 5\}$; it is actually shorthand for $\{1, .., 5\}$. This works for integers only; there is no $\{0.5..4.5\}$.

The form E ... E can also occur in an index position: $A_{1..5}$ or $A[1..5]$, and is then distributed over the A, resulting in the sequence $A_1A_2A_3A_4A_5$.

Syntactically this is again subsumed by accepting ... as a `primary_1` operand. The semantics must be retrieved by the postscan, based on the type of the operands, and its position inside { and } or in an index.

Accepting ... as a `primary_1` operand in expressions solves a number of syntactic problems. The price is that the postscan has to weed out abuse of the ... as for example in $x + ...$˚ ˚ postscan

4 User Operators

4.1 The Precedence of User Operators

Built-in prefix and postfix operators have only one precedence. That of prefix is just lower than that of exponentiation, so $-a^2$ parses as $-(a^2)$ rather than $(-a)^2$. Postfix is just lower than subscripting, to avoid $a![k]$. There is no reason to assign user prefix and postfix different precedences.

Built-in infix operators, however, occupy several precedence levels. Some of them are of the kind the user cannot or will not want to supply, for example $\wedge$. The rest falls into two groups: relational operators ($=$, $\leq$, etc.) and value operators ($\cap$, $+$, $\times$, etc.).

The relational operators all have the same precedence; they are used in comparisons ($a < b$), in comparison chains ($a < b = c \leq d$), in type definitions ($0 \leq i < 10$), and in generators ($0 \leq i < 10$). Note that $=$, $>$, and $\geq$ are allowed in comparisons and in comparison chains, but are disallowed in type definitions and in generators.

A similar restriction applies to user operators. They cannot be used in type definitions or generators since there is no way to generate the xs in a form like $p \prec x \preceq q$, where $\prec$ and $\preceq$ are user-defined relational operators˚. (unless the user supplies a successor operation on this data type, but that is unusual in abstract algorithms). They can still be used in chained comparisons without problems: $0 \doteq i \prec j \ll 10$ is perfectly manageable, and is equivalent to $0 \doteq i \wedge i \prec j \wedge j \ll 10$. ˚ postscan test

The value operators basically have two precedence levels, that of $\{+, -, \cup, \setminus\}$, and that of $\{\times, \cap\}$; and then there is the invisible operator. The others, $/$, exponentiation, and subscription, are marginal, and are handled by specific syntax constructions.

The invisible operator optically binds its operands more tightly than any other operator, so probably it should have the highest precedence. This is the more fortunate, since then $a\ b\ c\ \xi\ d\ e\ f$ parses as $(a\ b\ c)\ \xi\ (d\ e\ f)$, regardless of what precedence we give to the user infix operator. This is important since abstract algorithms often employ sequences, so user operators will want to operate on them.

It is also likely that user operators will quite often manipulate sets (and much less frequently numbers, since the operators for that are already available). These sets can also be manipulated by the built-in operators $\cup$ and $\cap$. In '$a\xi b \cap c$'

the user operator seems to bind tighter, because it draws more attention; so we prefer the parsing '$(a\xi b) \cap c$' over '$a\xi(b \cap c)$'.

The set operators and the arithmetic operators cannot work on each others results, to their relative precedences do not matter. Still it seems reasonable to have $\{\cup, +\}$ together, and $\{\cap, \times\}$.

Since $\times$ and the invisible operator both represent multiplication, one would like to give them the same precedence. But they behave differently syntactically: the right operand of $\times$ can be prefixed (as in $a \times -b$), but that of the invisible operator cannot, or $a - b$ would also parse as $a\ (-b)$ and be ambiguous. How about $2x \times 3y \times 4z$? Is it $(((2x) \times 3)y \times 4)z$ or is it $(2x) \times (3y) \times (4z)$? Does it matter? In fact it does not: $\times$ is defined only for numerals, and then the forms are equivalent, since Teckel operates with unbounded precision rationals.° How about something like $2xy^2 \times F(\zeta)G(\psi)$? It should certainly be permitted.

° ????

So we have

$$\{\cup, \setminus\} \prec \cap \prec \xi \prec \varepsilon$$
$$\{+, -\} \prec \times \preceq \varepsilon$$

where ε is the invisible operator.

This leaves three possibilities for the precedence relation between the user infix operator and $\times$: $\xi \asymp \times$, $\xi \prec \times$, and $\times \prec \xi$. If we assign ξ the same precedence as $\times$ we get

$$\begin{array}{lll} a\ \xi\ b \times c & \text{is parsed as} & (a\ \xi\ b) \times c \\ a \times b\ \xi\ c & \text{is parsed as} & (a \times b)\ \xi\ c \end{array}$$

which looks inconsistent.

So we can assign ξ a lower precedence than $\times$, which places it between $+$ and $\times$, or we can give it a higher precedence than $\times$, which places it between $\times$ and the invisible operator. If we consider the six possible permutations of $+$, $\times$ and a user operator,

$$\begin{array}{l} a + b \times c\ \xi\ d \\ a + b\ \xi\ c \times d \\ a \times b + c\ \xi\ d \\ a \times b\ \xi\ c + d \\ a\ \xi\ b + c \times d \\ a\ \xi\ b \times c + d \end{array}$$

the impression is that the user operator, being more unusual, binds tighter than the $\times$. This gives us the overall precedence

$$\{\cup, \setminus, +, -\} \prec \cap \prec \times \prec \xi \prec \varepsilon$$

and the six expressions parse as follows:

$$\begin{array}{l} a + (b \times (c\ \xi\ d)) \\ a + ((b\ \xi\ c) \times d) \\ (a \times b) + (c\ \xi\ d) \\ (a \times (b\ \xi\ c)) + d \\ (a\ \xi\ b) + (c \times d) \\ ((a\ \xi\ b) \times c) + d \end{array}$$

There is, however, another option:° give $\times$ and the user infix operator the same precedence but forbid them to be used together. The precedence level can be occupied by only one of them in a given `factor`. This gives

° decide, in due time

$$\{\cup, \setminus, +, -\} \prec \cap \prec \begin{matrix}\times \\ \xi\end{matrix} \prec \varepsilon$$

and only the expressions

$$a \times b + c \;\xi\; d$$
$$a \;\xi\; b + c \times d$$

are legal; they indeed parse in the obvious way. The other four would need parentheses:

$$a + b \times (c \;\xi\; d)$$
$$a + (b \;\xi\; c) \times d$$
$$a \times (b \;\xi\; c) + d$$
$$(a \;\xi\; b) \times c + d$$

This restriction can be enforced in the grammar or in a postscan. Putting it in the grammar is less work, but the postscan might give better error messages.

4.2 User Operators in Chained Comparisons

We want to allow *chained comparisons* like $0 < i = j \leq 10$. Do we also want to allow $0 < i = j \geq k \geq 10$? If not, how do we prevent it?

We cannot just allow the user-defined operators in chained comparisons, for the relational operators in them have a lower precedence than the user-defined ones. This is because we want

$$a \;\xi\; b = c \;\phi\; d$$

to be parsed as

$$(a \;\xi\; b) = (c \;\phi\; d)$$

So we need two levels of user-defined operators: relational ones and value ones. But how do we distinguish them? The solution comes from Table 13 in "The Comprehensive LaTeX Symbol List", which explicitly defines a large set of relational operators. Some are already defined by Teckel ($\geq$, $\in$, $\leq$, $\ni$, $\neq$, $\subset$, $\subseteq$, $\supset$, $\supseteq$). (Note: $\in$ and $\ni$ are not relational operators in that their both operands do not have the same type; so they cannot take part in chained comparisons.°) The rest can be used by users as relational operators, with the precedence of the built-in relational operators. This way

° $\in$ and $\ni$ are not relational operators

$$a \;\xi\; b \;\dagger\; c \;\prec\; d \;\preceq\; e \;\oplus\; f$$

is parsed as

$$((a \;\xi\; b) \;\dagger\; c) \;\prec\; d \;\preceq\; (e \;\oplus\; f)$$

a parsing on which the interpretation $((a \;\xi\; b) \;\dagger\; c) \;\prec\; d \wedge d \;\preceq\; (e \;\oplus\; f)$ can be based, rather than as

$$((((a \;\xi\; b) \;\dagger\; c) \;\prec\; d) \;\preceq\; e) \;\oplus\; f$$

which cannot be the basis of a similar interpretation.

Since we have no means to know what such user-defined relational operators mean, we cannot prevent chained comparisons like $p \prec q \succeq r$ (unless we split them in three groups, leq, eq and geq, based on the shapes of the operators, which seems quite over the top). So there is little point in trying to suppress such monstrous but still meaningful expressions as $0 < i = j \geq k \geq 10$.

5 Parse Tree and Program Generation

The Teckel program is strictly a binary tree, so a gap must be bridged between the varying number of children in the Teckel grammar and the 2-child policy of the run-time system. Traditionally one would have data structures for the main components of the grammar, and then use code generation to convert these to the "object code" required. Such a conversion process is required anyway in the traditional set-up, since for parsing purposes the grammar will have been made deterministic (LL(1), LALR(1), etc.) and these distortions must be undone.

GLR parsing, however, allows the use of the publication grammar almost directly, and one argument for code generation, the undoing of the distortions, disappears. If the grammar could be written so that it easily leads to a binary parse tree a second argument for it (plus a lot of data structures) would disappear as well, leaving only the need for checking the context conditions.

5.1 AND-OR-binary Grammars

A programming language grammar consists of named (but not always uniquely named) alternatives with a variable number of members, many of which end up as child nodes in the program. Almost all Teckel nodes consist of a node type and two pointers to child nodes. The node type tells the teckel which actions to perform when it stops at the node and the semantics of a program node derives from the nature of the named alternative in the grammar. So a connection must be made between the node type and the alternative. The natural way to do so is through the name of the alternative. This name is, however, not always unique. The rule

```
expr:
  term |
  expr '+' term |
  expr '-' term ;
```

is shorthand for

```
expr:  term ;
expr:  expr '+' term ;
expr:  expr '-' term ;
```

so all the alternatives `term`, `expr '+' term`, and `expr '-' term` have the same name `expr`.

The solution is to turn some of the alternatives into named rules:

```
expr:
  term |
```

```
  sum |
  difference
sum:
  expr '+' term
difference:
  expr '-' term
```

Now `sum` and `difference` will yield nodes with node types `sum` and `difference`. Assuming that `term` yields a node with node type `term`, and employing Bison's property to pass on the first argument in the absence of semantics code, `expr` now yields a node with the proper node type, exactly the way we want it.

This leads in a natural way to AND-OR grammars.° Each composite grammatical structure gets a name in an AND-rule, and unions of these structures are combined automatically in OR-rules. ° litref?

An AND-rule may in principle contain an arbitrary number of members but must in Teckel correspond to one binary node with its name as node type. So the grammar we need is an AND-OR-binary (AOb) grammar, very similar in appearance to Chomsky Normal Form. This means that we have to impose restrictions on an AND-rule.

5.2 Generating code from an AOb grammar

A Bison alternative contains three kinds of items: tokens (e.g. `'+'`); named terminals (defined in `%token` lines); and named non-terminals. We observe that generally the tokens and terminals only serve to determine the syntactic structure, and we decide to ignore them in extracting information from Bison. If this is not desirable, the token or terminal can be put in an AND-rule, and so receive a name. (See Section 5.4).

If this leaves exactly two non-terminals in the alternative, they can be combined into a binary node, using code like

```
{$$ = create_node(R_X, $i, $j);}
```

for an AND-rule `X` with non-terminals at positions `i` and `j` in the alternative.

If there is only one, its value is passed on; for details see `g2y/rule.c`.

A check of the grammar shows that if there are three non-terminals in the alternative, it is usually a binary expression, for example

```
arith_expression term_operator term.
```

The most generic solution is to call a special routine:

```
{$$ = create_expression_tree(R_X, $i, $j, $k);}
```

which sorts it out. This feature has probably been abused° in the rules for ° !!!!

```
infix_definee
range_head
range_tail
compound_nominator
compound_factor
prefixed_primary_6
big_specification
```

```
postfixed_primary_3
range_source_head
range_source_tail
```

If there are more than three non-terminals in the alternative, the grammar is not AOb.

5.3 Processing the AOb grammar: g2y

The AOb grammar of Teckel, named `syntax.g` is prcessed by `g2y`, resulting in a Bison file `syntax.y` and a header file `syntax.h`. The grammar `syntax.g` has to be in AOb form; this property is checked by `g2y`. Running `syntax.y` through `bison` with the `%GLR` option creates a parser which will produce a binary tree for the program. The tree still requires context checking and perhaps a bit of tune-up.

5.4 Informative Terminals/Tokens

Since `g2y` considers tokens and terminals as syntactic entities devoid of semantics, some information may get lost. For example, a number is returned by the lexical analyzer as a token `number_constant`, with the result that it will be ignored in a grammar rule like

```
assignment:
  variable assignment_token number_constant
```

This can be remedied by putting the `number_constant` in a rule:

```
assignment:
  variable assignment_token number
number:
  number_constant
```

5.5 Informative Unit Rules

An AND-rule with more than one child automatically leads to a node in the parse tree, but a rule with only one child is taken for an OR-rule and is disappears. Sometimes it is necessary for a rule with only one child to result in a node. Prime example is the `program` rule / node:

```
program:
  unit_sequence
;

unit_sequence:
  unit
| unit unit_sequence
;
```

Since the first rule is a unit rule, `g2y` does not generate code for it, and there is no way to distinguish the program node from other `unit_sequence` nodes. Still this is required, since `reduce_unit_sequence()` signals the `Parent` when it is done, but the top node has no parent.

This is remedied by turning the unit rule into an AND-rule by adding the member `empty`:

```
program:
  empty unit_sequence
;
```

The non-terminal `empty` forces code for a call of `create_node()` to be created, which results in a node with node type `R_program` and an absent left child and a `unit_sequence` right child.

The same technique is used to force the creation of a node for the invisible operator:

```
invisible_operator:
  empty empty
;
```

which produces a node with node type `R_invisible_operator` and two absent children.

6 Type Definitions and Source Definitions

Although type definitions in formal operands

```
\Tdefine{ \oplus_k x, x \in \Sigma^* }{...}
                          ^^^^^^^^^^^^
```

and source definitions in generated sets and and repetition expressions

```
\{ ( x^2 + y^2 ) , x, y \in \mathbb{N} \}
                             ^^^^^^^^^^^^^^

\bigcup_{a \in A} (a a)
           ^^^^^
```

look very similar, there are important differences between the two, which is why they have different places in the syntax.

Broadly speaking, the difference is that type definitions just supply the type, and source definitions also supply the data.

Properties of the type definition:

- The expression in a type definition is evaluated only when that is required for parameter matching, as in

  ```
  \Tdefine{ \oplus n, n \in \{1..5\} }{...}
  ```

  which is a definition of the operator $\oplus$ that applies only to operands in the set $\{1..5\}$; but in

  ```
  \Tdefine{ \oplus n, n \in \mathbb{N} }{...}
  ```

  the set $\mathbb{N}$ is not evaluated, since it only brings in the type NUMERIC.

- The expression need not even be evaluatable; it could, for example, be $\mathbb{Q}$, the set of all rationals.

- The type definition may be left out if Teckel's type inference mechanism° can discover the type and membership of a specific set is not required.

° Presently non-existent

Properties of the source definition:

- The expression in a source definition, on the other hand, *is* evaluated, and this evaluation process drives the set generation:

```
\{ ( x^2 + y^2 ) , x, y \in \mathbb{N} \}
```

  This is expanded into

```
\{ ( x^2 + y^2 ) , x \in \mathbb{N}, y \in \mathbb{N} \}
```

  and the two $\mathbb{N}$ sets are evaluated independently, leading to the set $\{2, 5, 8, 10, 13, ...\}$.

- The expression must be evaluatable; $\mathbb{Q}$ is not allowed.
- Source definitions cannot be left out.

7 Error rules

Some syntax errors cause the GLR parser to signal an error far away from its original place; leaving out the `source_definition_list` in a `filtered_generated_set` is such an error:

```
\{ a^2 | a \% 2 = 0 \}
```

which produces the following very confusing error message:

```
unexpected token \} at position 20
```

The reason is that in the absence of a `source_definition_list` the `|` is taken as the left `|` of a `set_size`, preceded by the invisible multiplication operator. Since the right `|` does not show up, the `\}` is indeed unexpected.

The most convenient way to produce reasonable error messages in a GLR parser with an AOb grammar is through error rules. We allow the incorrect construct in an error rule. When the rule is reduced, the attempt to construct a node with an error rule type is caught in `create_node()`, where an appropriate error message can be given:

```
generator without source definition(s) at position 3
```

8 The Module Structure of the Parser

There are two problems with the module structure of the parser, both originating from the idiosyncratic nature of the `syntax.tab` module as produced by *bison*: the place of the auxiliary routines, and the type of the lexical token.

8.1 Placing the auxiliary routines

As usual the "boss" in the conversion process from program text to program tree is the parser, `yyparse()`. It reads its input from the lexical analyser through calls of `yylex()`, and writes its output through calls of `create_...()`. The `yyparse` and `yylex` interfaces differ from each other since they work in different directions. Also, they are determined by *bison* and they are not under our control. The `create_...` interface is under our control.

Such auxiliary routines are often/traditionally placed or included in the *bison* file, represented here by `syntax.g`. This leads, however, to a messy structure, and was considered unsatisfactory. Instead, three additional modules were defined, and placed as follows:

`main.c`	`parser.c`	`syntax.g`		`lex.c`	`codegen.c`
top level		`syntax.tab.c`	`syntax.tab.h`		
(Teckel)	(Teckel)	(bison)	(bison)	(Teckel)	(Teckel)
`U parse`	`D parse`				
	`U dot`			`D dot`	
	`U yyparse`	`D yyparse`	`not E yyparse*`		
			`D yytokentype*`		
		`U YYSTYPE`	`U YYSTYPE`	`D YYSTYPE`	
	`U yylval`	`D yylval`	`E yylval`	`U yylval*`	
		`U yylex`		`D yylex`	
		`U yyerror`		`D yyerror`	
		`U create_...`			`D creat_...`

Here `D` indicates the definition of an item; `E` its declaration (as `extern`; and `U` its use. This clearly shows one of the sources of the structuring problem: the `syntax.tab` module requires a left and right interface, one for the parser, which calls upon `syntax.tab`, and one for the lexical analyzer, which is called upon by it. This way a proper hierarchical structure is almost achieved.

There are problems left, however. Proper modules define entities in the code module, which they then declare in the corresponding header files, but `syntax.tab` does not conform to that, leading to the problems indicated by the asterisks in the table above.

- The module `syntax.tab` defines `yyparse()` in its code, but does not declare it in `syntax.tab.h`. This forces `parser.c` to import the name through an `extern` declaration. (It also defines `yychar`, `yylloc`, and `yynerrs` without declaring them in `syntax.tab.h`, but Teckel does not use these.)

- The module `syntax.tab` defines `yylval` in its code, and declares it in `syntax.tab.h`, but the lower module `lex` has to use it, since it has to set it. This is necessary because `yylex()` does not return the entire token, but only its node type. So `lex.c` is forced to include `syntax.tab.h` as an up-call.

- `Syntax.tab.h` defines `enum yytokentype`, the type of a token. Unsurprisingly this type is used by almost any module in the system, forcing almost all modules to include `syntax.tab.h`, whether they have something to do with syntax or not.

It is admitted that the technique of including the lexical analyser and the parser driver in the *bison* file has the advantage that the `yy...` names remain static and local. However, given their special form, there is little danger of confusion on the global level.

8.2 The type of `node_type`

The most important `struct` in Teckel is `struct node` and probably its most important field is `node_type`. Its values can come from three places: the *bison*-generated file `syntax.tab.h`, where their type is defined as `enum yytokentype`; the *g2y*-generated file `syntax_types.h`, where they are integers; and the hand-written file `node_types.h`, where they are again integers.

In practice this is not a problem because the C-compiler happily confuses `enum`s and `int`s, but it leads to sloppy structure and it would be good to have them all be of type `enum yytokentype`: *g2y* can generate the proper casts, and we can write them by hand in `node_types.h`. But a number of features of the programming environment conspire to make this a less than straightforward exercise.

- The node type is used in `struct node {...}` in `node.h`, and defined by *bison* in `syntax.tab.h`. So `syntax.tab.h` must be included before `node.h`.
- The *bison*-generated file `syntax.tab.c` needs both `node.h` and `syntax.tab.h`. The command `#include "node.h"` is placed in the first C-code section in `syntax.g`; the command `#include "syntax.tab.h"` is inserted by *bison after* the user C-code. Actually this makes sense, because `syntax.tab.h` uses the macro type `YYSTYPE`, the type of the *bison* stack items; this type has to be set by the user through a `#define` in the first C-code section.
- Putting another `#include "syntax.tab.h"` in front of the `#include "node.h"` in the C-code section throws compiler errors due to double definitions, since `syntax.tab.h` is not equipped with a `#ifndef SYNTAX_TAB_H` mechanism allowing it to be included multiple times.

So `syntax.tab.h` must both precede and follow `node.h`.

This is solved by realizing that it is not the *file* `syntax.tab.h` that needs to precede `node.h` but only its contents, and that the command `#include "syntax.tab.h"` in `syntax.tab.c` will include any file of that name. So we rename `syntax.tab.h` to `syntax_tab.h`; produce an empty file `syntax.tab.h`; and include `syntax_tab.h` anywhere `syntax.tab.h` should have been included.

This still leaves the macro `YYSTYPE` which is used by `syntax_tab.h`. To avoid the awkward situation that every file that needs `node.h` (which is pretty much every file in the system) is obliged to include the definition of `YYSTYPE` and `syntax_tab.h` before it, both have been inserted in `node.h`, together with `syntax_types.h` and `node_types.h`.

Including `node.h` in a C-file now provides all information about `struct node` and its `node_type`.

9 Resolved issues

1. Where *exactly* do we allow `\not`?

   Answer: In LaTeX `\not` is defined only in front of relational operators. Since these are a syntactic class in Teckel we do the same. Allowing it also in front of user-defined Boolean operators would be possible, but awkward, since in LaTeX it puts a left-aligned slash through the operator, and it would be of limited use.

2. The precedence of the "big" prefix operators is subtle. We want $\sum_{0<i<10} i^2$ to parse as $\sum_{0<i<10}(i^2)$ rather than as $(\sum_{0<i<10} i)^2$, so their precedence must be lower than that of the postfixed things. And we also want to avoid barbarisms like $\sum_{0<j<10} - \sum_{0<i<10} i \times j$, so it must be higher than that of the prefixed things.

3. Should we supply & for, and in addition, to $\wedge$?

   Answer: no, since we cannot use | for the corresponding $\vee$, and cannot come up with an acceptable alternative.

4. The backslash `\backslash` ($\backslash$) is indistinguishable from the set difference `\setminus` ($\setminus$). Forbid the `\backslash`?

   Answer: No, declare them to be the same.

5. Does $F^2(x)$ mean $F(F(x))$ or $(F(x))^2$?

   Answer: it shouldn't mean anything. See Wikipedia / Abuse of Notation / Trigonometric functions. But we do accept exponentiation of operators: $\rho^3 a$ means $\rho\rho\rho a$, unambiguously.

6. End-of-line/end-of-command conventions

   Usually commands in abstract algorithms come one to a line, as in

   $$V := \varepsilon$$
   $$A := \{a, V\}$$

   but when they are short enough, putting more than one on a line is not unusual:

   $$V := \varepsilon;\ A := \{a, V\}$$

   Since the "structured" commands cannot appear with more than one on a line, only a few commands have this problem. These are:

   ```
   \Tdefine
   \Tassign
   \Tcall
   \Toutput
   \Tcomment
   ```

   To accommodate both options in Teckel, we put commands one to a line by default, and revoke that default by having a semi-colon command, e.g. `\Tsc`. This requires some flag or redefinition of macros in the LaTeX package for the Teckel commands.

7. There are enumerated sets, e.g. $\{0, 1, n^2, K + 1\}$, in which the elements are enumerated; and there are generated, optionally filtered sets, e.g. $\{a \in A \cap B \mid a\%2 = 0\}$, or $\{a^2, a \in A\}$.

   Do we allow combinations, e.g. $\{0, 1, n \in X, K + 1\}$?

   Answer: No. This would be confusing ($\{p < q, n \in X\}$!), and is superfluous: one can write $(\{0, 1, K+1\} \cup \{n \in X\}$, or, simpler, $(\{0, 1, K+1\} \cup X)$.

   Do we allow sets of implicit tuples, e.g., $\{a \in A, b \in B\}$, to generate a set of pairs (a, b)?

   Answer: No. Again this would be confusing, and the correct notation is $\{(a, b), a \in A, b \in B\}$. As a compromise we allow $\{a \in A \mid a\%2 = 0\}$ for the singleton, but $\{a \in A\}$ is a set containing one Boolean.

   Is $\{a, a \in A \mid a\%2 = 0\}$ the same as $\{a \mid a \in A, a\%2 = 0\}$?

   Answer: No. The filter contains a list of Boolean expressions, so $a \in A$ is a type definition in the first form and a test in the second.

8. A sequence is a linear object; it seems useless to have or parameter-match multidimensional sequences.

   We want to keep multidimensional arrays, though. E.g. parse tables. So we need to match $A_{1..k,1..m}$ to identify A, and to set k and m.

   So a formal parameter with a subscript may be

   ZZ

10 Current issues

1. In the flex-bison world there are two kinds of tokens, those < 256, and those > 257. The first are used for tokens that are single characters and the second for compound tokens: both can be syntax tokens and operators. There are, for example, two kinds of operators in `factor`, the `/` (< 256) and the `\times` (> 257). Those < 256 are indicated by themselves in the grammar and the program; those > 257 have names. The `/` shows up in the grammar as `'/'`; the `\times` as `times_token`, which is kind of ugly. Do we want to give names to those < 256?

   Pro:

   * is uniform
   * abstracts from the actual shape

   Con:

   * long-winded

   How does this relate to our plans for an AND/OR grammar?

2. The grammar syntax.g contains the following token names in %token definitions that are not used in the syntax:

   formal_identifier
   variable_identifier

invisible_operator_token
max_operator_token
min_operator_token
power_operator_token

comment_text_token
Tcomment_token

These token names are used in the C files but this abuses the fact that *bison* accepts useless token (but not useless rules), and they should be defined in `node_types.h`.

3. The expression $1..n$ clearly means $1\ 2\ ...\ n-1\ n$, and $X_1..X_n$ clearly means $X_1\ X_2\ ...\ X_{n-1}\ X_n$. But then why doesn't it mean $X_1\ X_1{+}1\ ...\ X_n{-}1\ X_n$?

4. Terminology: what is a "subscript" and what is an "index"?

   * The expression evaluating to the position in the array is the "index". If it specifies a sequence of numbers, it is a "range".
   * The syntactic form attached to the array name is called the "subscript", even if it does not result in lowered text: the $[i..j]$ in $a[i..j]$ is the subscript.
   * The combination of array identifier and subscript is called an "element" (or "single-length segment") if there are no ranges involved, and a "segment" otherwise.

5. There are two ways to write down a multi-dimensional range: `1,1..m,n` and `1..m,1..n`. Since having them both implies conflicting precedences for `,` and `..`, it seems better to only allow `1..m,1..n`.

6. The syntactic problems with relational set operators remain unsolved. Since they can be defined locally, there is no way to parse things like `a \xi b` and `\xi (a, b)` correctly. They are parsed using the invisible operator, the nature of which will have to be assessed during type analysis, as it will for `2n` (multiplication) or `x y z` (concatenation).

7. $(3, 9) \in \rho$, ρ is a relation.

8. $\rho \circ \sigma$, $\rho \cdot \sigma$ or $\rho\sigma$ for composition? We also need an `operator_expression`, for example $\rho \cdot (\sigma^2 \cdot \tau)^*$.

   We probably cannot use the Invisible Operator for composition. ˚

   ˚ horrible example wanted

9. Partial parametrization: $F(a, , c)$.

10. Type definitions inside parameters in function definitions:

    function $F(a \in A, bcd, e \in \Sigma, b, d \in T, c \in T^*)$:
        return 1

    and, with parentheses, in operator definitions:

    $a\ \sigma\ (b \in I) \equiv 1$

    Likewise in generators:

$$a \in A, bcd, e \in \Sigma, b, d \in T, c \in T^*$$

11. Expression Calls

    Do we allow expression calls like

    $$(x^2 + 4x + 7)|_{x=2}$$

    resulting in 4+8+7=19? Or even

    $$(x^2 + 4x + 7)|_{x=2}^{x=5}$$

    for $\{19, 28, 39, 52\}$?

    Answer: yes, and we supply a Teckel command for it.˚ ˚ which?

12. Is there something useful we can do with `\langle` (⟨) and `\rangle` (⟩)?

13. Formal parameter check:

    $\xi\ a_{1..k} \equiv ...$ is OK;

    $\xi\ a_k \equiv ...$ is not OK.

11 Addendum: The manual parser

The manual parser (2007) did not work satisfactorily. The main problems were:

- The grammar for abstract algorithms was not known, and had to be developed on the fly.
- There was no good parsing technique for half-backed grammars and the "incremental" approach proved unsatisfactory. It was based on the idea of transforming the linear list of input token gradually into a parse tree, each time a new feature was added. In the beginning this worked well, mainly by supplying precedences to many tokens. But when, after adding N features, feature $N - k$ had to be changed or removed, all code for features $N - k + 1...N$ had to be checked and often adjusted.

See more detailed problems in `history/2007,manual_parser/parser.c`. In the end I decided to try designing a real grammar.