Mango: A Parser Generator for Self

Ole Agesen

SMLI TR-94-27 June 1994

Abstract:

Mango is a parser generator that is included in Release 3.0 of the Self system. Mango goes beyond LEX/YACC in several respects. First, Mango grammars are structured, thus easier to read and map onto parse trees. Second, Mango parsers automatically build parse trees rather than merely provide hooks for calling low-level reduce actions during parsing. Third, Mango automatically maintains invariance of the structure of parse trees, even while grammars are transformed to enable LR parsing. Fourth, Mango and the parsers it generates are completely integrated in the Self object world. In particular, a parser is an object. Unlike YACC, several independent parsers can co-exist in a single program.

We show how to generate a Mango parser and how to use it by means of an example: a simple expression language. Furthermore, we show how to add semantic properties to the parse trees that the Mango parser produces.

Mango is a realistic tool. A parser for full ANSI C was built with Mango.

Sun Microsystems Laboratories, Inc.
A Sun Microsystems, Inc. Business
M/S 29-01
2550 Garcia Avenue
Mountain View, CA 94043

email address:
ole.agesen@eng.sun.com

Mango: A Parser Generator for Self[*]

Ole Agesen

Computer Science Department
Stanford University, Stanford, CA 94306, USA

Sun Microsystems Laboratories, Inc.
2550 Garcia Avenue
Mountain View, CA 94043, USA

1 Introduction

Mango is a generator for integrated lexers and parsers in the Self system (release 3.0). In this document we describe how to use Mango and give some design rationale. No attempt is made to give a complete description of the inner workings of Mango. The intended audience for this document is primarily the person who wants to use Mango to build a parser for some application. Secondarily, the person who is designing an object-oriented parser generator may benefit from reading the discussion sections.

We assume some familiarity with parsing terminology and technology such as is described in the “dragon book” [1]. Experience with the use of other parser generators, e.g., LEX/YACC [2], is also an advantage. Like YACC, Mango generates lexers and parsers that take streams of characters as input and deliver streams of tokens or parse trees as output. Both

* The Self project began at Stanford University, but in early 1991, the project moved to its present home, Sun Microsystems Laboratories, Inc.

the parsers and lexers are of the LR kind, using a choice of SLR(1), LALR(1), or LR(1) parse tables.

Mango offers several advantages over YACC to the Self programmer.

• YACC is file-based; Mango is object-based. YACC was designed for ease of use in a UNIX® shell or a make file; YACC processes input files to produce output files. Mango, on the other hand, was designed to be a convenient tool for the Self programmer; Mango itself executes in the Self world.

• YACC generates a parser that interfaces well with C or C++ code; Mango generates a parser that is itself a Self object. Attempting to use a YACC parser in the Self environment would require extensive use of "glue" (Self's mechanism for calling C/C++ routines) and translation of data structures between C and Self representations. A Mango parser, on the other hand, is integrated within the Self world; the parser is a Self object, it executes entirely in the Self world, it parses Self objects (e.g., strings) and produces parse trees that are Self objects.

• While a YACC parser can be tailored to do complex things by inserting appropriate parsing actions, writing such actions is often tedious since their execution order is fixed by the order that a bottom-up parser performs reductions. In contrast, Mango parsers provide a higher level abstraction; they produce parse trees which can be decorated with attributes *after* the parsing is completed. Hence nodes can be visited in any order desired, including top-down.

• YACC grammars are powerful and flexible because they are unstructured, but the structured grammars of Mango are often easier to read and debug, and also impose structure on the parse trees that the parsers produce. On the negative side, it may be harder to write structured grammars that do not lead to conflicts in the resulting parse tables; more on this can be found in Section 2.

Simply by being object-oriented, Mango has addressed a major weakness of YACC. Mango parsers are reified as first class objects. Thus, once the first parser for a given language has been constructed with Mango, additional independent parsers can be obtained in an instant by cloning. Unlike Mango, YACC does not support multiple independent parsers in the same program.

Mango is based on structured grammars, a variant of context-free grammars. Structured grammars are described in Section 2. The description is not specific to Mango. Section 3 illustrates how to use Mango to build a parser for a simple language of arithmetic expressions. Use of the parser is also demonstrated. The parse trees produced by Mango parsers are described in Section 4. Section 5 describes how keywords such as `THEN` and `PROCEDURE` in Pascal are recognized by a Mango parser. Section 6 introduces a Mango parser for ANSI C [3]. This parser can be used as a skeleton for implementing grammar-based tools for ANSI C. It also constitutes a large concrete example for study. Section 7 gives an overview of the files that implement Mango. Section 8 outlines possible future work, and finally Section 9 offers some conclusions.

2 Structured Grammars

Mango is based on structured grammars, which are a variant of context-free grammars. A *structured context-free grammar* (structured CFG) is a grammar for which:

- each nonterminal has exactly one production, and
- each production is structured.

A *structured production* is a production that has one of the following five forms (throughout this document, `A` is a nonterminal; `E`, `S`, and `Xi` are terminals or nonterminals):

- *Construction:* `A ::= X1 X2 ... Xn`

  "The nonterminal `A` derives the sequence `X1 X2 ... Xn`."

- *Alternation:* `A ::| X1 X2 ... Xn`

  "The nonterminal `A` derives one of `X1`, `X2`, `...`, `Xn`."

- *Non-empty list:* `A ::+ E S`

  "The nonterminal `A` derives a non-empty sequence of the form `ESES...ESE`." We refer to `E` as the element and `S` as the separator of the list derived by `A`. The separator is optional. If it is left out, `A` derives `EE...E`.

- *Possibly-empty list:* `A ::* E S`

  "The nonterminal `A` derives a possibly empty sequence of the form `ESES...ESE`." In other words, `A ::* E S` either derives ε or something that is also derived by `A ::+ E S`. The separator is again optional.

- *Optional:* `A ::? E`

  "The nonterminal `A` derives `E` or ε."

The concept of structured grammars is described in detail in *An Object-Oriented Metaprogramming System* [5]. Structured grammars, in fact, are the basis of the Mjølner programming environment [4].

The semantics of structured productions may be precisely defined in terms of equivalent unstructured productions. Table 1 shows the expansions used by Mango. If a list production has no separator, simply leave out the separator from the expansion given in the table. For the list productions, it may seem that simpler expansions would suffice. This is correct in principle; however, Mango needs the extra level of indirection provided by `A'` in order to correctly terminate list nodes when building parse trees.

Mango generates a parser from a structured grammar in two steps. First, it expands the structured grammar into an unstructured grammar. The structured and unstructured gram-

Structured production	**Equivalent unstructured productions**
`A ::= X1 X2 ... Xn`	`A → X1 X2 ... Xn`
`A ::| X1 X2 ... Xn`	`A → X1 | X2 | ... | Xn`
`A ::+ E S`	`A → A′,   A′ → E | A′ S E`
`A ::* E S`	`A → A′,   A′ → A′′ | ε,   A′′ → E | A′′S E`
`A ::? E`	`A → E | ε`

Table 1. Structured productions and their expansion into unstructured productions.

mars both define the same language. Second, Mango uses standard techniques to generate a parser from the unstructured grammar [1].

2.1 Structured vs. unstructured grammars

Structured grammars are often easier to read than unstructured grammars, because common idioms such as lists and optionals are expressed directly using an easily recognized syntax. For example, with an unstructured grammar, the reader has to look for cycles involving a given nonterminal to determine whether it derives a list. In contrast, there is a single structured production that captures the list concept. Another factor improving legibility is that each nonterminal has exactly one production in a structured grammar.

A disadvantage of structured grammars is that they tend to be verbose, since it is necessary to name "intermediate" nonterminals that are often implicit in unstructured grammars. For example, the unstructured productions

```
A → UX | UY
```

have the following equivalent structured productions

$$\begin{aligned} &A \;\; ::| \; A_1 \; A_2 \\ &A_1 \; ::= UX \\ &A_2 \; ::= UY. \end{aligned}$$

Giving explicit names (A_1, A_2) to the alternatives (UX, UY) may improve legibility if the names are well chosen, but more importantly, the names can be used to define the interface to parse trees nodes, allowing these to be understood entirely in terms of the structured grammar (see Section 4).

While it is important that a grammar is easy to read, it is equally important that a parser can be effectively derived from it. Here structured grammars *a priori* stand at a disadvantage compared with unstructured grammars. If the expansion of the structured grammar yields an unstructured grammar that is not LR(1), the grammar writer must address the problem indirectly, at the structured level. To do this effectively, he must have some understanding of the relationship between the structured and unstructured grammar. A good analogy is the difference between working in Pascal and machine code: the Pascal programmer has given up the total control of which machine instructions are executed, but in return he has gained the ability solve the problem at a higher abstraction level than the machine code programmer. While it would be desirable to hide the existence of the intermediate unstructured grammar from the user of Mango, we have not taken this step, mostly due to lack of time.

James Roskind's C and C++ grammars [6] are good examples of how the low-level nature of unstructured grammars may be exploited. He has written unstructured grammars that are well suited for LALR(1) parsing in the sense that there are few conflicts in the parse tables. There is a price, however: his grammars are hard to read and extend with semantic actions. In order to eliminate conflicts, Roskind had to do massive inline expansion of nonterminals, resulting in grammars that are not highly factored, have multiple occurrences of several patterns, and have many right hand sides (such as `UX` and `UY` above) whose meaning in terms of the parsed language is not obvious (they do not directly correspond to concepts that the typical C or C++ programmer recognizes).

2.2 Transformations

Simply expanding each production of a structured grammar may yield an unstructured grammar that is not well suited for LR parsing. For example, the expansion of perhaps-empty list productions and optional productions introduce ε-productions into the unstructured grammar. When there are many ε-productions in a grammar, conflicts often show up in the parse tables.

To counter these problems, Mango offers a set of transformations that can be applied to the unstructured grammar to eliminate certain production patterns that often lead to conflicts. Transformations in Mango are both *language* and *structure* preserving: applying a transformation to a grammar does not change the language that the grammar defines, and it does not change the structure of the parse trees that the resulting parser produces. Another way of saying this is that transformations have one effect only: making it more "likely" that a grammar is LR(1) (or LALR(1)/SLR(1)) by eliminating certain production patterns that often imply conflicts.

Structure preservation is crucial because it allows the programmer to fully understand parse trees in terms of the source grammar. It also relieves the programmer from a potentially significant burden when he starts adding semantic actions to the parse trees: he can write the actions in terms of the untransformed, well-structured grammar, and rest assured that their effect remains as he expects, despite the transformations applied to the grammar.

In contrast, the hand-inlining that Roskind did to make his C and C++ grammars YACC-able, cannot be ignored, because the programmer using these grammars will have to recognize "related" productions and give them suitably "related," but probably not identical, parse actions.

We illustrate the effect of transformations with a concrete example in Section 2.2.1. Section 2.2.2 gives a detailed description of the transformations supported by Mango and offers some guidelines in choosing which ones to apply.

2.2.1 How transformations help grammars become LR(1)

To illustrate the effect of transformations, consider this structured grammar:

```
<start>   ::|  <alt1> <alt2> ;
<alt1>    ::=  <d0> <c0> <d0> 'a' ;
<alt2>    ::=  <d0>      <d0> 'b' ;
<c0>      ::?  'c' ;
<d0>      ::=  'd' ;
```

Expanding it to an unstructured grammar according to Table 1 and computing the LR(1) parse table, one shift/reduce conflict will be encountered. The reason is intuitively that an LR parser does not have enough information to choose between `<alt1>` and `<alt2>` (this choice is forced to occur early because of the `<c0>` in the right hand side of `<alt1>`; the `<d0>` on the other hand makes the parser see the same lookahead in the two cases it is trying to choose between). The expansion of these structured productions, using the equivalences given in Table 1, is the following unstructured productions:

```
<start>→ <alt1> | <alt2> ;
<alt1> → <d0> <c0> <d0> 'a' ;
<alt2> → <d0> <d0> 'b' ;
<c0>   → 'c' | ε ;
<d0>   → 'd' ;
```

In a Mango grammar, the line

```
Transformations: 'elimEpsilons';
```

directs Mango to post-process the unstructured grammar to eliminate ε-productions. The result in the concrete case is the following unstructured productions:

```
<start>→ <alt1> | <alt2> ;
<alt1> → <d0> 'c' <d0> 'a' | <d0> <d0> 'a' ;
<alt2> → <d0> <d0> 'b' ;
<d0>   → 'd' ;
```

This set of productions has no LR(1)-conflicts, intuitively because the choice between `<alt1>` and `<alt2>` is made at a time when the parser has enough information available.

2.2.2 Transformations supported by Mango

The situation illustrated by the example in the previous section is typical for the unstructured grammars resulting from expanding structured grammars. To handle this situation and several similar ones, a set of transformations can be requested of Mango. The transformations direct Mango to inline certain nonterminals in the unstructured grammars that are results of the straightforward expansion of structured grammars. Distinct sets of transformations can be applied to the syntactical and lexical parts.

The two most important transformations that Mango supports are “`elimEpsilons`” and “`elimSingletons`.” `elimEpsilons` remove all ε-productions and `elimSingletons` remove nonterminals with only a single production. When writing a grammar, we recommend applying both of these transformations since the cost is small (the parse trees remain unchanged). If the transformations blow up the size of the grammar too much (always an inherent danger when inlining), taking out one or both may be a worthwhile experiment, watching out to see if this creates (additional) parse table conflicts. On the other hand, if conflicts remain after applying both elimEpsilons and elimSingletons, some of the nonterminal-specific transformations described below should be tried. It is possible, of course, that even this does not solve the problem, in which case it may be necessary to refactor the grammar by hand.

The following is a complete list of the transformations supported and a precise description of how they affect the unstructured grammar:

- *elimEpsilons*. This transformation eliminates all ε-productions (except one, if the grammar derives ε) from the unstructured grammar. The elimination is done by repeating the following until either there are no more ε-productions, or there is only one and its left hand side is the start symbol:

  1. Pick a nonterminal, `A`, which has an ε-production: `A` → ε | α1... αn.
  2. Replace each right hand side, βAγ, in which `A` occurs, with `n+1` new right hand sides: βγ, βα1γ, ..., βαnγ.

  A concrete example was given in Section 2.2.1.

- *elimSingletons*. This transformation eliminates all nonterminals that have only a single production (in the unstructured grammar). The right hand side of the nonterminal being eliminated is simply inlined in all places where the nonterminal occurs on a right hand side. For example, the productions

  ```
  A → DBE | FG ;
  B → XY ;
  ```

  will be transformed to

  ```
  A → DXYE | FG;
  ```

- *inline: nonterminal-name*. This transformation inlines the given nonterminal at all its uses, whether or not the nonterminal has one or several productions.

- *flatten: nonterminal-name*. This transformation performs a complete recursive expansion of the named nonterminal which of course must be non-recursive, i.e., can only derive a finite number of terminal strings. For example, given these structured productions:

  ```
  <integer> ::= <sign> <length>      ;
  <sign>    ::| 'signed' 'unsigned'  ;
  <length>  ::| 'short' 'int' 'long' ;
  ```

the result of expanding and then applying `flatten: integer` are these unstructured productions:

```
<integer> →    'signed'   'short'   |
             'unsigned'   'short'   |
               'signed'   'int'     |
             'unsigned'   'int'     |
               'signed'   'long'    |
             'unsigned'   'long'    ;
```

• *dontInline: nonterminal-name*. This transformation prevents the named nonterminal from being inlined (by any of the sweeping transformations, `elimSingletons` and `elimEpsilons`). This may be necessary to ensure that reductions are performed in a certain order when the parse tree initialization methods have side effects. (In particular, the parser for ANSI C utilizes this to ensure timely recognition of `typedef`'ed identifiers).

• *useCharClasses*. This transformation applies to the lexers only. When lexing, it is often the case that many different input characters are treated the same way. For example, when lexing C, whenever a `2` is allowed on input so is a `3`. The size of the lexer tables can be significantly reduced by partitioning characters into equivalence classes and lexing based on the equivalence class of the input characters (this can be done with no loss of speed). The space savings are particularly large when working with a 16-bit character set, but even with an 8-bit character set the savings are considerable. For example, the tables in the lexer for ANSI C have 810 states without character classes, but only 318 states with.

The use of all these transformations is illustrated by two grammar files, `stGrammar.grm` and `Ansi-C.grm`. These files are part of the set of files that implement Mango (an overview can be found in Section 7). The files are also included in the Appendix. The transformations are summarized in Table 2.

Transformation	**Effect**	**Restrictions**
`elimEpsilons`	Eliminate ε-productions except perhaps one if the grammar derives ε.	
`elimSingletons`	Eliminate nonterminals that have only one production	
`inline: nonterminal`	Inline all right hand sides of nonterminal at all occurrences of nonterminal	
`flatten: nonterminal`	Recursively expand given nonterminal	Nonterminal must be nonrecursive
`dontInline: nonterminal`	Prevent nonterminal from being inlined by `elimEpsilons` and `elimSingletons`	
`useCharClasses`	Use character equivalence classes (saves space)	Only allowed in lex part

Table 2. Summary of transformations supported by Mango.

3 Building a Parser

This section shows how use Mango to build a parser for a language (see Figure 1 for an overview). First, a grammar for the language must be written. Then, Mango processes the grammar to construct a parser for the language. The parser can be used to parse strings in the language yielding parse trees. We illustrate this three step process by example, building a parser for the language defined by the grammar shown in Figure 2. The grammar defines arithmetic expressions consisting of integer literals, parentheses, and the standard four arithmetic operators, "`+-*/`." This grammar can also be found in the file `mangoTest.grm`.

Section 3.1 describes the syntax of grammar files, then Section 3.2 shows how to process a

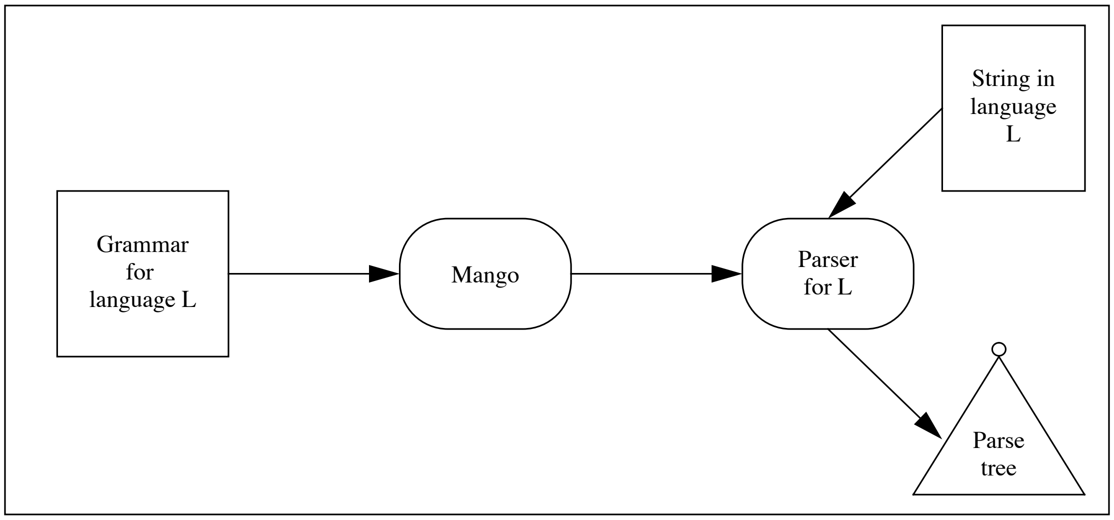


Figure 1. Mango takes grammars as input and produces parsers. The parsers, in turn, take strings as input and produce parse trees.

grammar file to obtain a parser, and finally Section 3.3 demonstrates how to use the generated parser.

3.1 Grammar file syntax

Figure 2 shows the complete grammar file for the expression language. We will explain grammar files by referring to this example.†

† A formal and precise description of grammar files can be found in the file stGrammar.grm (see the Appendix). It is a meta-grammar, i.e., the grammar for grammars, that is used for bootstrapping the parser generator.

```
Name:     'mangoTest'                  (* Name of this grammar. *)
Behavior: 'mangoTest.behavior.self'  (* File with behavior.   *)

Syntax:  SLR(1)
Transformations: 'elimEpsilons', 'elimSingletons' ;
  <exp>           ::+  <term>   <addOp> ;
  <term>          ::+  <factor> <mulOp> ;
  <factor>        ::|  <parenthesized> {number} ;
  <parenthesized> ::=  '(' <exp> ')' ;
  <addOp>         ::|  '+' '-' ;
  <mulOp>         ::|  '*' '/' ;

Lex:  SLR(1)
Transformations: 'elimEpsilons', 'elimSingletons', 'useCharClasses';
  {whitespace}    ->  [ \t\n]+ ;
  {number}        ->  {digits} ('.' {digits})? ;
  {digits}         =  [0-9]+ ;
```

Figure 2. Structured grammar for arithmetic expressions.

First, comments in grammar files are Pascal style, i.e., bracketed within "`(*`" and "`*)`". They can nest.‡ Mango grammar files consist of three parts: header, syntax, and lexical part.

The *header*, which is the first two lines in Figure 2, contains two pieces of information: a name for the grammar and the name of a file containing behavior that will be added to the resulting parse trees.

The significance of the name is two-fold. First, Mango parsers use the name to generate their `printString` to make them easily recognizable. Second, during parser generation, Mango may produce log files that give a summary of shift/reduce and reduce/reduce conflicts. The log files' names are constructed by appending `-syntax-unstruct.conflicts` and `-lex.conflicts` to the name of the grammar (the former log file is for conflicts found while generating the parser; the latter is for conflicts encountered while generating the lexer).

‡ So if you need to write a grammar for a language that allows nested comments, you can find an example to start from in `stGrammar.grm`. Incidentally, the lexer can handle nested comments because Mango translates the lexical definitions into a context-free grammar, and thus uses a full LR parser to lex. In contrast, LEX/YACC use a simple automaton to lex, hence must defer dealing with nested structures to the parsing phase.

There is an inherent performance penalty for introducing the full generality of an LR parser at the lexical level. The penalty stems from the requirement that the lexer finds the *longest* matching token. To find the longest token, the lexer must snapshot its state whenever it "leaves" a match to look for a potential longer match. In these situations, a Mango lexer must snapshot the equivalent of the state of a stack automaton, whereas a LEX lexer only has to snapshot the equivalent of the state of a deterministic automaton. We estimate the overall performance penalty to be approximately 30%, but thus far we have no direct measurements.

The *syntax part* describes the syntactical structure of the language. It begins with the keyword `Syntax:` followed by the kind of parse table to be used, one of: SLR(1), LALR(1), and LR(1). Next comes an optional list of transformations (see Section 2.2). The remainder of the syntax part is the productions in the grammar. They must be of the five structured kinds. The first production is the start production. Nonterminals are written as a name enclosed in "<" and ">," for example: `<exp>` and `<factor>`. Terminals can have two different appearances. First, `{number}` refers to a terminal which is defined in the lexical part of the grammar file. Second, "`'+'`" (or any other string between single quotes) denotes a literal terminal. This inline definition of terminals is particularly useful for a succinct specification of language constructs containing keywords such as `IF`, `THEN`, and `ELSE`.

The *lexical part* starts with the keyword `Lex:` followed by the parse table kind, one of: SLR(1), LALR(1), and LR(1). Again a set of transformations may be specified (see Section 2.2). The bulk of the lexical part consists of a set of definitions. There are two kinds of definitions: terminal definitions and internal definitions.

Terminal definitions define terminal symbols which may be used in the syntax part to express the syntax of the language. Terminal definitions use the binder "`->`" between the defined name and the defining expression. For example:

```
        {whitespace} -> [ \t\n]+ ;
```

defines that a whitespace token consists of a non-empty sequence of blank, tab, or newline characters. Since the definition uses the binder "`->`" the name being defined is a terminal, hence `{whitespace}` can be used in the syntax part of the grammar.

Internal name definitions define names that can be used as part of expressions in the lexer part only. They are not visible in the syntax part of the grammar. Internal names are defined using the binder "=." For example:

```
        {digits} = [0-9]+ ;
```

defines the internal name `{digits}` as a non-empty sequence of digits. The internal name `{digits}` is used in the definition of the terminal `{number}` (see Figure 2).

In terms of a lexer/parser chain, the difference between terminals and internal names is that the lexer may send tokens corresponding to terminals to the parser, but will never send anything that corresponds to an internal name.

The right hand side of both kinds of lexical definitions is a regular expression. Table 3 shows the operators that can be used. Each has its standard meaning.

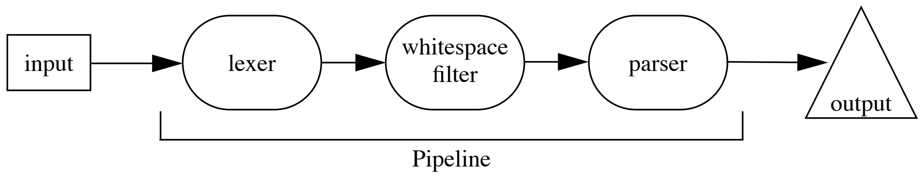
Operator	**Meaning**	**Fix**	**Arity**	**Priority**
`?`	optional	postfix	unary	high
`+`	non-empty closure	postfix	unary	high
`*`	perhaps-empty closure	postfix	unary	high
(blank)	concatenation	infix	binary	medium
|	alternation	infix	binary	low

Table 3. Regular expression operators that can be used in Mango lexical definitions.

The operators have priorities as indicated in the table, but parentheses can be used to group expressions when the priorities must be resolved differently. The primary expressions are: lexical names (e.g., `{digits}`), char sets as in LEX/YACC (e.g., `[a-zA-Z]`; however, that the "`.`" meta character is not supported in Mango), and string literals (e.g., `'IF'`).

3.1.1 The whitespace filter

It is often convenient to ignore whitespace when defining the syntax of a language. The definition of whitespace must, of course, still be specified in the lexical structure for the language—the convenience is merely that it is not necessary to include whitespace occurrences in the syntactical specification. To facilitate this, Mango treats the terminal `{whitespace}` specially. If this terminal is defined in the lex part, but does not appear in the syntax grammar, it will be filtered out before the syntax parser sees the stream of tokens. This is accomplished by inserting a filter in the stream of tokens between the lexer and the parser:



When the filter is inserted, the parser generator will print the message, "Inserting whitespace filter," during generation of the parser. For example, in the expression grammar, the lexical part defines `{whitespace}` to be any non-empty sequence of spaces, newlines and tabs. These tokens are never seen by the parser, hence, need not complicate the syntax grammar.

The definition of the `{whitespace}` is not hardwired into Mango. Instead it must be defined like any other nonterminal. For example, in Figure 2 `{whitespace}` is defined as a non-empty sequence of blank, tab, or newline. We could conceivably generalize the filter idea by allowing tokens other than `{whitespace}` to be filtered out. However, we have not found a need for this generality yet.

The filtered-out whitespace tokens are not thrown away. Instead they are stored in the preceding non-whitespace token, and can be obtained by sending `whitespace` to this token. If the input to the parser has a trailing whitespace, it is stored in a special `endMarker` token that is sent through the pipe when the end of the input is encountered.

3.2 Processing a grammar

We now describe how to generate a parser from a grammar. Please refer again to Figure 2. Before working through the example, `cd` to the directory that contains Mango, and check that the Self `_DirPath` contains “`.`”.

The first step is to read in the parser generator (if you already have a snapshot that includes the parser generator, you may ignore this step):

```
Self 1> 'mango.self' _RunScript
```

This will take a few minutes. Upon completion, it will either give you an error (hopefully not!) or inform you that the parser generator has been successfully bootstrapped. This may be a good time to write out a snapshot. The result of the bootstrapping is a parser for structured grammars. Now use this parser to *parse* the grammar for our expression language (remember, the expression grammar is found in the file `mangoTest.grm`):

```
Self 2> shell _AddSlots: (| grmParser. expParser. |)
Self 3> grmParser: mango parsers stGrammarParser copy
Self 4> grmParser parseFile: 'mangoTest.grm'
Self 5> expParser: grmParser output makeParser
```

Here’s an explanation of the commands issued. First (`Self 2>`) we add a couple of slots to the shell for holding objects. Next (`Self 3>`) we store a copy of the parser for structured grammars in `grmParser`. Then (`Self 4>`) we parse the file containing the expression grammar. The result, if everything goes well, is obtained by sending `output` to the grammar parser (`Self 5>`). In other words, `grmParser output` is the result of parsing `mangoTest.grm`. This result is a parse tree which has special behavior added to its nodes. One of the messages that it understands at the root node is `makeParser`. The result of sending `makeParser` to the parse tree is a parser for the expression grammar.** We store the resulting parser in `expParser`. If the parsing in step (`Self 4>`) fails, an error message will be displayed, detailing the nature of the error. For example, if the error is a syntax error on the file being parsed, the line number and column of the file will be displayed together with the offending token on input and a list of possible alternatives (you

** The expression `grmParser output makeParser` takes a little while. A couple of warnings may be printed; don’t worry about this for now.

can see this by temporarily introducing an error in the “mangoTest.grm” file and retrying the command numbered 4 above).

To help debugging, there is a global slot, `noiseLevel` that can be set to an integer between 0 and 3. The higher the number is, the more diagnostic messages are printed out during generation of parsers. To set the noise level to `3`, type

```
mixins mango tracer noiseLevel: 3.
```

3.3 Using a parser

In this section, we demonstrate how to use the parser we just generated. Here are some typical commands:

```
Self 6> expParser
parserPipeline(charStreamer('') ->
               lrParser(mangoTest-lex) ->
               filter(whitespace) ->
               lrParser(mangoTest-syntax-unstruct))
Self 7> expParser parseString: '3 + 4 * 8'
parserPipeline(...)
Self 8> expParser output
exp_node
Self 9> expParser output eval
35
```

The first command (`Self 6>`) simply shows the `printString` of the generated parser. It is a pipeline with four stages:

- The first stage is a character streamer that is responsible for receiving input to parse. A parser pipeline can either parse a file (when sent the message `parse-File:`) or a string (when sent the message `parseString:`, see command 7).

- The second stage is the lexer. It collects individual characters received from the character streamer into tokens. For example `234` would be a token. The lexer is itself an LR parser.

- The third stage is the aforementioned whitespace filter (see Section 3.1.1). It eliminates `{whitespace}` tokens from the token stream produced by the lexer, but lets all other tokens pass through to the parser.

- The fourth stage is what is normally referred to as simply “the parser.” It is an LR parser. It receives a stream of tokens from the whitespace filter. It will execute until it has received enough tokens to produce a single item of output which will be a parse tree.

Returning to the Self commands in the code above, in (`Self 8>`) the result of parsing the string `'3 + 4 * 8'` is retrieved by sending `output` to the pipeline. The result is a single parse tree node of type `exp_node` which matches our expectations since the start symbol in the grammar is `<exp>`. If the input stream has enough symbols on it to produce more than one output item, the next output item will be presented on the output when the pipeline receives the message `skipOutput`.

Finally in (`Self 9>`), we send `eval` to the `exp_node`. `eval` is a method that will evaluate the expression and return its result. `eval` is defined in the behavior file (see Section 4.2 below).

3.3.1 Important messages that can be sent to parser pipelines

The following is a list of the most important messages that can be sent to a parser pipeline:

- `copy` returns a copy of the entire parser pipeline. Thus, it is easy to obtain multiple parsers for a given language once the first parser has been generated.
- `parseString:` parses a string. There is also a version that takes a failure block: `parseString:IfFail:`.
- `parseFile:` parses a file. The argument is the name of the file which must have read permission. Again, there is a version that takes a failure block: `parseFile:IfFail:`.
- `output` returns the current output of the parser. This message is idempotent.
- `skipOutput` advances the parser to the next output item and returns self.
- `hasMoreInput` tests if there is unconsumed input on the parser pipeline.
- `lastParser`, `firstParser` return the last/first parser of the parser pipeline. The returned parsers can be sent `prevParser` and `nextParser`.

Finally, the following messages may be useful for debugging and/or timing individual stages of a parser pipeline:

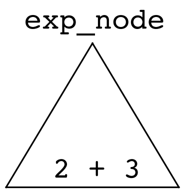
- `copySize:` copies the first `n` stages of the parser pipeline (`n`, an integer, is the argument).
- `->` splices a new parser onto the end of the pipeline and returns self. Mango uses this method internally to build the parser pipelines.

4 Parse Trees and Adding Behavior

The result of parsing a string or file is a parse tree that is obtained by sending `output` to the parser pipeline. The structure of the parse tree corresponds to the grammar that the parser was generated from. This section explains the correspondence between a grammar and its parse trees. We first describe the parts of parse trees that are defined for any grammar. Next we describe how to add grammar-specific behavior to the parse trees by specifying a behavior file.

4.1 General behavior of parse trees

Suppose a string is derived from some grammar using a certain sequence of productions. The parse tree resulting from parsing this string is the standard derivation tree as described in [1]. It has an interior node for each nonterminal expanded in the derivation and a leaf node for each terminal in the string. The root of the parse tree will correspond to the first production used (the start production). For example, the parse tree corresponding to the derivation of `2 + 3` from the grammar in Figure 2 will have this form (ignoring the inner structure for simplicity):



It is preferable to be able to understand what the interface of a parse tree node is by simply looking at the grammar. This can easily be accomplished with structured grammars. In the next section we describe how the interface of nonterminal nodes is determined, Section 4.1.2 describes the interface for terminal nodes, and finally Section 4.1.3 describes the interface that is common for both nonterminal and terminal nodes.

4.1.1 The interface of nonterminal nodes

With Mango, a node corresponding to a nonterminal will have an interface that is determined entirely by the nonterminal's production. This direct relation is ensured by having a parse tree *node type* corresponding to each nonterminal. A node corresponding to a nonterminal `A` is denoted an `A`-node. A node type, as is customary in Self, is implemented by a pair of objects: a traits and a prototype. We will, however, omit the traits/prototype distinction from the figures we give below, to ensure greater clarity.

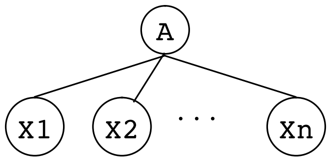
The interface of `A`-nodes is defined by the format of `A`'s structured production. There are five cases, depending on the kind of structured production that `A` has:

- *Alternation*. If the nonterminal `A` has the production

  ```
  A ::| X1 X2 ... Xn ;
  ```

  the effect is that the `A`-node will be an abstract supertype for the `X1`-, `...`, `Xn`-nodes ("`X1` is an `A`, `X2` is an `A`, ..., `Xn` is an `A`"):

  The supertype relations are realized by copy-down inheritance, i.e., each `Xi`-traits has a parent pointer referring to the `A`-traits and the slots in the `A`-prototype are copied down into each `Xi`-prototype. For a concrete example,



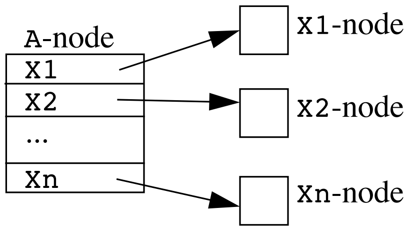
consider the node types that result from the grammar in Figure 2: `factor-node` is an abstract supertype of `parenthesized-node` and `number-node`.

If a symbol `Xi` occurs on the right hand side of several alternations, the corresponding node will have several supertypes (using Self's multiple inheritance).

- *Construction*. Suppose `A` has the production

  ```
  A ::= X1 X2 ... Xn ;
  ```

  then the `A`-nodes (the instances) in the parse tree will have slots named `X1`, `X2`, `...`, `Xn` ("`A` is composed of `X1`, `X2`, `...`, `Xn`"):



  Returning to the expression grammar example, a `parenthesized-node` has three slots, corresponding to "`(`," `{number}`, and "`)`" respectively.

  If some symbol kind, such as "`(`" is not a legal slot name in Self, a legal slot name will be automatically derived from it; in this case the derived slot name is `beginParen_`. To find out what a given string is mapped to, type

  ```
  mango symbols symbol asSlotName: 'some-string'.
  ```

- *List productions*. If `A`'s production is one of

  ```
  A ::+ E S ;
  A ::+ E ;
  A ::* E S ;
  A ::* E ;
  ```

  the `A`-nodes are list nodes. A list node understands the message `has_separator` (returning true for the first and third productions above and false for the second and fourth productions). All lists nodes respond to the message `elements` by returning a vector of the elements in the list; the elements will be `E`-nodes, of course. If the list has separators, the message `separators` will return a vector of the separators which will be one shorter than the list of elements (except if the vector of elements is empty in which case the separators will also be empty).

- *Optional*. If the `A` production has the form

  ```
  A ::? E ;
  ```

  the `A`-nodes will understand `opt_is_present`. The result will be `true` if the `E`-node is present, `false` otherwise. If the `E`-node is present, it can be obtained by sending `E` to the `A`-node, just as for a construction production with a single symbol on the right hand side.

4.1.2 The interface of terminal nodes

Terminal nodes form the leaves of parse trees. They can be sent the message `token` to obtain the token object produced by the lexer. This token object can in turn be sent `source` which returns the source string that was consumed by the lexer when the token was produced. The token object can also be sent `whitespace` which will return the whitespace token immediately following the receiver token (if the whitespace filter was used).

4.1.3 Common interface for terminal and nonterminal nodes

It is possible to determine the type of a node by sending it messages of the form `is_`*kind* where *kind* is the name of a grammar symbol. The return value is a boolean. For example, in the expression grammar introduced in Section 3, sending `is_exp` to the root node returns true:

```
Self 10> expParser output is_exp
true
```

Another group of messages implement iterators on parse trees. First, all nodes respond to the message `children_do:` by iterating the (block) argument over their direct children. List nodes, for example, will iterate over their elements and separators. Second, the messages `suffix_walk_do:` and `prefix_walk_do:` will iterate over entire subtrees (or trees, if the receiver is the root node).

4.1.4 Tags

Sometimes the default way of naming slots in parse tree nodes can fail. Tags can solve this problem by allowing the programmer to override the default names. Tags can also be used to give more meaningful or specific names to slots in parse tree nodes.

For example, suppose you are writing a grammar for Pascal. The production defining `IF` statements may look like this:

```
<if_stmt> ::= 'IF' <exp> 'THEN' <stmt>
                         'ELSE' <stmt> ;
```

Given an `if_stmt`-node, the problem with this definition is that there would be no way to distinguish the statements in the `THEN` part and the `ELSE` part: the two slots would have the same name. Using tags, different names can be assigned to occurrences of the same nonterminal. A better way to write the production for `<if_stmt>` is:

```
<if_stmt> ::= 'IF' <exp> 'THEN' <then_stmt:stmt>
                         'ELSE' <else_stmt:stmt>;
```

The tags, `then_stmt` and `else_stmt`, override the default slot names for `if_stmt`-nodes; thus, the two occurrences of the `<stmt>` nonterminal can be readily distinguished.

In general, a tagged nonterminal is written as `<tag:kind>`. The purpose of tags is to guarantee uniqueness of names in the public interface of parse tree nodes. When tags are present they will be used as the name of the corresponding slot in parse tree nodes. If tags are omitted they default to the "kind" of the grammar symbol. For example, `<stmt>` is equivalent to `<stmt:stmt>`. Any string of printable characters may be used as a tag, but if the string does not constitute a legal Self slot name a legal slot name will be derived and used instead (see Section 4.1.1). Tags containing the substring `internal0000` are reserved.

Terminals may also be tagged. A tagged terminal is written as `{tag:kind}`. Literals, however cannot be tagged; if you need to tag a literal in the syntax part, you must first promote the literal to be a nonliteral terminal, i.e., defined in the lex part, and then tag the relevant instances in the syntax part.

4.2 Adding behavior to parse trees

Parse trees themselves are rarely the final goal when a parser is used. Rather, they provide a structured foundation on which domain-specific behavior is built (such as, for example, a type checker and a code generator if the parser is part of a compiler).

Behavior can be added to Mango parser trees by means of a "behavior file." Mango will process this file during parser generation and ensure that the prototype parse tree nodes possess the behavior. Again, we illustrate this by referring to the expression grammar in Figure 2. The behavior file that is specified in the header of the grammar is `mangoTest.behavior.self`. Its contents are shown in Figure 3. It may not best demonstrate the Self programming style; however, we have tried to keep everything as straightforward as possible (rather than striving for the most elegant solution).

To understand the behavior file, observe that it describes a single Self object, the *behavior object*. The behavior object has a number of slots whose names are all also the names of symbols in the grammar (except for the `parent` slot which is there to make the object "well behaved"). For example, the behavior object in Figure 3 has a slot named `exp`. This slot contains the behavior that is added to the traits object of `exp`-nodes. It is also possible to add behavior to the prototype object of a parse tree node. This is done by specifying a slot with a name such as `exp_node_proto`. In the figure we have done this for illustration purposes only: the behavior added to the prototype of `exp`-nodes is never used.

```
( |
  _ parent* = traits oddball.

  ^ exp = ( |
    ^ eval = ( | s. |
       s: elements first eval.
       separators do: [|:sep. :idx|
          s: (sep performOpOn: s And: (elements at: 1+idx) eval).
       ].
       s.
     ).
  | ).

  ^ exp_proto = (|
     ^ thisSlotIsIgnored.
  | ).

  ^ term = (
    ^ eval = ( | s. |
       s: elements first eval.
       separators do: [|:sep. :idx|
          s: (sep performOpOn: s And: (elements at: 1+idx) eval).
       ].
       s.
     ).
  | ).

  ^ parenthesized  = (| ^ eval = ( exp eval. ). |).
  ^ number         = (| ^ eval = ( token source eval. ). |).
  ^ plus_          = (| ^ performOpOn: x And: y = ( x + y. ). |).
  ^ minus_         = (| ^ performOpOn: x And: y = ( x - y. ). |).
  ^ star_          = (| ^ performOpOn: x And: y = ( x * y. ). |)
  ^ slash_         = (| ^ performOpOn: x And: y = ( x / y. ). |).
| )
```

Figure 3. The behavior file that corresponds to the grammar in Figure 2.

In general, suppose the grammar contains a symbol `X`. The behavior that will be added to the traits object of `X`-nodes is found in the `X` slot in the behavior object. Similarly, the behavior that will be added to the prototype `X`-node object is found in the `X_proto` slot in the behavior object. One or both of `X` and `X_proto` may be omitted from the behavior object, in which case no behavior is added to the corresponding parse tree nodes.

The naming, `X` vs. `X_proto` reflects the fact that it is most common to add behavior to traits nodes. Slots added to the prototype parse tree nodes will be copied down into subtypes in accordance with the hierarchy specified by the alternation productions. Typically, state is added to the prototype nodes, and methods are added to the traits nodes.

The behavior for the expression grammar is quite simple since its only purpose is to evaluate the parsed expression. For example, the nodes corresponding to the operators "`+-*/`" simply define a message that performs the relevant operation (note how, e.g., the terminal kind "+" is mapped to the Self slot name `plus_`). The operator nodes are invoked by the `exp` and `term` nodes that simply send `eval` to the arguments and then perform the relevant binary operation on the results of the `eval`s.

Two optional slot names in the behavior object have special meanings. The slot name `shared_behavior` contains behavior that will be added to a common supertype of all node types in the given grammar. In other words, if a message is defined in the `shared_behavior` slot, it will be understood by every parse tree node, no matter what its type is. In the same way, the slot `shared_behavior_proto` describes behavior that is copied down into every concrete parse tree node type.

For an extensive example of how to add behavior to grammars, please refer to the file `stGrammar.behavior.self`. It is a companion file to the meta-grammar file `stGrammar.grm` and contains behavior that builds a parser from a parse tree of a grammar.

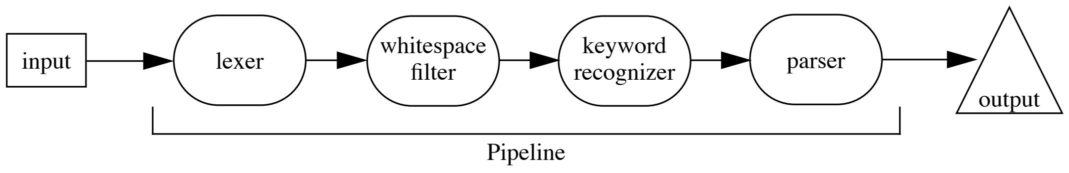
5 Keyword Recognizer

Suppose you are parsing Pascal. In Pascal an identifier has the following lexical characterization: `[A-Za-z]+[_A-Za-z0-9]*`. However, Pascal also contains numerous keywords that should preferably be tokenized to a token other than "identifier" in order to make parsing easier. (If a keyword such as `THEN` was lexed as "identifier," you might end up with a parser that accepts statements such as

```
THEN := 9;
IF x = y BLIP x := 9;
```

and would need a potentially complicated checker to catch such errors in a phase after parsing). A much more elegant solution is to provide a convenient way for tokens such as `THEN` and `IF` to be presented to the parser as something other than "identifier."

The keyword recognizer accomplishes this very thing. It is a simple kind of parser or filter that is inserted between the whitespace filter and the syntax parser. It is not always needed (when it is not needed, Mango avoids inserting it; for example, it is not needed in the expression grammar example).



The keyword recognizer has a set of those literals occurring in the grammar that also match the definition of some terminal (such as `THEN` in a Pascal grammar). The keyword recognizer works by simply watching the stream of tokens flowing from the lexer to the parser, changing the type of the tokens which are in its set of literals to the types of the corresponding literals.

6 The C Parser

The file `Ansi-C.grm` (see the Appendix) and an accompanying behavior file, `Ansi-C.behavior.self`, implement a parser for ANSI C. The parser is based on the grammar found in the book by Kernighan and Ritchie [3]. The grammar in the book is unstructured; the grammar in `Ansi-C.grm` is a structured version of it. There is currently no behavior added to the grammar (except what is needed for parsing, as described below). Adding behavior to this grammar would be a quick way to get started writing grammar-based C tools.

The structured and unstructured ANSI C grammars are similar (although there is one exception which is explained below). For example, in [3] the unstructured productions for expressions include:

```
<exclusive_OR_exp> → <AND_exp>                   |
          <exclusive_OR_exp> '^' <AND_exp> ;

<AND_exp> → <equality_exp>                  |
          <AND_exp> '&' <equality_exp> ;

<equality_exp> → <relational_exp>                  |
          <equality_exp> '==' <relational_exp> |
          <equality_exp> '!=' <relational_exp> ;

<relational_exp> → <shift_exp>                |
          <relational_exp> '<'  <shift_exp> |
          <relational_exp> '>'  <shift_exp> |
          <relational_exp> '<=' <shift_exp> |
          <relational_exp> '>=' <shift_exp> ;
```

And the corresponding structured productions in the Mango grammar are:

```
<exclusive_OR_exp> ::+ <AND_exp>        '^'      ;
<AND_exp>          ::+ <equality_exp>   '&'      ;
<equality_exp>     ::+ <relational_exp> <eq_op>  ;
<relational_exp>   ::+ <shift_exp>      <rel_op> ;

<eq_op>            ::| '=='  '!='                ;
<rel_op>           ::| '<'   '>'   '<='  '>='    ;
```

The productions for the `<type_specifier>`, `<type_qualifier>`, and `<storage_class>` nonterminals (not shown here, but they can be found in the Appendix) differ significantly between the unstructured and structured grammar. This is not just due to the difference between unstructured and structured grammars. For example, the unstructured grammar derives the string `float int x;` which is not correct C since there are two type names in the declaration. The structured grammar enforces that declarations can have at most one type name (it is correct C to have *no* type name in a declaration, since `int` is implied). Because the structured grammar is more restrictive, certain errors can be caught during parsing rather than being postponed to a later semantic checking phase. The rewriting also helps limit ambiguities in the grammar.

To build the C parser, simply read the file `genansi.self` into a Self image containing Mango. The result is that several slots are added to the `shell`. Among the slots is a data slot, `cParser`, that will be initialized to contain the generated parser for C. There is also a method that will invoke the standard UNIX preprocessor. For example, you can type the following:

```
Self 12> 'genansi.self' _RunScript
... lots of output ...
Self 13> ppAndParser: '/usr/include/stdio.h'
Preprocessing file '/usr/include/stdio.h'... done.
Parsing... done.
<shell>
Self 14> cParser output
translation_unit_node
Self 15> [|a<-0|
          cParser output suffix_walk_do: [a: a+1].
          a print. ' nodes.' printLine] value
455 nodes.
Self 16> cParser output fullSource
... the contents of the file "/usr/include/stdio.h".
```

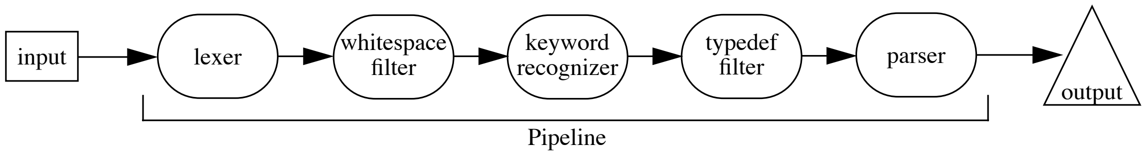
At a later date, it is possible that a C pre-processor written in Self will be integrated with the Mango C parser. Until then we will rely on the crude way of invoking the UNIX C preprocessor.

6.1 Resolving the typedef ambiguity

The C grammar contains a well-known ambiguity that must be resolved in order to parse successfully. The ambiguity has to do with the use of identifiers as type names after they have been `typedef`'ed.

The Mango C parser resolves the ambiguity by providing context dependent information when it is needed. This is done by splitting the terminal kind `{identifier}` into two: `{identifier}` and `{typedef_name}`. Selected `{identifier}` tokens are modified by a filter that is inserted between the lexer and parser in the pipeline. The filter is analogous to the `keywordFilter` described in Section 5. It is called a `typedefFilter`. It looks at all the tokens produced by the lexer. The default behavior is to pass the tokens onto the parser unchanged. However, if the token has kind `{identifier}`, and the particular instance has a name that matches a name that has previously been typedef'ed, the kind of the token is changed from `{identifier}` to `{typedef_name}`.

The `typedefFilter` must maintain a set of names that have been `typedef`'ed in the current context. Since C allows `typedef`'ed names to be re-declared as non-`typedef` things in inner scopes, the `typedefFilter` must be informed whenever a scope is entered or left. To do this, and to register when identifiers are `typedef`'ed, the parser must feed information back through the pipeline to the `typedefFilter`. The most critical, and hardest to implement part of this is that the feedback must take place during parsing to continuously keep the `typedefFilter` up to date; else the parser may go wrong because it bases its actions on information provided by the `typedefFilter`.



The feedback from the parser is implemented by methods called `initialize_node:` in the behavior for selected nonterminals. These methods are invoked whenever a parse tree node is created, i.e., during parsing (this is akin to YACC's low-level way of allowing the user to insert actions that are executed during parsing). The `initialize_node:` methods are invoked in the order that parse tree nodes are created. This order depends to some degree on whether nonterminals are inlined or not. In the C parser, it is crucial that the feedback to the `typedefFilter` arrives early, so we need precise control over which nonterminals are inlined. To express this, Mango has a special transformation, `dontInline:`, which is used to prohibit inlining of a nonterminal. The C grammar file contains several uses of it.

The inclusion of the `typedefFilter` in the parser pipeline is controlled by the presence of an option, `typedefKludge` (see the header of the file `Ansi-C.grm`). This option is supported by Mango solely for the purpose of parsing C.

7 Files

Table 4 gives a brief overview of the files that make up Mango. For each file, we give a high-level description of what the file implements.

File Name	What the file implements
`mango.self`	Reading in this file pulls in the entire parser generator and bootstraps it. Aside from reading in the other Mango files, this file creates a number of initialized name spaces.
`grammar.self`	This file defines unstructured grammars, terminals, nonterminals, productions, LR parse tables, and transformations.
`stGrammar.self`	This file defines structured grammars, productions, their expansion into unstructured grammars, and controls the creation of parse tree nodes.
`gramBuild.self`	This file contains an ad-hoc lexer and parser for grammar files. It is only used to bootstrap Mango.
`parsers.self`	Filters, LR parsers, and parser pipelines are defined here.
`prodSet.self`	Defines a prodSet which maintains a set of productions in a representation that supports efficiently performing transformations.
`ptokens.self`	Defines tokens that are passed between the lexers and parsers. It also defines the roots for parse tree nodes.
`treeBuilders.self`	This file implements a specialized "code generator" that generates the Self code that builds parse trees during parsing.
`typedefFilter.self`	This is a hack needed to parse C and possibly C++.
`stGrammar.grm`	The meta grammar or grammar for grammars. It is parsed during bootstrapping. This file is included in the appendix.
`stGrammar.behavior.self`	This file contains the behavior that is added to parse trees for grammars, allowing a parser pipeline to be constructed. It also defines how the regular expressions in the lex part are expanded into a context free grammar.
`mangoTest.grm`	This is the example grammar for expressions used in this manual (see Figure 2).
`mangoTest.behavior.self`	The behavior file for `mangoTest.grm` that allows expressions to be evaluated (see Figure 3).
`Ansi-C.grm`	The structured grammar for ANSI C. This file is included in the Appendix.
`Ansi-C.behavior.self`	The minimal behavior file for the C grammar. It implements the feedback of `typedef` information to the `typedefFilter`. This file is included in the Appendix.

Table 4. Overview of the files that make up Mango.

8 Future Work

Putting Mango into frequent use will doubtless identify several areas that need improvements or enhancements. Some areas where changes may be needed have already been identified. We list them here so that the reader may be aware that future versions of Mango may differ.

The expansion of structured grammars into unstructured grammars should be hidden to a greater extent. Among other things, hiding the expansion requires re-phrasing any parse table conflicts in terms of the structured grammar. Along the same lines, it would be preferable to have Mango automatically apply transformations as needed, rather than imposing the task upon the user.

The two-object (traits + prototype) implementation of tree nodes may be replaced with a single-object implementation. A default rule, stating that assignable slots are copied down and constant slots are inherited through a parent pointer, would probably handle virtually all cases correctly. This change would make the terminology for and use of semantic routines more lightweight.

A preprocessor for C has been written in Self. It should be integrated with the Mango C parser so that the crude invocation of the UNIX preprocessor is avoided.

9 Conclusions

Mango is an integrated LR(1) parser generator written in Self. It offers several powerful features that a Self programmer can benefit from if he/she needs to construct a parser. The areas in which Mango goes beyond a traditional parser generator such as YACC include: Mango is object-oriented, Mango grammars are structured, and Mango parsers produce parse trees whose nodes are arranged in a subtype hierarchy according to the grammar they are derived. The parse trees provide a structured foundation for defining semantic routines that can implement, e.g., type checking or code generation if the parser is part of a compiler. Mango does not only support adding methods to parse tree nodes; slots for holding state and parents for inheriting additional behavior can also be added. This level of expressive power can make it significantly easier to implement complex semantic routines than if the programmer is restricted to using actions that are invoked during parsing such as YACC insists.

A parser for full ANSI C has been constructed with Mango. C is not an easy language to parse since it has a large grammar and an ambiguity involving `typedef`'s that must be resolved with context dependent information. The ability to cope with C demonstrates clearly that Mango is a robust tool.

The C parser is interesting on its own since it gives the "full" picture of the C syntax. Specifically, we started with the C grammar in the standard C reference [3], but had to address two major issues before we had a satisfactory parser. First, we restructured the grammar to correct a deficiency that would otherwise allow illegal C type declarations with multiple type names to be derived. Second, we resolved the `typedef` ambiguity by timely supplying context dependent information to the parser. The restructured grammar and the Self code that collects the context dependent information is given in the appendix. Neither of these issues are addressed in Kernighan and Ritchie's *The C Programming Language* [3].

10 Acknowledgments

The author would like to thank David Ungar and Mario Wolczko for comments on an earlier draft of this paper. Also, thanks to the Self group in general for many useful discussions. The author would additionally like to acknowledge the support of the Natural Science Faculty of Aarhus University and the Danish Research Academy.

11 References

1. Aho, A. V., R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Massachusetts: Addison-Wesley, 1986.

2. Johnson, S. C. “YACC—Yet Another Compiler Compiler.” *AT&T Bell Laboratories Computing Science Technical Report* 32 (1978).

3. Kernighan, B. W. and D. M. Ritchie. *The C Programming Language*. 2d edition. New Jersey: Prentice Hall, 1988.

4. Knudsen, J. L., O. L. Madsen, C. Nørgaard, L. B. Petersen, and E. S. Sandvad. “An Overview of the Mjølner BETA System.” *Proceedings on Conference on Software Engineering Environments (SEE91)* (1991).

5. Madsen, O. L., and C. Nørgaard. “An Object-Oriented Metaprogramming System.” *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences* (January 1988): 406–415.

6. Roskind, J. *ANSI C and C++ Grammars*. Suitable for YACC parsing. Available by anonymous ftp from ics.uci.edu (128.195.1.1) in the ftp/gnu directory. Alternative site: mach.1.npac.syr.edu (128.230.7.14) in the ftp/pub/C++ directory.

Appendix A

The following are February 1994 versions of `stGrammar.grm` (the meta grammar or grammar for grammars), `Ansi-C.grm` (grammar for ANSI C, see Section 6), and `Ansi-C.behavior.self` (behavior file for C grammar, see Section 6).

stGrammar.grm

```
(* Sun-$Revision: 9.3 $ *)

(* Copyright 1992 Sun Microsystems, Inc. and Stanford University.
   See the LICENSE file for license information. *)


(* This (structured) grammar describes a representation for structured
   grammars. *)

Name:      'stGrammar'
Behavior: 'stGrammar.behavior.self'

Syntax:  SLR(1)
Transformations: 'elimEpsilons', 'elimSingletons' ;

  <start>          ::=  'Name:'       {name:string}
                        'Behavior:'   {behaviorFile:string}
                        <optionsPartOpt>
                        <syntaxPart>  <lexPart>                            ;

  <optionsPartOpt> ::?  <optionsPart>                                      ;
  <optionsPart>    ::=  'Options:' <options> ';'                           ;
  <options>        ::+  {string} ','                                       ;

  <prologue>       ::=  <parseTableKind>  <transformsOpt>                  ;
  <parseTableKind> ::|  'SLR(1)'  'LALR(1)'  'LR(1)'                       ;
  <transformsOpt>  ::?  <transforms>                                       ;
  <transforms>     ::=  'Transformations:'  <transNames>  ';'              ;
  <transNames>     ::+  <transName> ','                                    ;
  <transName>      ::=  {trans:string}                                     ;

  <syntaxPart>     ::=  'Syntax:'  <prologue>  <productions>               ;

  <productions>    ::*  <production0>                                      ;
  <production0>    ::=  <production>  ';'                                  ;

  <production>     ::|  <alternation> <construction>
                        <lst0> <lst1> <optional>                           ;

  <alternation>    ::=  <lhs:nonterminal> '::|' <alternatives>             ;
  <construction>   ::=  <lhs:nonterminal> '::=' <symbols>                  ;
  <lst0>           ::=  <lhs:nonterminal> '::*' <elm:symbol> <sep:symOpt>  ;
  <lst1>           ::=  <lhs:nonterminal> '::+' <elm:symbol> <sep:symOpt>  ;
  <optional>       ::=  <lhs:nonterminal> '::?' <elm:symbol>               ;

  <alternatives>   ::+  <symbol>                                           ;
  <symbols>        ::*  <symbol>                                           ;
  <symOpt>         ::?  <symbol>                                           ;
  <symbol>         ::|  <nonterminal> <terminal> <literalTerm>             ;
  <literalTerm>    ::=  {string}                                           ;
  <terminal>        ::|  <terminalNT>    <terminalT>                          ;
  <nonterminal>     ::|  <nonterminalNT> <nonterminalT>                       ;

  <terminalNT>      ::=  '{' {kind:identifier} '}'                            ;
  <nonterminalNT>   ::=  '<' {kind:identifier} '>'                            ;
  <terminalT>       ::=  '{' {tag:identifier} ':' {kind:identifier} '}'      ;
  <nonterminalT>    ::=  '<' {tag:identifier} ':' {kind:identifier} '>'      ;

  <lexPart>         ::=  'Lex:'  <prologue>  <lexdefs>                        ;
  <lexdefs>         ::*  <lexdef>                                             ;

  <lexdef>          ::=  <regExpName> <binder> <regExp> ';'                   ;
  <binder>          ::|  <external> <internal>                                ;
  <external>        ::=  '->'                                                 ;
  <internal>        ::=  '='                                                  ;

  <regExp>          ::+  <term>  '|'                                          ;
  <term>            ::*  <factor>                                             ;
  <factor>          ::=  <base> <unaryOpOpt>                                  ;
  <base>            ::|  <regExpName> <literalExp> <parenExp>                 ;
  <parenExp>        ::=  '(' <regExp> ')'                                     ;
  <literalExp>      ::|  {string} {charSet}                                   ;
  <unaryOpOpt>      ::?  <unaryOp>                                            ;
  <unaryOp>         ::|  <closure0> <closure1> <lexoptional>                  ;
  <closure0>        ::=  '*'                                                  ;
  <closure1>        ::=  '+'                                                  ;
  <lexoptional>     ::=  '?'                                                  ;
  <regExpName>      ::=  <terminalNT>                                         ;

Lex:  SLR(1)
Transformations: 'elimEpsilons', 'elimSingletons', 'useCharClasses' ;

  {whitespace}          ->  {blank}+                                          ;
  {string}              ->  '\'' {stringStuff}* '\''                          ;
  {charSet}             ->  '[' {charSetStuff}* ']'                           ;
  {identifier}          ->  {letter} ({letterOrDigit} | '_')*                 ;

  {letterOrDigit}        =  {letter} | {digit}                                ;
  {letter}               =  [A-Za-z]                                          ;
  {digit}                =  [0-9]                                             ;
  {octalDigit}           =  [0-7]                                             ;
  {hexDigit}             =  [0-9a-fA-F]                                       ;
  {blank}                =  [ \t\n] | {comment}                               ;

  {comment}              =  '(*'  {commentStuff}                              ;

  {commentStuff}         =  {notStarOrParenBegin} {commentStuff}  |
                            '('  {commentStuffHasPB}              |
                            '*'  {commentStuffHasStar}                        ;

  {commentStuffHasPB}    =  {notStarOrParenBegin} {commentStuff}  |
                            '('  {commentStuffHasPB}              |
                            '*'  {commentStuff} {commentStuff}                ;

  {commentStuffHasStar} =  {notStarOrParenEnd} {commentStuff}     |
                            '*'  {commentStuffHasStar}            |
                            ')'                                               ;
  {stringStuff}          =  {notQuoteOrBS}  | {commonEscape}                 ;

  {charSetStuff}         =  {notEndBraceOrBS}  |
                            '\\]'  |  '\\-'  | {commonEscape}                ;

  {commonEscape}         =  '\\' {shortEscape} | '\\' {numericEscape}        ;
  {shortEscape}          =  [tbnfrva0\\'"?]                                  ;
  {numericEscape}        =  {hexEscape} | {decimalEscape} | {octalEscape}    ;
  {hexEscape}            =  'x' {hexDigit}   {hexDigit}                      ;
  {decimalEscape}        =  'd' {digit}      {digit}      {digit}            ;
  {octalEscape}          =  'o' {octalDigit} {octalDigit} {octalDigit}       ;

  {innocentChar}         =  [^*()\\\]']                                      ;
  {notQuoteOrBS}         =  {innocentChar} | [*()\]]                         ;
  {notEndBraceOrBS}      =  {innocentChar} | [*()']                          ;
  {notStarOrParenBegin} =  {innocentChar} | [)\\\]']                        ;
  {notStarOrParenEnd}   =  {innocentChar} | [(\\\]']                        ;

  (* This {innocentChar} thing is mostly just a manual refactoring to avoid
     too many productions in the lex grammar. The same thing is partially
     done automatically by specifying the 'useCharClasses' transformation,
     but doing it in the source has the advantage that the factoring is
     obvious to the reader and takes place as early as possible. *)
```

Ansi-C.grm

```
(* Sun-$Revision: 9.3 $ *)

(* Copyright 1992 Sun Microsystems, Inc. and Stanford University.
   See the LICENSE file for license information. *)

(* This structured grammar has been obtained by "structuring" the grammar in
   the book "The C Programming Language" by Kernighan/Ritchie (ANSI version).

   The terminal and nonterminal names used in this grammar are the same as
   in the book, except for the following abbreviations:

      decl  for  declaration
      exp   for  expression
      def   for  definition

   (and perhaps a few other abbreviations that I have forgotten right now).

   The major change in the grammar, apart from using structured productions,
   is a refactoring of the type_specifier/type_qualifier/storage_class
   related productions. The refactoring expresses that there can only be
   one type name in a declaration (e.g. the ANSI grammar derives float int x;).
   The refactoring also helps control ambiguities even though some
   occurrences of {identifier} have been replaced with
   <identifier_or_typedef_name>.

   The only other thing in this grammar that is not explained in the book
   is the typedef hack. So what is this about typedef?

   The C grammar contains an ambiguity (sigh!) which can be resolved by
   providing context dependent information. The context dependent information
   is obtained by splitting the token kind 'identifier' into two: 'identifier'
   and 'typedef_name'. Next a filter is inserted between the lexer and parser.
   The filter looks at the tokens produced by the lexer. The default behavior
   is to pass the token onto the parser unchanged. However, if the token has
   kind 'identifier', and the particular instance has a name that matches a
   name that has previously been typedeffed, the kind of the token is changed
   from 'identifier' to 'typedef_name'.

   The filter, which we call typedefFilter, contains a set of names that
   have been typedeffed. Since C allows typedeffed names to be redeclared
   as non-typedef things in inner scopes the typedef filter must be informed
   whenever a scope is entered or left.

   The bad side of all this is that the typedefFilter relies heavily on
   side effects that must take place during parsing. We perform these
   side effects as a part of initializing the nodes in the parse trees.
   This means that the order that certain nodes are constructed in
   is critical. Inlining of productions changes this order so we need
   a mechanism for controlling (preventing) inlining. This is the
   'dontInline:' transformation.

   Expect 4 s/r conflicts, 4 r/r conflicts.
   Since there are r/r conflicts in this grammar, it is important that
   productions are not arbitrarily moved around (the default resolution
   rule for reduce/reduce conflicts is to reduce with the production
   occurring earlier in the grammar).
*)
```

```
Name:     'Ansi-C'
Behavior: 'Ansi-C.behavior.self'
Options:  'typedefKludge';

Syntax: LALR(1)
Transformations: 'dontInline: start_scope',
                 'dontInline: end_scope',
                 'dontInline: core_decl',
                 'elimEpsilons',
                 'elimSingletons',
                 'flatten: real',     'inline: real',
                 'flatten: integral', 'inline: integral' ;

<translation_unit> ::+ <external_decl> ;

<external_decl> ::| <function_def> <decl> ;

<function_def> ::= <decl_specifiers_opt>
                   <declarator>
                   <decl_list_opt>
                   <compound_stmt> ;

<decl_list> ::+ <decl> ;

<decl>      ::= <core_decl> ';' ;
<core_decl> ::= <decl_specifiers> <init_declarator_list> ;

<decl_specifiers> ::| <no_sc_decl_specifiers>
                        <sc_decl_specifiers> ;

<no_sc_decl_specifiers> ::| <decl_specifiers1>
                            <decl_specifiers2>
                            <decl_specifiers3> ;

<sc_decl_specifiers>    ::| <decl_specifiers4>
                            <decl_specifiers5>
                            <decl_specifiers6>
                            <decl_specifiers7> ;

<decl_specifiers1> ::=                                  <type_specifier> ;
<decl_specifiers2> ::=                <type_qualifier>                   ;
<decl_specifiers3> ::=                <type_qualifier>  <type_specifier> ;
<decl_specifiers4> ::= <storage_class>                                   ;
<decl_specifiers5> ::= <storage_class>                  <type_specifier> ;
<decl_specifiers6> ::= <storage_class> <type_qualifier>                  ;
<decl_specifiers7> ::= <storage_class> <type_qualifier>  <type_specifier> ;

<storage_class>    ::| 'auto'
                       'register'
                       'static'
                       'extern'
                       'typedef' ;

<type_specifier>   ::| 'void'
                       <real>
                       <integral>
                       <struct_or_union_specifier>
                       <enum_specifier>
                       {typedef_name} ;
```

```
<real>             ::| <float_specifier>
                       <double_specifier>
                       <long_double_specifier> ;

(* Disallows "long long double". *)
<float_specifier>       ::= 'float'  ;
<double_specifier>      ::= 'double' ;
<long_double_specifier> ::= 'long' 'double' ;

<sign_size>  ::| <sign_size1> <sign_size2> ;
<sign_size1> ::= <sign> <size_opt> ;
<sign_size2> ::= <size> <sign_opt> ;
<size>       ::| <short_size> <long_size> ;
<sign>       ::| 'signed'  'unsigned' ;

<short_size> ::= 'short' ;  (* To avoid MI. *)
<short_type> ::= 'short' ;
<long_size>  ::= 'long'  ;
<long_type>  ::= 'long'  ;

(* Disallows "long long int". *)
<integral>  ::| <integral1> <integral2> <integral3> <integral4> <integral5> ;
<integral1> ::= 'int' ;
<integral2> ::= <sign_size> 'int' ;
<integral3> ::= <int_opt> <sign_size> ;
<integral4> ::= <sign_opt> <nonsizable_integral> ;
<integral5> ::= <nonsizable_integral> <sign_opt> ;

<nonsizable_integral> ::| 'char' <short_type> <long_type> ;

<type_qualifier> ::| 'const'  'volatile' ;

<struct_or_union_specifier> ::| <su_def> <su_decl> ;

<su_def>  ::= <struct_or_union>
              <identifier_or_typedef_name_opt>
              '{' <struct_decl_list> '}' ;

<identifier_or_typedef_name> ::| <identifier2> <typedef_name2> ;

<typedef_name2> ::= {name:typedef_name} ;  (* To avoid MI. *)
<identifier2>   ::= {name:identifier} ;

<su_decl>          ::= <struct_or_union> <identifier_or_typedef_name> ;
<struct_or_union>  ::| 'struct'  'union' ;
<struct_decl_list> ::+ <struct_decl> ;

<int_opt>                    ::? 'int'                    ;
<sign_opt>                   ::? <sign>                   ;
<size_opt>                   ::? <size>                   ;
<decl_specifiers_opt>        ::? <decl_specifiers>        ;
<parameter_type_list_opt>    ::? <parameter_type_list>    ;
<declarator_opt>             ::? <declarator>             ;
<constant_exp_opt>           ::? <constant_exp>           ;
<exp_opt>                    ::? <exp>                    ;
<ellipsis_opt>               ::? <ellipsis>               ;
<abstract_declarator_opt>    ::? <abstract_declarator>    ;
<comma_opt>                  ::? ','                      ;
```

```
<pointer_opt>                      ::? <pointer>                      ;
<direct_abstract_declarator_opt> ::? <direct_abstract_declarator> ;
<decl_list_opt>                    ::? <decl_list>                    ;
<enum_initializer_opt>             ::? <enum_initializer>             ;
<identifier_or_typedef_name_opt> ::? <identifier_or_typedef_name> ;

<init_declarator_list> ::* <init_declarator> ',' ;

<init_declarator> ::| <i_declarator> <declarator_and_initializer> ;

<i_declarator> ::= <declarator> ;  (* To avoid MI in grammar. *)

<declarator_and_initializer> ::=  <declarator> '=' <initializer> ;

<struct_decl> ::= <no_sc_decl_specifiers> <struct_declarator_list> ';' ;

<struct_declarator_list> ::+ <struct_declarator> ',' ;

<struct_declarator> ::| <s_declarator> <field> ;

<s_declarator> ::= <declarator> ; (* To avoid MI in grammar. *)

<field> ::= <declarator_opt> ':' <constant_exp> ;

<enum_specifier>   ::| <enum_def> <enum_decl> ;

<enum_decl>        ::= 'enum' <identifier_or_typedef_name> ;
<enum_def>         ::= 'enum' <identifier_or_typedef_name_opt>
                       '{' <enumerator_list> <comma_opt> '}' ;
(* Is <identifier_or_typedef_name> really the right thing to use above,
   or should it just be {identifier}? gcc can't really make up it's mind. *)

<enumerator_list>  ::+ <enumerator> ',' ;
<enumerator>       ::= {identifier} <enum_initializer_opt> ;
<enum_initializer> ::= '=' <constant_exp> ;

<declarator>       ::= <pointer_opt> <direct_declarator> ;

<direct_declarator> ::= <direct_declarator_part1> <direct_declarator_part2> ;

<direct_declarator_part1> ::| <declarator_identifier> <paren_declarator> ;

<declarator_identifier> ::= <id:identifier_or_typedef_name> ;

<paren_declarator> ::= '(' <declarator> ')' ;

<direct_declarator_part2> ::* <a_or_f_declarator> ;

<a_or_f_declarator> ::| <array_declarator>
                        <fct_argtype_declarator>
                        <fct_argname_declarator>  ;

<array_declarator>       ::= '[' <constant_exp_opt>    ']' ;
<fct_argtype_declarator> ::= '(' <parameter_type_list> ')' ;
<fct_argname_declarator> ::= '(' <identifier_list>     ')' ;

<pointer> ::+ <star_type_qualifier_list> ;

<star_type_qualifier_list> ::= '*' <type_qualifier_list> ;
```

```
<type_qualifier_list> ::* <type_qualifier> ;

<parameter_type_list> ::= <parameter_list>  <ellipsis_opt> ;

<ellipsis> ::= ','  '...' ;

<parameter_list> ::+ <parameter_decl> ',' ;

<parameter_decl> ::= <decl_specifiers> <abstract_or_concrete_pd> ;

<abstract_or_concrete_pd> ::| <abstract_pd> <concrete_pd> ;

<concrete_pd> ::= <declarator> ;

<abstract_pd> ::= <abstract_declarator_opt> ;

<identifier_list> ::* {identifier} ',' ;

<initializer> ::| <simple_initializer> <compound_initializer> ;

<simple_initializer> ::= <assignment_exp> ;

<compound_initializer> ::= '{' <initializer_list> <comma_opt> '}' ;

<initializer_list> ::+ <initializer> ',' ;

<type_name> ::= <no_sc_decl_specifiers> <abstract_declarator_opt> ;

<abstract_declarator> ::| <pointer_only>
                          <pointer_opt_direct_abstract_declarator> ;

<pointer_only> ::= <pointer> ;

<pointer_opt_direct_abstract_declarator> ::= <pointer_opt>
                                             <direct_abstract_declarator> ;

<direct_abstract_declarator> ::| <paren_abstract_declarator>
                                 <other_abstract_declarator> ;

<paren_abstract_declarator> ::= '(' <abstract_declarator> ')' ;

<other_abstract_declarator> ::= <direct_abstract_declarator_opt>
                                <array_or_fct_ad> ;

<array_or_fct_ad> ::| <array_ad> <fct_ad> ;

<array_ad> ::= '[' <constant_exp_opt>        ']' ;
<fct_ad>   ::= '(' <parameter_type_list_opt> ')' ;

(* <typedef_name> ::= {identifier} ;    (* THE BAD ONE! *) *)

<stmt> ::| <labeled_stmt>
           <exp_stmt>
           <compound_stmt>
           <selection_stmt>
           <iteration_stmt>
           <jump_stmt>    ;
```

```
<labeled_stmt> ::= <label> ':' <stmt> ;

<label> ::| <ident_label> <case_label> ;

<ident_label> ::= <identifier_or_typedef_name> ;

<case_label> ::| <case_exp_label>  'default' ;

<case_exp_label> ::= 'case' <constant_exp> ;

<exp_stmt> ::= <exp_opt> ';' ;

<compound_stmt> ::= <start_scope> <decl_list_opt> <stmt_list> <end_scope> ;

<start_scope> ::= '{' ;    (* Necessary for typedef. *)
<end_scope>   ::= '}' ;    (* Necessary for typedef. *)

<stmt_list> ::* <stmt> ;

<selection_stmt> ::| <if_stmt> <if_else_stmt> <switch_stmt> ;

<if_stmt>      ::= 'if'  '(' <exp> ')' <thenpart:stmt> ;
<if_else_stmt> ::= 'if'  '(' <exp> ')' <thenpart:stmt> 'else' <elsepart:stmt> ;
<switch_stmt>  ::= 'switch' '(' <exp> ')' <stmt> ;

<iteration_stmt> ::| <while_stmt> <do_stmt> <for_stmt> ;
<while_stmt>     ::= 'while' '('  <exp> ')' <stmt> ;
<do_stmt>        ::= 'do' <stmt> 'while' '('  <exp> ')' ';' ;

<for_stmt>   ::= 'for' '(' <init:exp_opt> ';'
                           <step:exp_opt> ';'
                           <test:exp_opt>  ')' <stmt> ;

<jump_stmt>  ::| <goto_stmt> <continue_stmt> <break_stmt> <return_stmt> ;

<goto_stmt>     ::= 'goto' <identifier_or_typedef_name> ';' ;
<continue_stmt> ::= 'continue'                          ';' ;
<break_stmt>    ::= 'break'                             ';' ;
<return_stmt>   ::= 'return' <exp_opt>                  ';' ;

<exp> ::+ <assignment_exp> ',' ;

<assignment_exp> ::= <assignment_list> <conditional_exp> ;

<assignment_list> ::* <assign> ;

<assign> ::= <unary_exp> <assignment_operator> ;

<assignment_operator> ::| '='
                          '*='
                          '/='
                          '%='
                          '+='
                          '-='
                          '<<='
                          '>>='
                          '&='
                          '^='
                          '|=' ;
```

```
<conditional_exp> ::= <cond_stuff_list> <logical_OR_exp> ;

<cond_stuff_list> ::* <cond_stuff> ;

<cond_stuff> ::= <logical_OR_exp> '?' <exp> ':' ;

<constant_exp> ::= <conditional_exp> ;

<logical_OR_exp>     ::+ <logical_AND_exp>    '||' ;
<logical_AND_exp>    ::+ <inclusive_OR_exp>   '&&' ;
<inclusive_OR_exp>   ::+ <exclusive_OR_exp>   '|'  ;
<exclusive_OR_exp>   ::+ <AND_exp>            '^'  ;
<AND_exp>            ::+ <equality_exp>       '&'  ;
<equality_exp>       ::+ <relational_exp>     <eq_op>    ;
<relational_exp>     ::+ <shift_exp>          <rel_op>   ;
<shift_exp>          ::+ <additive_exp>       <shift_op> ;
<additive_exp>       ::+ <multiplicative_exp> <add_op>   ;
<multiplicative_exp> ::+ <cast_exp>           <mul_op>   ;

<eq_op>    ::| '=='  '!='              ;
<rel_op>   ::| '<'   '>'   '<='  '>=' ;
<shift_op> ::| '<<'  '>>'              ;
<add_op>   ::| '+'   '-'               ;
<mul_op>   ::| '*'   '/'   '%'         ;

<cast_exp>  ::= <cast_list> <unary_exp>  ;
<cast_list> ::* <cast>                   ;
<cast>      ::= '(' <type_name> ')'      ;

(* Could perhaps be structured more. But beware of possible conflict
   introduced by two uses of sizeof. *)
<unary_exp> ::| <postfix_exp>
                <preinc_unary_exp>
                <predec_unary_exp>
                <sizeof_unary_exp>
                <sizeof_type_name>
                <unary_operator_cast_exp> ;

<preinc_unary_exp>        ::= '++' <unary_exp> ;
<predec_unary_exp>        ::= '--' <unary_exp> ;
<sizeof_unary_exp>        ::= 'sizeof' <unary_exp> ;
<sizeof_type_name>        ::= 'sizeof' '(' <type_name> ')' ;
<unary_operator_cast_exp> ::= <unary_operator> <cast_exp> ;

<unary_operator> ::| <deref>
                     <addr_of>
                     <unary_plus>
                     <unary_minus>
                     <bit_complement>
                     <negation>   ;

<deref>          ::=  '*' ;
<addr_of>        ::=  '&' ;
<unary_plus>     ::=  '+' ;
<unary_minus>    ::=  '-' ;
<bit_complement> ::=  '~' ;
<negation>       ::=  '!' ;
```

```
<postfix_exp> ::= <primary_exp> <postfix_op_list> ;
<postfix_op_list> ::* <postfix_op> ;

<postfix_op> ::| <indexing> <fct_call> <dot> <arrow> <postinc> <postdec> ;

<indexing> ::= '[' <exp> ']' ;
<fct_call> ::= '(' <argument_exp_list> ')' ;
<dot>      ::= '.'  <identifier_or_typedef_name> ;
<arrow>    ::= '->' <identifier_or_typedef_name> ;
<postinc>  ::= '++' ;
<postdec>  ::= '--' ;

<primary_exp> ::| <identifier_exp> <constant> <string_list> <paren_exp> ;

<string_list>    ::+ {string} ;
<identifier_exp> ::= {identifier} ;
<paren_exp>      ::= '(' <exp> ')' ;

<argument_exp_list> ::* <assignment_exp> ',' ;

<constant> ::| {integer_constant} {character_constant} {floating_constant} ;

(******************************** Lex part ***********************************)

Lex:  SLR(1)
Transformations: 'elimEpsilons',
                 'elimSingletons',
                 'useCharClasses' ;

(* output tokens *)

  {identifier}         -> ('_' | {letter}) ({letter} | {digit} | '_')* ;
  {string}             -> '”'  {stringChar}*  '”'  ;
  {character_constant} -> '\'’ {charChar}*    '\'’ ;
  {integer_constant}   -> {basicInteger}   {intSuffix} ;
  {floating_constant}  -> {basicFloat}   {floatSuffix} ;
  {whitespace}         -> [ \t\v\f\n]+ ;

(* string productions *)

  {stringChar}          = '\'’ | {stringOrCharChar} ;
  {charChar}            = '”'  | {stringOrCharChar} ;
  {stringOrCharChar}    =  [^\\"'] | '\\' {escape} ;

  {escape}          =  {nlEsc} | {tabEsc} | {vtEsc}  | {bspEsc} | {crEsc}
                    |  {ffEsc} | {belEsc} | {bslEsc} | {qmEsc}  | {apEsc}
                    |  {dqEsc} | {octEsc} | 'x' {hexEsc}
                    |  {redundantEsc} ;
                       (* Apparently Ansi C is inconsistent: it disallows 'X'
                          as a hex specifier in hex escapes, but allows it in
                          hex integers. *)

  {bslEsc}          =  '\\';
  {qmEsc}           =  '?' ;
  {apEsc}           =  '\'’;
  {dqEsc}           =  '”' ;
  {belEsc}          =  'a' ;
  {bspEsc}          =  'b' ;
  {ffEsc}           =  'f' ;
  {nlEsc}           =  'n' ;
  {crEsc}           =  'r' ;
  {tabEsc}          =  't' ;
  {vtEsc}           =  'v' ;

  {octEsc}          =  {octalDigit} |
                       {octalDigit} {octalDigit} |
                       {octalDigit} {octalDigit} {octalDigit} ;

  {hexEsc}          =  {hexDigit}   |
                       {hexDigit}   {hexDigit}   |
                       {hexDigit}   {hexDigit}   {hexDigit} ;
                       (* Ansi C allows three hex digits. *)

  {redundantEsc}    =  [^\\?'"0-7abfnrtvx] ;

(* integers *)

  {basicInteger}    =  {octalInteger} | {hexInteger} | {decimalInteger} ;
  {octalInteger}    =  '0'      {octalDigit}* ;
  {hexInteger}      =  '0' [xX] {hexDigit}+   ;
  {decimalInteger}  =  [1-9]    {digit}*      ;
  {intSuffix}       =  {signedInt}   | {signedLong}
                    |  {unsignedInt} | {unsignedLong} ;
  {signedInt}       = ;
  {unsignedInt}     = [uU];
  {signedLong}      = [lL];
  {unsignedLong}    = 'ul' | 'lu' | 'Ul' | 'lU' | 'uL' | 'Lu' | 'UL' | 'LU' ;

(* floats *)

  {basicFloat}      =  {floatWithInt} | {floatNoInt} ;

  {floatWithInt}    =  {integralPart} ({fract0} | {exp} | {fract0} {exp}) ;
  {floatNoInt}      =  {fract1} {exp}? ;
  {integralPart}    =  '0' | {decimalInteger} ;
                       (* Note: defining {integralPart} to be {digit}+
                          results in reduce/reduce conflicts since the
                          parser is forced to decide early on whether it
                          is seeing an integer or a float. The present
                          solution is not quite right:  03.4 is not
                          recognized as a float. *)

  {fract0}          =  '.' {digit}* ;
  {fract1}          =  '.' {digit}+ ;
  {exp}             =  [eE] [+-]? {digit}+ ;

  {floatSuffix}     =  {singleFloat} | {doubleFloat} | {longFloat} ;
  {singleFloat}     =  [fF] ;
  {doubleFloat}     =     ;
  {longFloat}       =  [lL] ;

(* character classes *)

  {octalDigit}      =  [0-7] ;
  {digit}           =  [0-9] ;
  {hexDigit}        =  [0-9a-fA-F] ;
  {letter}          =  [a-zA-Z] ;
```

Ansi-C.behavior.self

```
"Sun-$Revision: 8.4 $"

"Copyright 1992 Sun Microsystems, Inc. and Stanford University.
 See the LICENSE file for license information."


( |
  _ parent* = traits oddball.

  ^ core__decl = ( |
      ^ isTypedef = (
          "Return true if this declaration is a typedef."
          decl__specifiers is_sc__decl__specifiers &&
          [decl__specifiers storage__class token source = 'typedef'].
        ).

      ^ declNames = (
          "Return list of names that are being declared."
          init__declarator__list declNames1.
        ).

      ^ initialize_node: stack = (
          | tdf. |
          false ifTrue: [
              (isTypedef ifTrue: 'typedef' False: 'non-typedef') print.
              ' decl of ' print.
              declNames printLine.
          ].
          tdf: (typedefFilter: stack).
          tdf enterNames: declNames Typedeffed: isTypedef.
          self.
        ).
    | ).

  ^ storage__class__specifier = ( |
      ^ isTypedef1 = ( 'typedef' =  token source. ).
    | ).

  ^ init__declarator__list = ( |
      ^ declNames1 = (
          "Return list of all names declared by this declarator_list."
          | names <- list. |
          names: names copy.
          elements do: [|:iDeclarator| names add: iDeclarator declName].
          names.
        ).
    | ).

  ^ init__declarator = ( |
      ^ declName = (
          "Return name that is being declared."
          declarator declName1.
        ).
    | ).

  ^ declarator = ( |
      ^ declName1 = (
          direct__declarator direct__declarator__part1 declName2.
        ).
    | ).

  ^ paren__declarator = ( |
      ^ declName2 = ( declarator declName1. ).
    | ).

  ^ type__specifier__or__type__qualifier = ( |
      ^ isTypedef1 = false.
    | ).

  ^ declarator__identifier = ( |
      ^ declName2 = ( id name token source. ).
    | ).

  ^ start__scope = ( |
      ^ initialize_node: stack = (
          | tdf. |
          tdf: (typedefFilter: stack).
          tdf enterScope.
          self.
        ).
    | ).

  ^ end__scope = ( |
      ^ initialize_node: stack = (
          | tdf. |
          tdf: (typedefFilter: stack).
          tdf exitScope.
          self.
        ).
    | ).


"shared_behavior describes slots to be added to the common ancestor
     (traits) of all tree nodes. This is the 'stGramNode traitsSkeleton'
     in the Ansi-C 'stGrammar' object.
     shared_behavior_proto describes slots to be added to
     'stGramNode protoSkeleton'.
     These slots will be copied down into all concrete tree nodes."

  ^ shared_behavior = ( |
      _ typedefFilter: stack = ( stack parser prevParser. ).

      _ traceInitialization = (
          false ifTrue: ['initializing ' print. printLine].
          self.
        ).
    | ).

  ^ shared_behavior_proto = ( |
    | ).
| )
```

About the author

Ole Agesen received a Master's degree in Computer Science (Kandidatgrad) from Aarhus University, Denmark in 1990 for work done within the BETA/Mjølner project. He is currently a doctoral candidate at Stanford University and a member of the Self group at SMLI. His research interests include object-oriented programming, programming environments, and static analysis of programs.