User Defined Syntax

Annika Aasa

DEPARTMENT OF COMPUTER SCIENCES

1992

User Defined Syntax

Annika Aasa

Department of Computer Sciences
Chalmers University of Technology
and University of Göteborg

1992

A Dissertation for the Ph.D. Degree in Computing Science
at Chalmers University of Technology

Department of Computer Sciences
S-412 96 Göteborg, Sweden

ISBN 91-7032-738-6
Göteborg 1992

This thesis is based on the work contained in the following published papers:

II Annika Aasa, Kent Petersson and Dan Synek, "Concrete Syntax for Data Objects in Functional Languages". In Proceedings of the 1988 ACM conference on LISP and Functional Programming, Snowbird, Utah, July 25-27, 1988.

IV This paper is an extended version of
Annika Aasa, "Precedences in Specifications and Implementations of Programming Languages". In Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming, Passau, Germany, August 26-28, 1991, Lecture Notes in Computer Science, Springer Verlag.

and the unpublished papers:

III Annika Aasa, "Conctypes: Extensions, Algorithms and Implementation"

V Annika Aasa, "Precedences for Context-free Grammars"

VI Annika Aasa, "A Recursive Descent Parser for User Defined Distfix Operators". This paper is one part of the licentiate thesis, May 1989, Department of Computer Sciences, Chalmers University of Technology, S-412 96 Göteborg.

User Defined Syntax

Annika Aasa
Department of Computer Sciences
Chalmers University of Technology
S-412 96 Göteborg, Sweden
E-mail: annika@cs.chalmers.se

Abstract

This thesis describes two examples of user defined syntax. The first, and most thoroughly investigated, is a new datatype construction, *the conctype*, the elements of which have a very flexible syntax. An embedded language can easily be introduced into a programming language using conctypes and computations are easily expressed using the concrete syntax and a special pattern matching form. The second example is user defined *distfix operators* which give a user possibility to extend the syntax for expressions in a programming language. We describe both a user's view and the implementation of these two examples.

In both cases, context-free grammars serve as a basis for the definition of the new syntax. A problem that is investigated is how to disambiguate grammars with precedences. To see how this should be done we investigate which language a grammar together with precedence rules defines. For a subclass of context-free grammars we give a predicate that defines the *precedence correct* syntax trees according to some precedence rules. We also give an algorithm that transforms such a grammar to an ordinary unambiguous context-free grammar and prove the correctness of the algorithm. We use the algorithm in our implementation of distfix operators. For more general grammars, we isolate one kind of ambiguity which is suitable to resolve with precedence rules. We define the generated language for such a grammar by an attribute grammar. This approach of resolving ambiguity is used in the implementation of conctypes.

Keywords and Phrases: User Defined Syntax, Concrete Syntax, Parsing, Grammars, Ambiguity, Precedence, Distfix Operators, Inductive Datatypes, Pattern Matching, Earley's algorithm, Recursive Descent.

Contents

Acknowledgements

First, I would like to thank my advisor Kent Petersson. I appreciate his patience with silly questions and his encouragement when I not believed in myself as a researcher. Kent has spent a lot of time on discussions and on reading and commenting on different versions of my papers. The result was often valuable ideas and good comments. He is also a co-author of the first paper on conctypes and without him the paper would not have been written, at least not at that time.

Dan Synek is the other co-author of the first paper on conctypes. It was he who did the first implementation of the parsing algorithm used for conctypes. Unfortunately, he has left the department and I miss his ideas, comments and theories among other things.

It was Kent and Dan who invited me into the syntax group and first suggested me to implement user defined distfix operators. Since I chose the parsing method recursive descent, that suggestion caused many interesting problems. Later, we worked together in the conctype project which originally was their idea.

The privilege to have Lennart Augustsson in the same corridor cannot be too highly prized when doing implementations in LML. I thank him for the answers to all my questions and for his interest in and comments about conctypes.

Mikael Rittri have suggested improvements in the proof of the precedence removing algorithm $\mathcal{M}$. I am also grateful for his careful reading and valuable comments on different versions of some of the papers contained in this thesis.

During the last hectic half year and especially during the last summer, Urban Boquist has, in a wonderful way, helped me to reduce the increasing feeling of panic. I also thank him for reading the thesis and for all comments.

Among others, I would like to thank Bengt Nordström for encouragement and for always believing that I would finish this thesis, Staffan Truvé for his willingness to help and for the suggestion to write a program that produces LaTeX-code for parse trees, Magnus Carlsson for discussions about type checking, Lena Magnusson for moral support and for clarifying some things, Sören Holmström for being my first year advisor, and Thomas Johnsson, Niklas Röjemo, Thomas Hallgren and Jan Smith for reading and commenting on the papers.

I also wish to thank all other members of the programming methodology group, and all persons who keep the department and its computers going. Special thanks are due to Christer Carlsson and Marie Larsson. I am also grateful to my family and all other friends.

Finally, I hope that a Ph.D. degree is worth all tears it has and will cost me. Thanks to everyone who have helped me to decrease the number of them.

Part I

Introduction

User Defined Syntax
an Introduction

Annika Aasa

The syntax of a programming language is the rules for how programs are allowed to be written. In most programming languages these rules are fixed. In this thesis, we will investigate different approaches of letting the programmer extend the syntax of a programming language. We will call this *user defined syntax*.

One reason for wanting user defined syntax in a programming language is the simple fact that the syntax cannot possibly include every notation a user may want to use. It is therefore desirable to give a user the possibility to extend the syntax and introduce new notation.

In many problem areas there is an established notation which would be good to use also in programs. We for example often use $f[x \mapsto v]$ to denote a function f updated with a value v for the argument x. When writing programs and if the programming language has a facility to introduce new notation it could be possible to use something like

```
f [ x -> v ]
```

instead of a function taking three arguments, for example `update f x v`.

User defined syntax can save work for programmers. A frequent programming task is to write *parsers*. A parser takes a string as input and if the string belongs to a certain set of strings, a *language*, the parser outputs another representation of the string. There exist tools which automatically generate a parser given a language description. Since these tools for different reasons can be awkward to use, parsers are anyhow often written by hand, especially for relatively simple languages. If the tools were integrated with the programming language they would be easier and more natural to use. One example of this is if the restricted syntax of the elements of user defined *datatypes* were more flexible, then sometimes it would not be necessary to write parsers that translate strings into the datatypes. In most programming languages, elements of user defined datatypes have a very restricted syntax.

User defined syntax can enhance the readability of programs, which certainly is desirable. A simple example of a feature which enhances the readability is user defined infix operators. An expression with repeated use of infix operators is usually more readable than the corresponding expression using functions with prefix notation. Many programming languages allow user defined infix operators, for example Algol 68, SML, LML, Haskell and Prolog .

However, there are also objections to allowing a user to extend the syntax. A user may introduce too much new syntax and in that way develop a new dialect which no one else understands. Of course, it is not possible to forbid a user to introduce poor syntax, but

the purpose of user defined syntax is not to give a user possibilities to change the whole language but only to do small extensions.

There are different kinds of user defined syntax. One kind is to have good possibilities to define an embedded language within the programming language. This is particularly useful when writing programs that manipulate languages, for example interpreters and compilers. Two parts of this thesis describe such a feature. A new datatype construction, the elements of which can have a very flexible syntax, is added to the functional programming language Lazy ML (LML) [AJ87, AJ89]. This new construction is called *concrete datatypes* [APS88], or *conctypes* for short.

Another kind of user defined syntax is to give the user possibilities to extend the syntax of the programming language itself. An example that we have already mentioned is user defined infix operators, which many programming languages allow. A generalization that we will consider here is user defined *distfix operators* [Jon86]. A distfix operator is an operator which is distributed among its operands, for example the `if-then-else` construction. Few programming languages allow user defined distfix operators. We have done an experimental implementation of them.

A designer and implementor of user defined syntax has several decisions to take and problems to solve. What kind of objects should the user be able to define new syntax for? Which restrictions should there be on the new syntax? The difficulty of implementing user defined syntax features depends on the answers to these questions.

There must, of course, be possibilities for the user to *define* the new syntax. It is preferable that the definition of the new syntax is as easy and natural as possible. Furthermore, to define the new syntax and to use it, must not require too much knowledge about parsing and formal language theory. Otherwise, ordinary programmers without this knowledge will not be able to use the feature.

However, the definition of the new syntax must also be formal enough to be automatically analyzed. The definition must contain enough information about the new syntax. In particular, the definition must not be ambiguous. Otherwise, the user and the program which analyzes the definition may disagree about the meaning of some sentences. Thus, a language definition formalism which is natural, readable and understandable by ordinary programmers and can be used also to automatically generate a parser would be desirable. The problems of finding such language definitions are investigated in this thesis.

Conctypes

Two parts of this thesis describe a user defined syntax feature, the *concrete datatypes* or *conctypes* [APS88]. Elements of datatypes in most programming languages can usually only be written in a very restricted form. When such elements become large they soon become unreadable, and the user is often forced to write a parser which transforms a string to an element of the datatype. The strings can be seen as a concrete representation of the abstract syntax that the datatype defines. Using conctypes, the problem that the elements become unreadable does not arise, since the programmer defines their concrete syntax.

Let us compare two datatype definitions, an ordinary one and a conctype, and their elements. Both define a small functional language. All examples are written in LML.

First the ordinary version:

```
type expr = Id     String
          + Num    Int
          + App    expr expr
          + Mul    expr expr
          + Div    expr expr
          + Add    expr expr
          + Sub    expr expr
          + Eq     expr expr
          + If     expr expr expr
          + Let    String expr expr
          + Lam    String expr
```

An element of this datatype is:

```
Let "fac"
    (Lam "x" (If (Eq (Id "x") (Num 0))
                 (Num 1)
                 (Mul (Id "x") (App (Id "fac") (Sub (Id "x") (Num 1))))))
    (App (Id "fac") (Add (Mul (Num 3) (Num 4)) (Num 2)))
```

A conctype for the same language can be defined as:

```
conctype expr =              [|<Id>|]
              +              [|<Number>|]
              +              [||(<expr>)||]
              + < leftassoc  [|<expr> <expr>|]
              + < leftassoc  [|<expr>*<expr>|]
              + =            [|<expr>/<expr>|]
              + < leftassoc  [|<expr>+<expr>|]
              + =            [|<expr>-<expr>|]
              + < nonassoc   [|<expr>=<expr>|]
              + <            [|if <expr> then <expr> else <expr>|]
              + =            [|let <Id>=<expr> in <expr>|]
              + =            [|\\<Id>.<expr>|]
```

The elements of this conctype are much more readable:

```
[|let fac=\\x.if x=0 then 1 else x*fac (x-1) in fac (3*4+2)|]
```

We assume that `Id` and `Number` are two already defined conctypes. In the future, such conctypes will be supplied as predefined conctypes. The enclosing of the elements in brackets makes it an embedded language and it is possible to use reserved words of the programming language in the elements. The symbols `<` and `=` and the reserved words `leftassoc` and `nonassoc` are precedence information. Part V of this thesis describes how this information is used to resolve ambiguity. Precedence information is not needed for the ordinary datatype since its elements can only be written in prefix format.

It is easy to define and use a conctype. The notation is very close to the well-known BNF-notation [Bac60, Nau63, Knu64]. Thus, for most programmers it would not be any

problem to understand and use the notation. A conctype definition is not much bigger than the definition of the corresponding ordinary datatype. Thus, better syntax of the elements can easily be obtained. The extra writing is not much compared to when a parser is needed together with the ordinary datatype. A conctype corresponds to a context-free grammar where the nonterminals correspond to types. The types can either be conctypes or ordinary types. It is possible to also have polymorphic conctypes.

Computations over conctype elements are easily defined. In functional languages, computations are preferably defined using pattern matching. A special pattern matching form is used when defining computations over conctype elements. The *antiquotation symbol*, `^`, is used to have LML-variables in the patterns. We will define a function which computes the value of an expression in the conctype `expr`. We let the values be elements of an ordinary datatype:

```
type Value = FN (Value -> Value)
           + B  Bool
           + I  Int
```

The function `eval` evaluates an expression in an environment which give values to the free identifiers in the expression. We represent the environment as a function.

```
eval :: expr → (Id → Value) → Value

rec eval [|^id|] w                  = w id
||  eval [|^n|] w                   = I (toInt n)
||  eval [|^(e1) ^e2|] w            = let FN f = eval e1 w
                                      in
                                         f (eval e2 w)
||  eval [|^e1*^e2|] w              =  (eval e1 w) ** (eval e2 w)
||  eval [|^e1/^e2|] w              =  (eval e1 w) // (eval e2 w)
||  eval [|^e1+^e2|] w              =  (eval e1 w) ++ (eval e2 w)
||  eval [|^e1-^e2|] w              =  (eval e1 w) sub (eval e2 w)
||  eval [|^e1=^e2|] w              =  B (eval e1 w = eval e2 w)
||  eval [|if ^(e1) then ^(e2) else ^e3|] w
                                    = case eval e1 w in
                                         B true  : eval e2 w
                                      || B false : eval e3 w
                                      end
||  eval [|let ^id=^(e1) in ^e2|] w =
                          let rec w1 = update id (eval e1 w1) w
                          in eval e2 w1
||  eval [|\\^id.^e|] w             = FN (\v. eval e (update id v w))
```

We assume that the following functions are already defined[1] :

```
**, //, ++, sub :: (Value × Value) → Value
toInt           :: Number → Int
update          :: Id → Value → (Id → Value) → (Id → Value)
```

A conctype for a nontrivial subset of Pascal together with an interpreter and a bigger example of a conctype element is given in an appendix to part III of this thesis.

[1] In the future the translation with `toInt` will be implicit and not necessary in the program.

Grammars, Languages, Ambiguity and Precedences

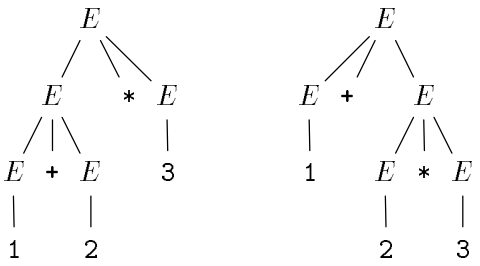
An important question when designing user defined syntax is how the new syntax should be specified by the user. As mentioned before, it is desirable that the definition of the new syntax is intuitive, readable and easy to use. The definition must of course also contain enough information about the new syntax and not be ambiguous.

Context-free grammars is a well-known formalism for defining languages. It is also possible to automatically generate a parser from a context-free grammar. These facts make context-free grammars useful in language definitions of user defined syntax. Our two examples of user defined syntax, the conctypes and the distfix operators, both have context-free grammars as a basis for the definition of the new syntax.

A problem with context-free grammars that we will focus on here is the ambiguity problem. A grammar is ambiguous if it generates a sentence which has more than one parse tree. A simple example is the following:

```
E  ::=  int
     |  id
     |  ( E )
     |  E * E
     |  E + E
```

The sentence `1+2*3`, for example, has two different parse trees:



Ambiguous grammars are often undesirable since different representations can have different meanings and we usually want to have an exact meaning of each sentence. The values, interpreted as usual, of the two parse trees above differs for example.

For user defined syntax definitions based on context-free grammars, there are different solutions to the problem with ambiguous sentences:

- We can have definitions based only on unambiguous context-free grammars. Unfortunately, unambiguous context-free grammars are often not the most natural ones. An unambiguous grammar generating the same language as the ambiguous one above is:

```
E  ::=  E + T
     |  T
T  ::=  T * F
     |  F
F  ::=  int
     |  id
     |  ( E )
```

We have written the grammar such that the binding power of the operators is the usual one. We will show in part IV that unambiguous grammars sometimes can be surprisingly complicated. As an example, a language which can be generated by 6 productions by an ambiguous grammar require 26 productions in an unambiguous grammar (see section 4.2 in part IV). The number of required productions depend on the desired structure of the parse trees.

There are also other reasons why unambiguous context-free grammars can be unsuitable to use. A conctype based on an unambiguous grammar can make computations over the elements more complicated. This is explained in part III. The definition of distfix operators, described in part VI, makes use of the fact that a language of distfix operator expressions can be defined by a grammar using one single nonterminal. Therefore, the definition of a new distfix operator can be done in a simple way. The underlying grammar is ambiguous and it is hard to see how such a simple definition could be possible if the underlying grammar was unambiguous.

- We can allow the definition to be ambiguous but require that all sentences are unambiguous. Thus, an ambiguous sentence is considered incorrect. This solution implies that many sentences will be hard to read. The sentence `1+2*3*4+5+6` generated by the ambiguous grammar on page 7 is ambiguous and would not be allowed in such case. Instead, a fully parenthesized sentence `((1+((2*3)*4))+5)+6` must be used. For other grammars there may not even exist a corresponding unambiguous alternative. As an example, remove the production with the parentheses from our example grammar.

- Since some languages are more easily represented by ambiguous grammars, a common solution is to use an ambiguous grammar augmented with some extra information telling what structure the ambiguous sentences have. Our ambiguous grammar on page 7 for example, is often used together with the extra information saying that the productions E ::= E `+` E and E ::= E `*` E have different *precedences*.

  A benefit with this solution is that a grammar can be both concise and unambiguous. One drawback is that it is not always clear which language such a grammar generates.

We have chosen the last solution to resolve ambiguities for our user defined syntax, both for conctypes and for distfix operators. Two parts of this thesis investigate different aspects of this method.

We must decide what kind of ambiguities to resolve, and if disambiguation is given by the user or by fixed rules. Here, we will only consider disambiguating rules that are given by a user. The ambiguities we consider are of the kind where there really is a choice between different possible representations, and not only one most natural.

Other questions are: how should the disambiguating information be given and the most important question: which language is generated? In part IV and V, solutions to these questions are suggested. The suggestion in part V is also used to disambiguate conctypes.

It is not obvious how to define which language a grammar together with disambiguating information generates. It is common to do the definition in terms of a parsing method. Apart from the methodology objection that a language definition should not involve a method to recognize the language, the dependency on the parsing method has at least two more disadvantages. First, if the disambiguation depends on the parsing method, the

understanding of how disambiguation works clearly requires knowledge of the method. Those who do not have the knowledge will be disqualified to understand the language definition. Second, it can be hard for an implementor to change the parsing method and preserve the disambiguation properties.

In this thesis we will present two other ways to define which language a grammar together with disambiguating rules generates. First, in part IV, we consider only a subclass of the context-free grammars. The generated language is defined in terms of the context-free grammar and a predicate. In part V more general grammars are considered. Here the attribute grammar formalism [Knu68a, Knu68b] is used to define which language a context-free grammar together with some disambiguating information generates. However, the two definitions have some ideas in common. In both cases, examples and theorems motivate that the definitions are reasonable.

Having a grammar together with disambiguation information and a definition of which language such a grammar generates, a problem is to construct a parser for that language. There are different ways to solve this problem:

- We can transform the grammar to an ordinary unambiguous context-free grammar and then use an ordinary parsing method. This solution is usually used if the chosen parsing method is recursive descent [DM81]. However, such a transformation is not always easy. In part IV we present an algorithm that transforms a grammar with infix, prefix and postfix operators of different precedence to an unambiguous context-free grammar. In part VI we use the algorithm to construct a recursive descent parser.

- We can modify a parsing method such that incorrect parse trees are thrown away during parsing. This is a common method [AJU75, Ear75, Sha88, Wha76]. Usually the parser is not constructed from a definition of a language but the parser can be said to define the language. In part V we show how Earley's algorithm [Ear70] can be extended to recognize a language defined by a precedence grammar. Precedence grammars and which language they generate are defined in the same part.

- We can transform the grammar to a simple unambiguous grammar which not necessarily express the correct structure of the sentences. An ordinary parser is used and a resulting parse tree is transformed to a correct one using the disambiguation information. An example of this method is described by Lalonde and des Rivieres [LdR81].

- We can use a parsing method that can handle ambiguous grammars and construct a parser from the ambiguous grammar without disambiguation information. The incorrect parse trees are thrown away after parsing.

Related Work

Many programming languages allow user defined infix operators, for example Algol68 [vWea75], Prolog [SS86], SML [MTH90], LML [AJ87] and Haskell [Hea91]. In Ada [Ge83] and C++ [Str86] it is possible to overload the predefined operators but it is not possible to define new ones. LML allow also user defined prefix and postfix operators. Not as many languages provide user defined distfix operators. Two examples are OBJ [FGJM85] and

Hope [BMS80]. Implementations of distfix operators are seldom described in the literature but in [Jon86] there is a description of how distfix operators can be parsed with YACC [Joh75].

Using infix operators as constructors, as we can for example in SML [MTH90], an object language can be defined. Such a language has a bit more flexible syntax than a language defined only by prefix constructors but the syntax is still very limited. It is for example hard to let constructors interfere with constructions of the programming language itself. We must also have a constructor for each part of the object language, even for those parts which do not have a constructor in the concrete syntax, for example application. These limitations remain also for distfix constructors.

Many parser tools have been developed during the years. The LALR parser generator YACC [Joh75] is widely available. A file containing a YACC specification of a language is transformed to a C program. Lex [Les75] is a lexical generator that could be used with YACC. Several implementations of ML, the SML from New Jersey [AM89] and the CAML from INRIA, Paris [CH90, WAL$^+$90], have parser generators in the style of YACC in their implementations. Wand [Wan84] has implemented a system for Scheme using YACC, to generate a parser that translates from abstract to concrete syntax. FPG [Udd88] is a functional parser generator. It is both written in and generates code for LML. Grune and Jacobs [GJ88] has implemented a parser generator that produces an LL(1)-parser.

The experimental programming language Lithe [San82] combines syntax directed translation with classes to allow the user to choose his own syntax. The Cigale system [Voi86] is a system for incremental grammar construction and expression parsing and allows flexible syntax.

The CAML system contains a facility to define parsers and printers [Mau89, MdR92]. A special pattern matching on a `stream` datatype is used. A way of writing parsers in functional languages using some kind of parser constructors has been explored by Burge [Bur75], Petersson and Holmström [Pet85], Fairbairn [Fai87] and Reade [Rea89].

In the Edinburgh LCF system [GMW79], elements of an object language can be entered using a concrete syntax but computations must be defined using the abstract syntax. Furthermore, LCF contains contains only one fixed object language. Konrad Slind is working on generalizing these ideas [Sli91] and compiler support for user-defined parsers and pretty-printers will soon be available in the SML from New Jersey.

An approach very similar to our conctypes has been implemented within the DML system [PF92a, PF92b]. A different parsing method (based on [HKR89]) is used and the DML system has a more sophisticated lexical analyzer. Another difference in DML is that keywords are denoted by string literals, and everything else are ML objects. This leads to readable syntax descriptions but expressing large elements in which each keyword is denoted by a string literal must be awkward. In conctype elements we instead use an *antiquotation symbol* to indicate *ML-objects*. It is not possible to resolve ambiguities with precedences in the DML system.

In the syntax definition formalism SDF [HHKR89] it is possible to use priorities to resolve ambiguities. The notation for these priorities is not as concise as our notation for precedences in the conctypes. Furthermore, the disambiguation does not work correctly for prefix and postfix operators.

Descriptions on how to construct a parser from an ambiguous grammar together with disambiguating rules can be found in [AJU75, Ear75, LdR81, Sha88, Wha76].

Overview

This thesis is divided into six parts, this introduction and five more parts numbered II–VI.

Part II : Concrete Syntax for Data Objects in Functional Languages

This part introduces conctypes. Some examples of conctypes and computations over the elements are given. The strong correspondence between a grammar and a conctype is shown and how well conctypes fit into the type system. It is explained briefly how the conctype elements are parsed.

Part III : Conctypes: Extensions, Algorithms and Implementation

This part describes the parsing algorithm used for conctypes. The parser is a generalization of Earley's algorithm and is integrated with a polymorphic type checker. Some extensions to conctypes and how to implement them are also described.

Part IV : Precedences in Specifications and Implementations of Programming Languages

This part contains two aspects concerning the use of precedences in distfix grammars, a subclass of context-free grammars. First, we present a parser independent definition of languages generated by distfix grammars together with precedences. We argue that this definition is reasonable both by a number of examples and some theorems. Second, we show that a transformation from a distfix grammar, where the operators have different precedence, to an unambiguous context-free grammar is quite complicated. An algorithm for such a transformation is given and the correctness of it is proved.

Part V : Precedences for Context-free Grammars

This part considers precedences for more general grammars than the distfix grammars introduced in part IV. One kind of ambiguity that is suitable to resolve with precedences is presented. The meaning of a grammar with precedences, in terms of the language that is generated, is given by an attribute grammar. We argue by some theorems that this definition is reasonable.

This part also describes how Earley's parsing algorithm can be extended to handle precedence rules. We also indicate how these rules can be used in LR-parsing.

Part VI : A Recursive Descent Parser for User Defined Distfix Operators

In this part an application of the algorithm in part IV is presented. A parser for user defined distfix operators is described. A new distfix operator is specified by the operator words, for example `if-then-else`, and optionally with precedence and associativity. These declarations, which essentially are a distfix grammar with precedences, must be transformed to an unambiguous grammar since we build the parser from parser constructors that construct a recursive descent parser.

Part II

Concrete Syntax for Data Objects in Functional Languages

Concrete Syntax for Data Objects in Functional Languages[2]

Annika Aasa *Kent Petersson* *Dan Synek*

1 Introduction

Many functional languages have a construction to define inductive data types [Hoa75] (also called general structured types [Jon87], structures [Lan64], datatypes [Mil84] and free algebras [GTWW77]). An inductive definition of a data type can also be seen as a grammar for a language and the elements of the data type as the phrases of the language. So defining an inductive data type can be seen as introducing an embedded language of values into the programming language. This correspondence is however not fully exploited in existing functional languages. The elements can presently only be written in a very restricted form. They are just the parse trees of the elements written in prefix form. A generalization, that we will consider in this paper, is to allow the elements to be written in a more general form. Instead of directly writing the parse trees of the embedded language, we would like to use a more concrete syntactical form and let an automatically generated parser translate the concrete syntactical form to the corresponding parse tree. We think that this is especially useful when we manipulate languages in programs, for example, when implementing compilers, interpreters, program transformation systems, and programming logics. It is also convenient if we want to use the concrete syntax for other kinds of data in a program.

By allowing distfix operators in a programming language [Jon86], it is possible to achieve some of the goals we have presented above. The problem is that the symbols comprising the distfix operator must not interfere with the constructions of the programming language itself. If we want to represent programs of a language in the language itself, this problem becomes acute. For example, to represent arithmetic expressions inside a functional language, it is difficult, but not impossible, to let 'x+23' in one situation be an expression which evaluates to an integer and in another a value that represents an arithmetic expression. We can solve this problem in at least two different ways. We can either say that the distfix operator must be built up from identifiers of the programming language or we can make a clear distinction between the programming language, the *metalanguage*, and the represented language, the *object language*. Of course we can relax the situation in the first case a little by allowing overloaded identifiers and operators in the metalanguage, but it is hard to imagine how pure syntactical constructions of the metalanguage, for example reserved words, could be overloaded.

[2]**Published in Proceedings of the 1988 ACM Conference on LISP and Functional Programming**

2 Concrete Data Types

We start by introducing a syntactical construction into our favorite functional language (ours is ML), to define a concrete data type of binary numbers[3] as:

```
conctype BinNumber =  [|0|]
                    |  [|1|]
                    |  [|<BinNumber>0|]
                    |  [|<BinNumber>1|]
```

Compare the type definition with the context free grammar for binary numbers

```
<BinNumber> ::=  0
               |  1
               |  <BinNumber>0
               |  <BinNumber>1
```

Since we must not confuse the symbols of the defined language (the Object Language – OL) with the symbols of the programming language (the Meta Language – ML), we enclose the elements in *quotation brackets*, `[|...|]`. Notice that the nonterminals in the grammar correspond to types in the type definition. The intention is to introduce a data type for binary numbers and let the elements be written in the familiar way. So in a program we would like to write the elements as `[|101|]` and `[|101001010|]`. We use the name *quotation expression* for this new form of expression.

We also want to be able to define computations over the elements, so we need a construction that separates the different forms a binary number can take and selects the components of a particular form. The modern way to do this in a functional language is to use pattern matching. We therefore introduce a pattern matching form for the elements of a type defined by our new constructor. A pattern is a sentential form of the language defined by the concrete datatype with ordinary ML-patterns of type A instead of nonterminals A. Since our patterns may contain ML patterns, we use an *antiquotation symbol*, '`^`', to write ML variables and patterns in the object language phrases. Variables can be written just following the antiquotation symbol but more complicated patterns must be enclosed in parentheses. Blanks after a variable are ignored. Examples of patterns for the concrete datatype of binary numbers are `[|101|]`, `[|^x|]`, `[|^x 101|]` and `[|^([|10|])1|]`. We use the name *quotation pattern* for this new form of pattern. By using this construction it is possible to write a function that takes a binary number as argument and gives its successor as result:

```
fun succ [|0|]      = [|1|]
|   succ [|1|]      = [|10|]
|   succ [|^b 0|]   = [|^b 1|]
|   succ [|^b 1|]   = [|^(succ b)0|]
```

Notice that we have used the antiquotation symbol also in the quotation expressions in the right hand sides of the function definition. In the example, there is first a quotation expression `[|^b 1|]` which is intended to construct a phrase from the value bound to the ML-variable `b` and the symbol `1`. The variable must, of course, be bound to a value of

[3] We ignore that binary numbers are not a free data type.

type `BinNumber` since such a value is expected in this position. Secondly, there is the more complicated expression `[|^(succ b)0|]` where the ML-expression `succ b` is evaluated to a value of type `BinNumber` and this number is then composed with the symbol `0` to produce a binary number. All ML-expressions inside a quotation must evaluate to complete phrases of the language and not to strings of characters.

Patterns in a function definition need not directly correspond to the cases in the definition of the concrete datatype, as can be seen by the following example:

```
fun div4 [|^x 00|] = true
|   div4 y         = false
```

The pattern matching using the concrete syntax is important in our approach to representing object languages. Without it, one has to introduce somewhat arbitrary names for the operations that decides what form an element has and for the operations that selects the components of a compound element. Compare our quotation brackets and antiquotation symbol with the corresponding constructions in the Edinburgh LCF system [GMW79] and also with Quine's quasi-quotation [Qui81].

As a second example consider implementing the denotational description of a very simple imperative language. We first define the language

```
conctype Pgm
  =  [|program <Cmds> end|]
and Cmds
  =  [|<Cmd>|]
  |  [|<Cmd>; <Cmds>|]
and Cmd
  =  [|if <Bexp> then <Cmds> else <Cmds> fi|]
  |  [|while <Bexp> do <Cmds> end|]
  |  [|<Var>:=<Exp>|]
and Exp
  =  [|<Var>|]
  |  [|<Integer>|]
  |  [|<Exp>+<Exp>|]
  |  [|<Exp>*<Exp>|]
  |  [|(<Exp>)|]
and Bexp
  =  [|<Exp>=<Exp>|]
  |  [|<Bexp>|<Bexp>|]
  |  [|<Bexp>&<Bexp>|]
  |  [|(<Bexp>)|]
```

where we assume we already have defined two concrete datatypes `Var` and `Integer` and a function `toint` that maps a concrete integer to the corresponding ML value. Using the concrete datatype defined above, we can define an interpreter in a natural way. Notice that the definition is very close to how Gordon describes the denotational semantics of a language in [Gor79]. We assume we already have implemented an abstract data type for states, with operations `sinit`, `update` and `valof`.

```
fun P  [|program ^(cs) end|] = Cs cs sinit
and Cs [|^c|] s      = C c s
|   Cs [|^c; ^cs|] s = Cs cs (C c s)
and C  [|if ^(b) then ^(s1) else ^(s2) fi|] s =
       if B b s then Cs s1 s else Cs s2 s
|   C  [|while ^(b) do ^(cs) end|] s =
       let fun f s = if B b s then f (Cs cs s)
                              else s
       in f s
       end
|   C  [|^x:=^e|] s = update(x, E e s, s)
and E  [|^x|] s       = valof(x,s)
|   E  [|^n|] s       = toint n
|   E  [|^e1+^e2|] s  = E e1 s + E e2 s
|   E  [|^e1*^e2|] s  = E e1 s * E e2 s
|   E  [|(^e)|] s     = E e s
and B  [|^e1=^e2|] s  = E e1 s = E e2 s
|   B  [|^b1|^b2|] s  = B b1 s orelse  B b2 s
|   B  [|^b1&^b2|] s  = B b1 s andalso B b2 s
|   B  [|(^b)|] s     = B b s
```

This example raises a problem of how to decide what different patterns means. The first and second cases in the definition of `E` just consist of ML-variables and the problem is how to decide that the first is the variable case and the second the integer case. We use information from the type inference mechanism to choose between the two possibilities. The pattern is not parsed until the typechecker already has typechecked the right hand sides of the definition and then we know that `x` in the first case must be of type `Var` and `n` in the second must be of type `Integer`. From this information it is possible to distinguish which case a pattern is supposed to denote. It is of course possible to try to define a function where one can not decide what cases two patterns are supposed to denote. Take for example the definition of a function that counts the number of variables in an expression:

```
fun Vars [|^x|]         = 1
|   Vars [|^n|]         = 0
|   Vars [|^e1+^e2|]    = Vars e1 + Vars e2
|   Vars [|^e1*^e2|]    = Vars e1 + Vars e2
|   Vars [|(^e)|]       = Vars e
```

In this example it is impossible to choose between the two cases if we do not use the variable name to indicate its type, as we do in denotational descriptions and Fortran(!!). Our solution is to allow the user to explicitly type the variables. So the first two cases in the definition must be written as:

```
fun Vars [|^(x:Var)|]     =  1
|   Vars [|^(n:Integer)|] =  0
...
```

We consider it to be an error if we do not have enough information to parse a quotation pattern unambiguously.

We have another problem if we want to define a function

```
fun isadd [|^x+^y|] = true
|   isadd z         = false
```

because the type checker and quotation parser can not give a unique type to the variables `x` and `y`. They can either be of type `Var`, `Integer` or `Exp`, so the pattern is ambiguous and therefore erroneous. To make it unambiguous the user must provide type information. Notice that the type information distinguishes the more restrictive pattern `[|^(x:Var)+^(y:Var)|]` from `[|^(x:Exp)+^(y:Exp)|]`. Problems with ambiguities in patterns are discussed in a paper [DGKLM84].

The concrete data types fit nicely into the ordinary typesystem in ML and we can for example define polymorphic concrete data types such as trees with information in the nodes.

```
conctype 'A Tree = [|o|]
                 | [|{<'A Tree>-<'A>-<'A Tree>}|]
```

with elements like

```
[|{o-^("HEJA")-{o-^("BARACKEN")-o}}|]   :  String Tree
```

and

```
[|{o-1010-o}|] : BinNumber Tree
```

and a function that swaps the left and right part of a tree

```
fun swaptree [|o|]          = [|o|]
|   swaptree [|{^x-^y-^z}|] = [|{^z-^y-^x}|]
```

As can be seen from the String Tree example above, it is possible to use ordinary ML types when defining concrete types.

3 Lexical Analysis

It is not obvious what should be treated as a lexical token in the embedded languages. In order to be flexible and allow as many and as different concrete data types as possible, we have decided to view every character as a lexical token. The only exceptions to this are that a sequence of blanks is treated as one blank and that the escape character '\' gives the following character its literal meaning. The result of this is that blanks are not handled nicely. If we want to have blanks in a quotation expression then there must be a blank character in the corresponding position in the grammar, and if a blank is present in the grammar there must be at least one blank in the quotation.

Having a more sophisticated lexical analyzer give us another problem. We can not use parts of a lexical token in the grammar. For example if we use ML:s lexical analyzer, as they do in the LeML system [INR85], we can not define the binary numbers as we do in section 2 since a sequence of zeros and ones is treated as an integer in ML.

The best solution would probably be to give the user the possibility to define her own lexical analyzer.

4 Parsing and Type Derivation

In this section we describe how the new constructions are translated during the compilation to ordinary data types and constructors.

After the compilation nothing of the new constructions remains and they have therefore no effect on the execution speed of the new syntactical constructions. A program with concrete datatypes, quotations and anti-quotations runs at the same speed as one without them.

Let us give an overview of the translation process. A concrete datatype is translated into the following objects.

- A datatype definition, which could be seen as the type of parse trees for the language defined by the grammar in the concrete datatype.
- A parser that recognizes the language described by the grammar and translates a phrase of the language into the parse tree.
- A (pretty?) printer that prints elements of the concrete datatype using the concrete syntax.

The type definition of binary numbers,

```
conctype BinNumber =  [|0|]
                    |  [|1|]
                    |  [|<BinNumber>0|]
                    |  [|<BinNumber>1|]
```

will be translated into the following datatype definition.

```
datatype BinNumber = BinNumber1
                   | BinNumber2
                   | BinNumber3 of BinNumber
                   | BinNumber4 of BinNumber
```

and a parser that, for example, translates `[|101|]` to
`BinNumber4(BinNumber3(BinNumber2))`

The parser is used in the compiler to translate quotation patterns and quotation expressions to ordinary patterns and expressions. For example, the function

```
fun succ [|0|]     = [|1|]
|   succ [|1|]     = [|10|]
|   succ [|^b 0|]  = [|^b 1|]
|   succ [|^b 1|]  = [|^(succ b)0|]
```

is translated to

```
fun succ (BinNumber1)   = BinNumber2
|   succ (BinNumber2)   = BinNumber3 BinNumber2
|   succ (BinNumber3 b) = BinNumber4 b
|   succ (BinNumber4 b) = BinNumber3 (succ b)
```

Usually the syntax of an ML program is checked in two distinct steps. First it is parsed with a context free parser and then typechecked with a type checker utilizing Milner's algorithm [Mil78]. This simple sequence is no longer possible when concrete data types are added to ML since the parsing of a concrete element depends of its type. That is, if a concrete expression is in a context where an element of type A is expected it should be parsed with A as the start symbol. Correspondingly, the type of an ML expression in a quotation is dependent on the parsing of the quotation. To achieve this, we have to integrate the parsing of the concrete elements with Milner's type derivation algorithm. We have built the parser around a generalized version of Earley's algorithm [Ear70]. It is generalized for two reasons:

- Concrete data types seen as grammars are more powerful than context free grammars for which Earley's algorithm is constructed. This stems from the fact that we can have polymorphic concrete data types. A simple example of a language that can be defined by a polymorphic data type but not with a context free grammar are the trees introduced in section 2.
- A parser defined with Earley's algorithm usually takes a sequence of lexical tokens as input and gives a parse tree as output. However, the parser that we want should take a sequence of lexical tokens *and unquoted ML objects* as input and give an ML expression as output.

The algorithm we have developed differs from Earley's in that we can have type variables in the nonterminals and these can be instantiated during parsing. For example, given the conctype

```
conctype 'A List = [|$|]    |    [|<'A><'A List>|]
and
conctype D       = [|0|]    |    [|1|]
```

and the input `[|0$|]`, `'A` will be instantiated to `D`. To support this, each item in Earley's algorithm contains, apart from the dotted production and the item set pointer, a type substitution. The substitutions are handled in the following way by the parsing operations:

- In the predict operation in Earley's algorithm an item is added only once even though it might be predicted from more than one item. In our version each new item is created with an initial substitution. The more information we let this substitution inherit from the predicting item the less is the chance that we can share it. To assure maximal sharing we let each item start with the empty substitution. When the dot is to the left of an uninstantiated type variable all conctypes known in the context are predicted.
- In the completion of an item I, we return to the item set pointed to and add updated versions of all items I' which have a type T' to the right of the dot that can be unified with the type T we have just completed. Since there can be more than one item with this property we have to make sure that we use fresh variables for the type variables in T before unification. We update I' by moving the dot one step to the right and compose the substitution with the substitution in I.

Let us illustrate this with an example: Corresponding to the input `[|$$|]` and the conctype

```
conctype 'A List = [|$|]
                 | [|<'A><'A List>|]
```

we get the item sets:

I_0

$$
\begin{array}{lcll}
\text{<S>} & ::= & \cdot\text{<}'A\text{ List>}, s_0, 0 & (1)\\
\text{<}'A\text{ List>} & ::= & \cdot\$, [], 0 & \\
\text{<}'A\text{ List>} & ::= & \cdot\text{<}'A\text{><}'A\text{ List>}, [], 0 &
\end{array}
$$

I_1

$$
\begin{array}{lcll}
\text{<}'A\text{ List>} & ::= & \$\cdot, [], 0 & (2)\\
\text{<S>} & ::= & \text{<}'A\text{ List>}\cdot, s_0['A \mapsto 'X_1], 0 & (3)\\
\text{<}'A\text{ List>} & ::= & \text{<}'A\text{>}\cdot\text{<}'A\text{ List>}, & \\
 & & \quad ['A \mapsto 'X_2\text{ List}], 0 & \\
\text{<}'A\text{ List>} & ::= & \cdot\$, [], 1 & \\
\text{<}'A\text{ List>} & ::= & \cdot\text{<}'A\text{><}'A\text{ List>}, [], 1 &
\end{array}
$$

I_3

$$
\begin{array}{lcl}
\text{<}'A\text{ List>} & ::= & \$\cdot, [], 1\\
\text{<}'A\text{ List>} & ::= & \text{<}'A\text{><}'A\text{ List>}\cdot,\\
 & & \quad ['A \mapsto 'X_2\text{ List},\\
 & & \quad\ 'X_3 \mapsto 'X_2\text{ List}], 0\\
\text{<}'A\text{ List>} & ::= & \text{<}'A\text{>}\cdot\text{<}'A\text{ List>},\\
 & & \quad ['A \mapsto 'X_4\text{ List}], 1\\
\text{<S>} & ::= & \text{<}'A\text{ List>}\cdot s_0['A \mapsto 'X_5,\\
 & & \quad\ 'X_5 \mapsto 'X_2\text{ List},\\
 & & \quad\ 'X_3 \mapsto 'X_2\text{ List}], 0\\
\text{<}'A\text{ List>} & ::= & \text{<}'A\text{>}\cdot\text{<}'A\text{ List>},\\
 & & \quad ['A \mapsto 'X_6\text{ List},\\
 & & \quad\ 'X_6 \mapsto 'X_2\text{ List},\\
 & & \quad\ 'X_3 \mapsto 'X_2\text{ List}], 0\\
\text{<}'A\text{ List>} & ::= & \cdot\$, [], 2\\
\text{<}'A\text{ List>} & ::= & \cdot\text{<}'A\text{><}'A\text{ List>}, [], 2
\end{array}
$$

The composition of substitutions s and s' is written ss'. s_0 is the substitution obtained from Milner's type derivation algorithm applied to the ML parts of the program.

Notice that before we unify the left hand side of the production in (2) with the type to the right of the dot in (1) to get (3) we substitute a new variable $'X_1$ for $'A$ in (2).

We now explain what to do with unquoted expressions in the input. An unquoted expression of type T can be viewed as an already parsed part of the input that can be accepted whenever we have an item in the item set with the dot to the left of a type that can be unified with T. We illustrate this with the same grammar as above and the input `[|^(x:Int)$|]`

I_0

$$
\begin{array}{lcl}
\text{<S>} & ::= & \cdot\text{<}'A\text{ List>}, s_0, 0 \\
\text{<}'A\text{ List>} & ::= & \cdot\$, [], 0 \\
\text{<}'A\text{ List>} & ::= & \cdot\text{<}'A\text{><}'A\text{ List>}, [], 0
\end{array}
$$

I_1

$$
\begin{array}{lcl}
\text{<}'A\text{ List>} & ::= & \text{<}'A\text{>}\cdot\text{<}'A\text{ List>}, [{}'A \mapsto \text{Int}], 0 \\
\text{<}'A\text{ List>} & ::= & \cdot\$, [], 1 \\
\text{<}'A\text{ List>} & ::= & \cdot\text{<}'A\text{><}'A\text{ List>}, [], 1
\end{array}
$$

I_3

$$
\begin{array}{lcl}
\text{<}'A\text{ List>} & ::= & \$\cdot, [], 1 \\
\text{<}'A\text{ List>} & ::= & \text{<}'A\text{><}'A\text{ List>}\cdot, [{}'A \mapsto \text{Int}, \\
 & & \quad {}'X_1 \mapsto \text{Int}], 0 \\
\text{<}'A\text{ List>} & ::= & \text{<}'A\text{>}\cdot\text{<}'A\text{ List>}, \\
 & & \quad [{}'A \mapsto {}'X_2\text{ List}], 1 \\
\text{<S>} & ::= & \text{<}'A\text{ List>}\cdot, s_0[{}'A \mapsto {}'X_3, \\
 & & \quad {}'X_3 \mapsto \text{Int}, \\
 & & \quad {}'X_1 \mapsto \text{Int}], 0 \\
\text{<}'A\text{ List>} & ::= & \text{<}'A\text{>}\cdot\text{<}'A\text{ List>}, [{}'A \mapsto \text{Int List}, \\
 & & \quad {}'X_4 \mapsto \text{Int}, \ {}'X_1 \mapsto \text{Int}], 0 \\
\text{<}'A\text{ List>} & ::= & \cdot\$, [], 2 \\
\text{<}'A\text{ List>} & ::= & \cdot\text{<}'A\text{><}'A\text{ List>}, [], 2
\end{array}
$$

5 User Defined Representation

In the previous section we have described how the elements of the concrete datatypes are represented. Sometimes this representation is a bit inconvenient because we often want to represent sequences as ML lists and sometimes we would like to "forget" some productions in the grammar. These differences in representation should be indicated by constructions in the definition of conctypes.

Productions containing terminals whose only purpose are to indicate the structure of

a sentence are of course unnecessary when the sentence is represented as a tree. Take for example parentheses in arithmetic expressions. In the denotational description in section 2 we have the clauses `[|(<Exp>)|]` and `[|(<Bexp>)|]` because we want to have parentheses in elements of the concrete datatypes `Exp` and `Bexp`. These productions remain in the representation and we must therefore have the cases:

```
E [|(^e)|] s = E e s
B [|(^b)|] s = B b s
```

in the interpreter. To eliminate these let us introduce the notation

```
A = [||...<A>...||]
```

to indicate that the production should not remain in the representation. The dots stands for arbitrary terminal symbols. Our denotational example would then be written:

```
Exp  = ...
     | [||(<Exp>)||]
Bexp = ...
     | [||(<Bexp>)||]
```

The parentheses may only occur in quotation expressions. They are eliminated during parsing and must not appear in quotation patterns. The only situation when it is possible to forget productions in this way is of course when the clause just contains one nonterminal and this is the same as the conctype just being defined.

In grammars, we often use a special notation for sequences.

$$Cmds ::= Cmd\ \{;Cmd\}$$

In the conctypes described so far we have used recursion to define sequences. For example in the denotational description in section 2 where a sequence of commands is defined by:

```
conctype Cmds = [|<Cmd>|]
              | [|<Cmd>; <Cmds>|]
```

A sequence of commands `c1;c2;c3` is represented by the term:

```
Cmds2(c1,Cmds2(c2,Cmds1 c3))
```

Having this representation we must use explicit recursion when defining computations over the elements. If we instead represent the sequences as ML lists it is possible to use predefined list handling functions. When representing sequences as ML lists it is necessary to exclude the terminals which separate the elements. We therefore use a somewhat different notation for sequences than in ordinary grammar descriptions.

```
{<Cmd>; ...}+
```

The nonterminal inside the curly brackets defines the elements in the sequence. The terminals between the nonterminal and the three dots are the separators between the elements in the sequence. We use `+` to denote repetition one or more times. It is also possible to use `*` to denote zero or more times. Using this notation the command sequence in the denotational description could be defined

```
conctype Pgm = [|program {<Cmd>; ...}+ end|]
```

The conctype `Cmds` is then no longer necessary. Neither is the function `Cs` in the interpreter. Instead we define the function `P` thus

```
fun P [|program ^(cs) end|] = foldleft C sinit cs
```

where `foldleft` is a predefined list handling function.

The conctype `Pgm` is translated to the datatype:

```
datatype Pgm = Pgm1 of Cmd list
```

Using list notation, the definition of a language by conctypes becomes more comprehensible. A procedure head in a simple imperative language could with list notation be defined as:

```
conctype procedurehead
   = [|PROC <Id>;|]
   | [|PROC <Id>({<Parlist>;...}+);|]

and Parlist
   = [|{<Id>,...}+:<TypeId>|]
```

where `Id` and `Typeid` are two already defined conctypes. An element of this type is `[|PROC P(a,b:int;ch:char);|]` and a function that counts the number of parameters is:

```
fun countpars [|PROC ^id;|]       = 0
|   countpars [|PROC ^id (^ps);|] =
          sum (map (fn [|^ids:^t|] => (length ids)) ps)
```

where `sum`, `map` and `length` are predefined list handling functions. Without list notation the conctype definition must be defined as:

```
conctype Prochead
   = [|PROC <Id>;|]
   | [|PROC <Id>(<Parlist>);|]

and Parlist
   = [|<Idlist>:<Typeid>|]
   | [|<Idlist>:<Typeid>;<Parlist>|]

and Idlist
   = [|<Id>|]
   | [|<Id>,<Idlist>|]
```

and the function that counts the number of parameters:

```
fun countpars [|PROC ^id;|]      = 0
|   countpars [|PROC ^id(^ps);|] = countpar ps
```

```
fun countpar [|^ids:^t|]     = countids ids
|   countpar [|^ids:^t;^ps|] = countids ids + countpar ps

fun countids [|^id|]         = 1
|   countids [|^id,^ids|]    = 1 + countids ids
```

As previously mentioned, it is considered to be an error if a quotation expression can not be parsed uniquely. It is therefore desirable to have unambiguous conctypes. Consider arithmetic expressions. An unambiguous conctype would contain a lot of nonterminals and productions which are irrelevant in the representation. Many function definitions thus become quite complicated. Using precedences and associativity rules which resolve ambiguities, we can define the same language with a less complicated conctype, and have more natural patterns. A desirable extension is therefore to give the user the possibility of giving precedence and associativity rules, to productions in the conctypes.

A completely different representation could be obtained by defining a datatype and a function that maps the concrete object to the new representation. If we for example want to represent binary numbers as integers we could define a function `bintoint` which maps the concrete data type `BinNumber` to the corresponding integer.

```
fun bintoint [|0|]    = 0
|   bintoint [|1|]    = 1
|   bintoint [|^x 0|] = 2 * bintoint x
|   bintoint [|^x 1|] = 2 * bintoint x + 1
```

Compare this with the following definition in YACC.

```
binnumb: ZERO           {$$ = 0;}
|        ONE            {$$ = 1;}
|        binnumb ZERO   {$$ = $1*2;}
|        binnumb ONE    {$$ = $1*2+1;}
```

6 Implementation

The constructions we described in section 2 have been implemented in the functional language LML [AJ87, Aug84] and all examples in that section have been tested in the implementation. The constructions described in section 5 are not implemented yet.

LML has no input and output for user defined datatypes so we have not bothered to generate pretty printers for conctypes in our implementation.

7 Related Work

One system with an explicit notion of metalanguage and object language is the Edinburgh LCF system [GMW79]. In contrast to our proposal, LCF contains only one fixed object language. Furthermore, the object language is represented by an abstract type that defines the abstract syntax of the language, so the concrete syntax in quotations is seen just as a convenient way for the user to enter elements of this type. To define computations that uses the object language one has to use the constructors and selectors of the abstract

syntax and the user must therefore remember both the concrete and the abstract syntax of the object language.

In the LeML system from INRIA [INR85, Hue86], the user can easily define his own object language by using an interface with an ML version of YACC. But the concrete syntax must still be seen as a convenient form to write abstract syntax trees since all computations must be expressed in terms of the constructors and selectors of the abstract syntax. Nothing like our quotation patterns is available. Wand [Wan84] has implemented a similar system for Scheme also using YACC to generate the parser that translates from concrete to abstract syntax.

A more limited way to define an object language is to use infix operators as constructors, as we can do in ML [Mil84]. A type declaration for arithmetic expressions involving integers and the operators `+` and `*` can in ML be defined as

```
infix ++ **
datatype Expr = NUM of int
              | op ++ of Expr * Expr
              | op ** of Expr * Expr
```

A function which evaluates such an expression is:

```
fun E (NUM n)    = n
|   E (e1 ++ e2) = E e1 + E e2
|   E (e1 ** e2) = E e1 * E e2
```

This way of making the elements of a datatype more concrete has a number of disadvantages. Since ML does not allow overloading we can not use the symbols `+` and `*` as constructors. We have to choose other symbols, for example `++` and `**` as in the example above. We must also have a constructor for each part of the new language we want to define, even for those parts which do not have a constructor in the concrete syntax, like the integer case in the example above. The expression `2*3+4` must therefore be written `(NUM 2)**(NUM 3)++(NUM 4)`. If we want to write expressions in a more familiar way we must write a parser which translates strings to elements in the datatype.

8 Future Work

In the future we will implement the constructions described in the section on user defined representation. We will also define and implement constructions for expressing priorities and associativity. The problems with the lexical analyzer also have to be further investigated. Another interesting question is if the notion of subtype [FM88] could resolve some of the problems with ambiguities in patterns we have described.

Acknowledgements

We would like to thank the members of the Programming Methodology Group in Göteborg for their help and encouragement. In particular we would like to thank Lennart Augustsson for his support and all help he provided during the implementation and Sören Holmström for his useful comments.

Part III

Conctypes: Extensions, Algorithms and Implementation

Conctypes: Extensions, Algorithms and Implementation

Annika Aasa

1 Background

A new datatype construction, *the conctype* [APS88], the elements of which have a flexible syntax is described in part II of this thesis. A brief introduction is given in part I. A conctype definition corresponds to a grammar where the nonterminals correspond to types. The types in a conctype definition need not only be conctypes but can also be ordinary types. We may also define polymorphic conctypes. The elements of conctypes, *quotations*, are enclosed in brackets and are translated to a corresponding abstract syntax by the compiler. In appendix A we give a conctype for a subset of Pascal together with a quotation and an interpreter for the language. Conctypes fits nicely into the ordinary type system and the parsing of quotations is integrated with the ordinary type checking since the parsing of a quotation both determines its type and is dependent of types in its context.

In this part we describe the parsing and type checking algorithm. We also reconsider some special constructs which are mentioned or briefly described in part II and describe how these are implemented.

2 Special Constructions

In part II we described some occasions when the automatically obtained representation is a bit inconvenient. Sequences are preferably represented as lists and we sometimes would like to "forget" productions. We also mentioned a problem with ambiguous conctypes. In this part we will reconsider the sequence notation, and describe the possible patterns in more detail. We will also describe how to use precedences to resolve some ambiguities. The implementation of these features is described in section 3.3.

2.1 Sequence notation

Most languages contain sequences of different objects. For example in Pascal we can find sequences of type-, procedure- and function definitions, formal and actual parameters, statements and so on. When defining sequences with a grammar one can either use explicit recursion or some special notation. A sequence of statements can, for example, either be defined as:

```
Program     ::=  begin  Statementseq  end

Statementseq ::=  Statement
               |  Statement  ;  Statementseq
```

or as:

Program ::= *Statement* { ; *Statement* }

Special notation for sequences has the advantage of often decreasing the number of nonterminals in grammars, which enhances readability. For some sequences, the special notation eliminates two nonterminals.

For conctypes there is also another reason to have a special notation for sequences. The representation obtained using explicit recursion is sometimes unconvenient since explicit recursion must also be used when defining computations over the elements. It is better to represent sequences as LML lists since list handling functions are then possible to use.

The elements in many sequences are *separated* by some terminals.

```
var  x,y,z : integer
```
a sequence of variables separated by commas

```
var n:integer ; ch:char ; ready:boolean
```
a sequence of variable declarations separated by semicolons

When representing sequences as LML-lists it is convenient to exclude the terminals which separate the elements. We therefore use a somewhat different notation for sequences than in ordinary grammar descriptions:

```
{<vardef>; ...}+
```

The type inside the curly brackets defines the elements in the sequence. The terminals between the type and the three dots are the separators between the elements in the sequence. We use `+` to denote nonempty sequences and `*` for possibly empty sequences. In Pascal, the variable definition part without sequence notation is defined as:

```
conctype vardefpart = [|var <vardefseq>|]
                    + [||]

conctype vardefseq  = [|<idseq> : <Type> ; <vardefseq>|]
                    + [|<idseq> : <Type> |]

conctype idseq      = [|<Id>,<Idseq>|]
                    + [|<Id>|]
```

We assume that `Id` and `Type` are already defined conctypes. Using sequence notation, the definition becomes much more comprehensible:

```
conctype vardefpart = [|var {<vardef>; ...}+|]
                    + [||]

conctype vardef     = [|{<Id>, ...}+ :<Type>|]
```

The elements in a sequence must be defined by a type. It is not possible to have only terminals or a mixture of nonterminals and terminals. The reason for this restriction is that the elements of the sequence should have a type with a name. Thus, the introduction

of the conctype `vardef` in the previous example is necessary. Something like

```
[|var {{<Id>, ...}+ : <Type>; ...}+|]
```

is illegal.

As mentioned in part II each conctype has an associated type and quotations will be translated to elements of this type during compilation. The associated type for a conctype containing a sequence notation will have constructors taking LML lists as arguments instead of just elements of conctypes. The conctype `Exp` contains a sequence notation:

```
conctype Exp = [|<Num>|]
             + [|<Var>|]
             + [|<Exp> where {<Def>,...}+|]

conctype Def = [|<Var>=<Exp>|]
```

Therefore, the associated types for the conctypes `Exp` and `Def` are:

```
type Exp = Exp-0 Num
         + Exp-1 Var
         + Exp-2 Exp (List Def)

type Def = Def-0 Var Exp
```

An antiquotation of type `List`(T) can occur in the position of a sequence notation where T is the type of the elements in the sequence. Consider the following pattern given the conctypes `Exp` and `Def`:

```
[|^(e) where ^ds|]
```

During the parsing and type checking, the variable `e` will be assigned type `Exp` and `ds` the type `List(Def)`. It is possible to have more complicated antiquotations in the position of a sequence notation as long as the type of it is correct, for example:

```
[|^(e) where ^([ [|x=5|] ; [|y=7|] ])|]
```

Note that semicolon is the LML list element separator. More complicated patterns are also possible. Remember that antiquotations occur in the positions of nonterminals in the corresponding sentential form of a quotation. From the right hand side:

```
[|<Exp> where {<Def>,...}+|]
```

we can for example derive the following sentential forms:

```
<Exp> where {<Def>,...}+
<Exp> where <Def>
<Exp> where <Def> , {<Def>,...}+
<Exp> where <Def> , <Def>
<Exp> where <Def> , <Def> , {<Def>,...}+
                  ⋮
```

Therefore the possible patterns are:

```
[|^(e) where ^ds|]
[|^(e) where ^d1,^d2|]
[|^(e) where ^d1,^d2,^d3|]
            ⋮
```

As can be seen from the sentential forms, the last antiquote in each example could either denote an element in the list (with type `Def` in this case) or the rest of the list (with type `List(Def)`). Which it is denoting must be resolvable from the context or by explicit typing. Patterns corresponding to even more complicated sentential forms can also be used, for example:

```
[|^(e) where ^id1=^e1,^id2=^e2,^ds|]
```

We have chosen to let the terminals inside the curly brackets *separate* the elements in the sequence since this often is the case for sequences. The sequence notation can of course be used also for sequences which do not have a separator between the elements. Then the sequence notation is used without any terminals between the conctype and the three dots.

There are no problems to use the sequence notation also for sequences in which each element ends with something, rather than having separators between the elements. The "separator" is just repeated after the sequence notation. In fact, such a sequence can be hard to distinguish from a sequence in which the elements are separated by some terminals and then *followed* by the same terminals. An example from Pascal is the type definition part:

```
conctype typedefpart = [|type {<typedef>; ...}+;|]
                     + [||]

conctype typedef     = [|<Id> = <Type>|]
```

One interpretation is that each type definition ends with a semicolon. Another is that the sequence of type definitions are separated by semicolons and that the last semicolon ends the whole type definition part. Regardless of the interpretations, our notation is the same. Note that the last semicolon remains in the representation and patterns must therefore be written:

```
[|type ^tdefs ;|]
```

2.2 Precedences

If a quotation cannot be parsed uniquely and the ambiguity is not resolvable from the context, it is considered to be an error. One ambiguity that cannot be resolved from the context is if a quotation without antiquotations can be parsed in different ways and all parses have the same type. Consider for example a conctype for simple arithmetic expressions:

```
conctype Exp = [|<Int>|]
             + [||(<Exp>)||]
             + [|<Exp>*<Exp>|]
```

```
              + [|<Exp>/<Exp>|]
              + [|<Exp>+<Exp>|]
              + [|<Exp>-<Exp>|]
```

The quotation

```
[|2+3-4|]
```

is ambiguous and the ambiguity cannot be resolved from the context, since the corresponding context-free grammar is ambiguous. Most quotations derived from the conctype `Exp` are ambiguous so in order to use it we must make the conctype unambiguous in some way. Rewriting it, in order to both make it unambiguous and such that the structure of the parse trees will be as desired, forces us to introduce a lot of new conctypes:

```
conctype Exp  = [|<Exp>+<Term>|]
              + [|<Exp>-<Term>|]
              + [|<Term>|]
conctype Term = [|<Term>*<Fact>|]
              + [|<Term>/<Fact>|]
              + [|<Fact>|]
conctype Fact = [|<Int>|]
              + [|(<Exp>)|]
```

All new types are irrelevant in the representation and many function definitions will be quite complicated, for example an interpreter:

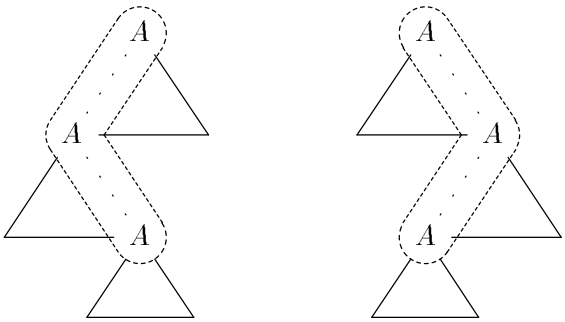
```
rec evalE [|^e+^t|] = evalE e + evalT t
||  evalE [|^e-^t|] = evalE e - evalT t
||  evalE [|^t|]    = evalT t

and evalT [|^t*^f|] = evalT t * evalF f
||  evalT [|^t/^f|] = evalT t / evalF f
||  evalT [|^f|]    = evalF f

and evalF [|^n|]    = n
||  evalF [|(^e)|]  = evalE e
```

Thus, the unambiguous conctype above is not only complicated in itself but forces also computations to be more complicated. However, there is another way to make a grammar (or conctype) unambiguous. We can use precedence and associativity rules for disambiguating. It is however not obvious what kind of ambiguity that is possible or suitable to resolve with precedences. Since it is not decidable if a grammar is ambiguous or not [AU72, Can62, Flo62]. We think it is impossible to resolve all ambiguity with precedence rules.

In part V of this thesis, the notion of *R-ambiguity* is defined. It is a kind of ambiguity that arises from the fact that a grammar generates *R-ambiguous* parse trees. An R-ambiguous parse tree is, or contains a subtree, of one of the following forms:



Motivations that R-ambiguities are suitable to resolve with precedences and a definition of what it means to give precedences to productions is given in part V. The precedences are given to the alternatives of every nonterminal which is both left- and right recursive.

For conctypes we use this approach of resolving ambiguity with precedences. The `<` sign indicates that a production has higher precedence[4] than the previous one. The `=` sign indicates that the production has equal precedence to the previous one. A production that is marked with a `<` sign can also be given an associativity. This is done by the reserved words: `leftassoc`, `rightassoc` and `nonassoc` indicating the production to be left associative, right associative or nonassociative. A production having the same precedence as another production also has the same associativity. An unambiguous conctype for arithmetic expressions can then be written.

```
conctype Exp =             [|<Int>|]
             +             [||(<Exp>)||]
             + < leftassoc [|<Exp>*<Exp>|]
             + =           [|<Exp>/<Exp>|]
             + < leftassoc [|<Exp>+<Exp>|]
             + =           [|<Exp>-<Exp>|]
```

If a grammar contains more than one conctype which is both left and right recursive, precedences are given to all these different conctypes.

```
conctype Exp  =              [|<Bexp>|]
              +              [|<Iexp>|]
     and Iexp =              [|<Int>|]
              + < leftassoc  [|<Iexp>*<Iexp>|]
              + =            [|<Iexp>/<Iexp>|]
              + < leftassoc  [|<Iexp>+<Iexp>|]
              + =            [|<Iexp>-<Iexp>|]
     and Bexp =              [|true|]
              +              [|false|]
              +              [|<Iexp>=<Iexp>|]
              + <            [|not <Bexp>|]
              + < leftassoc  [|<Bexp>&<Bexp>|]
              + =            [|<Bexp>|<Bexp>|]
```

[4] We use the a bit unusual convention that for example `+` has higher precedence than `*`.

Note that the production `Bexp = [|<Iexp>=<Iexp>|]` is not given a precedence. It is not necessary since there are no derivations:

$$\texttt{Bexp} \Rightarrow \texttt{Iexp=Iexp} \Rightarrow^{+} \texttt{Bexp}\ \alpha$$
$$\texttt{Bexp} \Rightarrow \texttt{Iexp=Iexp} \Rightarrow^{+} \alpha\ \texttt{Bexp}$$

Polymorphic conctypes are given precedences in the same way.

3 Parsing of Quotations and Type Derivation

The parsing of quotations is integrated with the type checking since the parsing of a quotation both determines its type and is dependent of types in its context. Type checking a quotation means that it is parsed and antiquotations are type checked.

There are two reasons why an ordinary parser for context-free grammars cannot parse quotations.

- Conctypes can be polymorphic and are therefore more powerful than context-free grammars. There are variables in the conctypes, and therefore a conctype can be seen as a special case of a two-level grammar [CU77].
- The input can be arbitrary sentential forms (and not only those consisting of solely terminal symbols) since a quotation can contain *antiquoted LML objects*.

To handle these two points we have chosen to generalize Earley's context-free parsing algorithm [Ear70]. Our motives for chosing Earley's algorithm are that it is a *general* context-free parsing algorithm and thus there are no restrictions on the context-free grammar behind a conctype. The algorithm can handle also ambiguous grammars. Since the algorithm is quite simple one could expect that it is not too hard to generalize it.

The type checker is based on Damas' and Milner's algorithm [DM82, Mil78] and is augmented to handle ambiguous quotations. A quotation that can be parsed ambiguously can be compared to an overloaded symbol. The meaning of an overloaded symbol is resolved by type information from its context. Similarly, the type of a quotation is determined from the context.

3.1 The type checker

A polymorphic type checker usually [Car87, DM82, Mil78, Jon87] takes a type environment and the expression that will be type checked as argument and returns a type substitution and a type expression:

$$\texttt{usualTCheckExp} :: \texttt{TypeEnv} \rightarrow \texttt{Expr} \rightarrow (\texttt{Subst} \times \texttt{Type})$$

A variant is described in [Rea89] where the substitution is also given as argument. The type environment contains assumptions about the type of identifiers. The substitution is a map from type variables to types. We will denote type variables $'a$, $'b$,... and substitutions as $[\tau_1/'a_1, \ldots, \tau_n/'a_n]$. Two important functions in our type checker involving substitutions are:

$$\texttt{unify} :: \texttt{Type} \rightarrow \texttt{Type} \rightarrow \texttt{Subst}$$
$$\texttt{combine} :: \texttt{Subst} \rightarrow \texttt{Subst} \rightarrow \texttt{Subst}$$

We will only explain these functions briefly and omit all details. The function `unify` takes two type expressions as argument and unifies them. The result is a substitution which when applied to the two type expressions make them equal. We have for example:

$$\texttt{unify}\ (\texttt{'}a \times \texttt{Int})\ (\texttt{List}\,\texttt{'}b \times \texttt{'}c) \;=\; [\texttt{List}\,\texttt{'}b\,/\texttt{'}a\,,\ \texttt{Int}/\texttt{'}c\,]$$

If the two types are not unifiable then `unify` fails and gives a special "bad" substitution as result. The function `combine` combines two substitutions. We have for example:

$$\texttt{combine}\,[\texttt{List}\,\texttt{'}b\,/\texttt{'}a\,]\,[\texttt{Int}/\texttt{'}b\,,\ \texttt{Bool}/\texttt{'}c\,] \;=\; [\texttt{List Int}/\texttt{'}a\,,\,\texttt{Int}/\texttt{'}b\,,\,\texttt{Bool}/\texttt{'}c\,]$$

We will sometimes use juxtaposition as a shorthand for the function `combine`. It is important to note that `combine` may fail and give a special "bad" substitution as result. An example:

$$\texttt{combine}\,[\texttt{Int}/\texttt{'}a\,]\,[\texttt{Bool}/\texttt{'}a\,] = \text{Bad Substitution}$$

There are variants of these two functions. Hancock [Jon87, chapter 9] describes for example a type checker where the function `unify` takes also a substitution as argument.

Our type checker takes an additional argument compared to the usual ones, the *expected type* of the expression. If nothing is known about the type then the expected type is simply a type variable. The main reason for the use of expected types is that explicit typing of expressions will improve the execution speed of the type checker. Since expressions can have different types, the type checker returns a list of possibilities to type check the expression. Each possibility consists of a substitution and a list of the parsed and translated quotations that appear in the expression. In a later phase of the compilation these parsed expressions will be substituted for the quotations.

```
TCheckExp :: TypeEnv → Expr → Expected type →
             List (List Expr × Subst)
```

The type checker for patterns returns also a variable-type association, the variables that are bound by the pattern and their types.

```
TcheckPat :: TypeEnv → Pattern → Expected type →
             List (List Pattern × Vbind × Subst)
```

Let us illustrate how the type checker works by some examples. In the examples we use the conctypes:

```
conctype Var  = [|x|]  +  [|y|]  +  [|z|]

conctype E =  [|<Int>|]
           +  [|<Var>|]
           +  [|<E>+<E>|]

conctype B =  [|<Bool>|]
           +  [|<Var>|]
           +  [|<E>=<E>|]
```

Both `Int` and `Bool` are the predefined ordinary types for integers and booleans in LML.

Let us start with the expression:

```
(\x.[|^x|]) 3
```

Suppose that there is no information about the type of this expression from the context so the type checker gets a type variable $'a$ as expected type. For applications like this, we first type check the argument. Regardless of the expected type of the whole application we do not know anything about the argument so it is type checked with a fresh type variable as expected type. The result of type checking `3` with an expected type $'b$ is the substitution [`Int`/$'b$]. Now this information is used when type checking:

```
\x.[|^x|]
```

Since the type of the whole application was expected to be $'a$ and the type of the argument is `Int` we know that the type of the abstraction must be `Int` $\rightarrow$ $'a$. Thus, this is the expected type of the abstraction. This leads to that

```
[|^x|]
```

is type checked with the expected type $'a$ in a type environment where `x` has type `Int`. Type checking `[|^x|]` means that it is parsed and since `x` is of type `Int` there is only one possible way to parse it. The substitution that is given as result is [`E`/$'a$], and from that we get that the type of the abstraction is `Int` $\rightarrow$ `E`. Note that without use of the expected type the abstraction has four possible types:

1. `Int` $\rightarrow$ `E`
2. `Var` $\rightarrow$ `E`
3. `Bool` $\rightarrow$ `B`
4. `Var` $\rightarrow$ `B`

The whole expression finally gets type `E`. The translated quotation is also given as result, that is `E-0 3` since `E-0` is the associated constructor for the production `E = [|<Int>|]` that was used in the parsing.

Let us now consider a slightly different application, namely:

```
(\x.[|^x|]) [|z|]
```

The argument has in this case three possible types: `Var`, `E` and `B`, and the result of type checking it with the expected type $'b$ is as follows:

Type checking `[|z|]` with expected type $'b$.			
possibility	translated quotations	substitution	type
1	`Var-2`	[`Var`/$'b$]	`Var`
2	`E-1 Var-2`	[`E`/$'b$]	`E`
3	`B-0 Var-2`	[`B`/$'b$]	`B`

Which type should we use as expected type when type checking the abstraction? Our solution is to use the type variable $'b$, so the abstraction is this time type checked with the expected type $'b$ $\rightarrow$ $'a$. The quotation `[|^x|]` is type checked with an expected type $'a$ in a type environment where `x` has type $'b$. The parsing of `[|^x|]` is now ambiguous:

Type checking `[|^x|]` with expected type $'a$ in a type environment where `x` has type $'b$.			
possibility	translated quotations	substitution	type
1	`E-0 x`	[`E`/$'a$, `Int`/$'b$]	`E`
2	`E-1 x`	[`E`/$'a$, `Var`/$'b$]	`E`
3	`B-0 x`	[`B`/$'a$, `Bool`/$'b$]	`B`
4	`B-1 x`	[`B`/$'a$, `Var`/$'b$]	`B`

The same possibilities remain for the abstraction:

Type checking `\x.[|^x|]` with expected type $'b \rightarrow 'a$.			
possibility	translated quotations	substitution	type
1	`E-0 x`	[`E`/$'a$, `Int`/$'b$]	`Int` $\rightarrow$ `E`
2	`E-1 x`	[`E`/$'a$, `Var`/$'b$]	`Var` $\rightarrow$ `E`
3	`B-0 x`	[`B`/$'a$, `Bool`/$'b$]	`Bool` $\rightarrow$ `B`
4	`B-1 x`	[`B`/$'a$, `Var`/$'b$]	`Var` $\rightarrow$ `B`

For the application we check which substitutions that are possible to combine. In this case it is possible to combine possibility 1 from the type checking of `[|z|]` with possibility 2 and 4 from the type checking of `\x.[|^x|]`:

Type checking `(\x.[|^x|]) [|z|]` with expected type $'a$.			
possibility	translated quotations	substitution	type
1	`E-1 x, Var-2`	[`E`/$'a$]	`E`
2	`B-0 x, Var-2`	[`B`/$'a$]	`B`

This ambiguity will hopefully be resolved later. Otherwise an error message will be given. One possibility to give more type information is to explicitly type the expression:

```
((\x.[|^x|]) [|z|]) : E
```

Of course, an explicitly typed expression has the explicit type as expected type, so the expected type of the application is in this case `E`. The argument `[|z|]` is still type checked with a type variable $'b$ as expected type and is of course still ambiguous in the same way. But the abstraction is now type checked with the expected type $'b \rightarrow$ `E` and then the quotation `[|^x|]` with the expected type `E`. The results are then:

Type checking `[|^x|]` with expected type `E` in an type environment where `x` has type $'b$.			
possibility	translated quotations	substitution	type
1	`E-0 x`	[`E`/$'a$, `Int`/$'b$]	`E`
2	`E-1 x`	[`E`/$'a$, `Var`/$'b$]	`E`

Type checking `\x.[|^x|]` with expected type $'b \rightarrow$ `E`.			
possibility	translated quotations	substitution	type
1	`E-1 x`	[`Var`/$'b$]	`Var` $\rightarrow$ `E`
2	`E-0 x`	[`Int`/$'b$]	`Int` $\rightarrow$ `E`

Finally after combining the substitutions in the application there is only one possibility left.

Type checking (\x.[|^x|]) [|z|] with expected type E.			
possibility	translated quotations	substitution	type
1	E-1 x, Var-2	$\emptyset$	E

A more complicated expression to type check is the case-expression.

```
case ce in
   p1   :   e1
   ⋮
|| pn   :   en
end
```

During the type checking of a case-expression a *case possibility list* consisting of translated quotations, a substitution, a pattern type and an expression type is used. After each clause $p_i : e_i$ is type checked this list is updated and all impossible combinations are thrown away. When the case possibility list contains only one element, the types are used as expected types in the type checking of the next pattern p_i and expression e_i. Then also the substitution is applied to the type environment.

The initial case possibility list is the possibilities resulting from type checking the choice-expression together with the expected type of the whole case-expression. In most cases the choice-expression has only one type checking possibility.

If a pattern p_i is type checked unambiguously then the expression e_i is type checked in a type environment in which the variables in the pattern have their only possible types. If a pattern is type checked ambiguously then the corresponding expression is type checked in a type environment in which the variables have fresh type variables as types. Information about the possible types of the type variables are held in the substitutions of the possibilities of type checking the pattern. Let us consider the following abstraction.

```
\x . case x in
         [|^b|]        : b
     ||  [|^e1=^e2|]   : Eval e1 = Eval e2
     end
```

We assume that `Eval` has type `E` $\rightarrow$ `Int` in the type environment and that the expected type is $'a \rightarrow 'b$. Irrelevant parts of the substitutions will sometimes be omitted in the examples. The initial case possibility list has only one element:

Initial case possibility list for the case expression.				
possibility	translated quotations	substitution	pattern type	expression type
1	None	$\emptyset$	$'a$	$'b$

Since the first pattern `[|^b|]` is ambiguous and the right hand side does not resolve the ambiguity, the case possibility list after the first clause contains several possibilities:

Case possibility list after type checking clause 1.				
possibility	translated quotations	substitution	pattern type	expression type
1	`E-0 b`	[`Int`/$'b$, `E`/$'a$]	`E`	`Int`
2	`E-1 b`	[`Var`/$'b$, `E`/$'a$]	`E`	`Var`
3	`B-0 b`	[`Bool`/$'b$, `B`/$'a$]	`B`	`Bool`
4	`B-1 b`	[`Var`/$'b$, `B`/$'a$]	`B`	`Var`

Since there are more than one possibility, fresh type variables are used as expected types when type checking the second clause. The second pattern is also ambiguous and can be parsed in nine different ways. Two of the parsing possibilities are `B-2 (E-0 e1) e2` and `B-2 (E-1 e1) (E-0 e2)`. The right hand side resolves the ambiguity and then the obtained possibility is used to resolve ambiguities in the case possibility list which after the second clause will contain only one element:

Case possibility list after type checking clause 2.				
possibility	translated quotations	substitution	pattern type	expression type
1	`B-0 b`, `B-2 e1 e2`	[`Bool`/$'b$, `B`/$'a$]	`B`	`Bool`

Since the list now contains only one element we know the type of both the pattern and the expression. If there had been more clauses, `B` would have been used as expected type when type checking the patterns and `Bool` when type checking the expressions.

From the substitution that results from type checking the case-expression, we deduce that the abstraction has type `B` $\rightarrow$ `Bool`.

3.2 Parsing of quotations

The parsing of a quotation expression has two aims, to determine the type of the quotation and to translate it to an element of an ordinary type. The parsing of a quotation pattern has an additional aim, to assign types to the variables in the pattern.

The original algorithm of Earley takes a grammar and an input string as argument. Our version takes also a type environment and an expected type as argument. The type environment is needed to type check antiquotations and the expected type is used to restrict the number of conctypes to be checked as possible types. The parsing of a quotation can be seen as the problem to determine which conctypes that can derive the quotation. Thus, all conctypes can be seen as possible start symbols. Then the expected type restricts the number of start symbols.

The result of the parsing is a list of possibilities to parse the quotation. Each possibility consists of the translated expression and a type substitution. A possible type of the quotation is obtained by applying the substitution of a possibility on the expected type.

```
ExpEarley :: TypeEnv → Grammar → Quotation Expression →
             Expected Type → List (Expr × Subst)
```

The generalized version works, as the original algorithm, by processing a set of item sets, one for each input symbol. An item contains a production such that we currently are trying to recognize its right hand side as a part of the input string. A dot in the production indicates how much of the right hand side we have recognized so far. An item also contains a pointer back to the position in the input string at which we began to look for the production. Besides the dotted production and the backward pointer, the items in

our version contain also a substitution and the associated constructor for the production. When parsing patterns, an item also contains a variable-type association. An item have the following components:

$\langle c,\ \mathtt{A} ::= es \bullet \beta, s, i \rangle$

where	c	is the associated constructor for the production,
	A	is the left hand side conctype of the dotted production
	es	a sequence of parsed substrings, translated to LML-expressions,
	β	is a sequence of types and terminals,
	s	is a substitution and
	i	is the backward pointer.

When the dot is last and the whole production has been recognized, the constructor, c, is applied to the expression sequence, es, and a new expression, $c(es)$, is built.

In the original algorithm, the processing starts in item set 0 with one item for each alternative of the start symbol, with the dot first and an endmark last. In the generalized version, items for alternatives of all conctypes that can be unified with the expected type are initially added to item set 0. As usual, the dot is first and a new endmark last in the right hand side. The substitution in an initial item is the one obtained from the unification of the conctype with the expected type. The substitution is applied to the types in the productions. Let us illustrate this with some examples. Suppose that we have the conctypes:

```
conctype Sym     = [|#|]  +  [|&|]

conctype Tree *a = [|o|]
                 + [|<Tree *a>-<*a>-<Tree *a>|]
```

As a first example, a quotation will be parsed with a type variable $'e$ as expected type. This means that nothing is known about the type of the quotation from the context. Then items for all productions are added to the initial item set:

Initial item set for expected type $'e$.

Sym-0	Sym ::= •# ⊣	[Sym/$'e$]	0
Sym-1	Sym ::= •& ⊣	[Sym/$'e$]	0
Tree-0	Tree $'a1$ ::= •o ⊣	[Tree $'a1$ /$'e$]	0
Tree-1	Tree $'a2$ ::= •Tree $'a2$ - $'a2$ - Tree $'a2$ ⊣	[Tree $'a2$ /$'e$]	0

As a second example, we have an expected type Tree $'b$. The conctype Sym is not unifiable with it, and only items for productions of the conctype Tree $'a$ are added to the initial item set:

Initial item set for expected type Tree $'b$.

Tree-0	Tree $'b$ ::= •o ⊣	∅	0
Tree-1	Tree $'b$ ::= •Tree $'b$ - $'b$ - Tree $'b$ ⊣	∅	0

Finally, as a third example, we have an expected type Tree Int. The conctype Tree $'a$ is unifiable with it and in such case we add instantiated items:

Initial item set for expected type Tree Int.

Tree-0	Tree Int ::= •o ⊣	∅	0
Tree-1	Tree Int ::= •Tree Int - Int - Tree Int ⊣	∅	0

As in the original algorithm, the parsing proceed by the operations *predict, scan* and *complete*, and more item sets are built. However, the operations are changed a bit to handle the substitutions in the items and antiquotations in the input string. The changes are shortly:

predict As usual, the operation predict is applicable when the dot is in front of a type τ. Instead of just adding items for all alternatives of τ we add new items for all conctypes which are unifiable with τ.

scan The operation scan is usually applicable when the dot is in front of a terminal and the next input symbol is the same terminal. In our version, the operation scan is applicable also when the dot is in front of a type and the next input symbol is an antiquotation. For every possible type of the antiquotation which can be unified with the type after the dot that all involved substitutions can be combined, a new item is added to the next item set. We call this variant of the operation scan *antiquotescan*.

complete It is not enough that an item, in the item set pointed to, once has predicted the completion item. It must also be possible to unify the type after the dot with the recognized type and combine all involved substitutions.

As soon as a new substitution is obtained, it is applied to the types in the sequence after the dot. The reason for this is to use as much information as possible during the parsing of the rest of the item. As an example, consider the following item:

`Tree-1` `Tree` $'a$ ::= • `Tree` $'a$ `-` $'a$ `-` `Tree` $'a$ s i

Suppose that we obtain a substitution [`Int`/$'a$] when an antiquotescan is performed. This information is used when we try to recognize the rest of the item. Thus, the item that is added to the next item set is:

`Tree-1` `Tree` $'a$ ::= e • `-` `Int` `-` `Tree` `Int` s[`Int`/$'a$] i

The domain of the substitution in an item can be greater than the set of type variables in the production due to antiquotations. Consider for example the following expression:

```
\x.[|^(x+3).....|]
```

When type checking the abstraction, the variable `x` will be assigned a type. If the expected type of the abstraction is not of function type or immediately leads to a type error, this type is a type variable, say $'a$. Thus, the antiquotation, `(x+3)`, will be type checked in a type environment where `x` has type $'a$. As a result of type checking the antiquotation we obtain a substitution [`Int`/$'a$]. This substitution will appear in the items for which the parsing succeed.

All antiquotations are type checked before an item set is processed. Thus the input symbol when processing an item set can either be a terminal symbol or an antiquotation with an associated list of possibilities obtained from the type checker. Each possibility consists of a list of translated quotations, a substitution and a type of the antiquotation. An antiquotation is always type checked with a type variable as expected type. There are

some disadvantages to type check an antiquotation as a part of the operation antiquotescan and use the type after the dot as expected type. Since antiquotescan can be applicable to more than one item, an antiquote can in such case be type checked more than once. This is less efficient. Moreover, if the type checker in such case cannot find a type for an antiquote, there need not be a type error in the program. There may exist other items for which the operation antiquotescan is applicable and other possible expected types.

When finished, the possible types of the quotation expression are all conctypes that appear on the left hand side in the last item set. Since the quotation can be ambiguous there can be more than one. Suppose that the following items are in the last item set after a parsing:

	Last item set		
A-2	A ::= $es \bullet$	s	0
B-4	B ::= $es' \bullet$	s'	0

The constructors are applied to the expression sequences and as result from the parsing we obtain:

Result from a parsing.			
possibility	translated quotations	substitution	type
1	A-2(es)	s	A
2	B-4(es')	s'	B

More detailed information about the operations can be seen from the following table. Besides the operations predict, complete, scan and antiquotescan, there are other operations to handle list constructions in the conctypes. These will be described in section 3.3.2.

Operations on items in item set i		
Operation	**Is applied when**	**Effect**
Predict	The dot is in front of a type, $\langle c,$ A ::= $es \bullet$ B$\beta, s, j\rangle$.	For every production, B′ ::= γ, such that B can be unified with B′ yielding substitution s', add the item $\langle c',\ s'$ B′ ::= $\bullet s'\ \gamma, \emptyset, i\rangle$ to item set i. The constructor c' is associated with the production. Introduce fresh type variables in the predicted items.
Scan	The dot is in front of a terminal, $\langle c,$ A ::= $es \bullet$ a$\beta, s, j\rangle$, and the input symbol at position $i+1$ is a.	Add the item $\langle c,$ A ::= $es \bullet \beta, s, j\rangle$ to item set $i+1$.
Antiquotescan	The dot is in front of a type, $\langle c,$ A ::= $es \bullet$ B$\beta, s, j\rangle$, and the input symbol at position $i+1$ is an antiquotation e with possible parsings, es', substitutions, s', and types, τ'.	Unify B with all τ'_k yielding substitutions s''_k. For possible combinations s, s'_k and s''_k into s'''_k, add the item $\langle c,$ A ::= $es@e[es'_k] \bullet s'''_k\ \beta, s'''_k, j\rangle$ to item set $i+1$.

Operations on items in item set i		
Operation	**Is applied when**	**Effect**
Complete	The dot is last, $\langle c,\ \texttt{A} ::= es\bullet, s, j\rangle$.	Consider items $\langle c',\ \texttt{B} ::= es' \bullet \texttt{A}'\gamma, s', k\rangle$ in item set j, that has predicted the item. Unify A with A′ yielding substitution s''. For all possible combinations s, s' and s'' into s''', add the item $\langle c',\ \texttt{B} ::= es'@c(es) \bullet s'''\gamma, s''', k\rangle$ to item set i. Use fresh type variables in the item before unification.

$e[es]$ denotes the expression e where the expressions in the expression sequence es have been substituted for the quotation occurrences in e.
$s\ \tau$ denotes the result of applying the substitution s on the type τ.
$s\ \alpha$ denotes the result of applying the substitution s on all types in α

3.2.1 Example

Suppose that we (only) have the following conctype:

```
conctype L *a = [||]  +  [|<*a>;<L *a>|]
```

Let us parse the following quotation with expected type $'e$:

```
[|^(3);^(4);|]
```

Initially we have:

Item set 0

L-0	L $'a1$ ::= • ⊣	[L $'a1/'e$]	0
L-1	L $'a2$ ::= • $'a2$; L $'a2$ ⊣	[L $'a2/'e$]	0

Notice that we have introduced fresh type variables in every item. The operation predict is applicable to the second item and since the dot is in front of a type variable we add items for all conctypes:

Item set 0, cont

L-0	L $'a3$ ::= •	∅	0
L-1	L $'a4$ ::= • $'a4$; L $'a4$	∅	0

Note that the items are added with the empty substitution. We ignore that the operation complete is applicable to the third item since it will not lead to anything. The operation antiquotescan is applicable to the two items with the dot in front of a type variable and since the next input symbol is an antiquotation ^(3) with type Int we get in item set 1:

Item set 1

L-1	L $'a2$ ::= 3 • ; L Int ⊣	[L Int$/'e$, Int$/'a2$]	0
L-1	L $'a4$ ::= 3 • ; L Int	[Int$/'a4$]	0

Note that the obtained substitution [Int/'a] is applied to the types after the dot in the item. Now, the operation scan is applicable to both items:

Item set 2

L-1	L 'a2 ::= 3 • L Int ⊣	[L Int/'e, Int/'a2]	0
L-1	L 'a4 ::= 3 • L Int	[Int/'a4]	0

Terminal symbols just disappear. When the dot, as here, is in front of an instantiated type, the result of the operation predict is that only that instance of the type is added:

Item set 2, cont

L-0	L Int ::= •	∅	2
L-1	L Int ::= • Int ; L Int	∅	2

Again we ignore the operations complete. In item set 2, the operation antiquotescan fails to move the dot over the type L Int since L Int cannot be unified with Int which is the type of the antiquotation ^(4). Instead, the operation antiquotescan is applicable to the item with the dot in front of the type Int:

Item set 3

L-1	L Int ::= 4 • ; L Int	∅	2

After applying the operations scan and predict:

Item set 4

L-1	L Int ::= 4 • L Int	∅	2
L-0	L Int ::= •	∅	4
L-1	L Int ::= • Int ; L Int	∅	4

The operation complete is now applicable to the second item in item set 4.

Item set 4, cont

L-1	L Int ::= 4 L-0 •	∅	2

Again is the operation complete applicable.

Item set 4, cont

L-1	L 'a2 ::= 3 (L-1 4 L-0) • ⊣	[L Int/'e, Int/'a2]	0
L-1	L 'a4 ::= 3 (L-1 4 L-0) •	[Int/'a4]	0

As a result of applying the operation complete on the last item some items from item set 0 are added but then nothing is applicable to them. After applying the operation scan to the item in item set 4 with the dot in front of the endmark symbol we are ready.

Last item set

L-1	L 'a2 ::= 3 (L-1 4 L-0) •	[L Int/'e, Int/'a2]	0

The result from the parsing is:

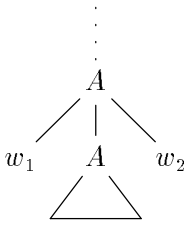
Result from the parsing of [|^(3);^(4);|] with expected type 'e.			
possibility	translated quotations	substitution	type
1	L-1 3 (L-1 4 L-0)	[L Int/'e, Int/'a2]	L Int

3.3 Parsing of special constructions

As described in section 2 there are special constructs for sequences and precedences and in part II we described "forgotten" productions. In this section we describe how these constructs are handled in the parser.

3.3.1 Forgotten productions

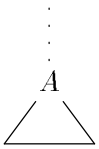
If the notation with double square brackets, for example `[||(<E>)||]`, is used for a production when defining a conctype it means that this production will not remain in the representation. Suppose a normal parse tree for some sentence is:



If the production

$$A ::= w_1\ A\ w_2$$

where w_1 and w_2 are sequences of terminals, is marked to be forgotten, then the parse tree we want for the same sentence is:



Translating this from parse trees to quotation expressions means that an associated constructor to a forgotten production must not appear anywhere in the translated expressions. Suppose that a translated expression without use of forgotten productions is:

```
B-2 (A-2 (A-4 (C-3 C-2)))
```

The same expression, where the constructor `A-2` is associated to a forgotten production, is:

```
B-2 (A-4 (C-3 C-2))
```

This effect is achieved during parsing by adding a boolean component to each item in Earley's algorithm. When a new item is constructed from a production in the grammar this component is set to true if the production would be forgotten. At the operation complete, the constructor is not applied to the list of parsed expressions if the boolean component is true.

3.3.2 Sequences

Recall that we with the sequence notation in conctypes can define a nonempty sequence of `A`:s separated by ; with:

```
{<A>;...}+
```

Without sequence notation we can introduce a nonterminal As which generates the same sequence:

$$\begin{array}{lcl} As & ::= & A \ ; \ As \\ & | & A \end{array}$$

The handling of sequences in our Earley-parser is closely related to this correspondence. As an example, let us process the following item:

$$c \qquad \mathtt{B} ::= es \bullet \{\mathtt{<A>;...}\}\mathtt{+}^{n} \ \alpha \qquad s \qquad i$$

The effect of processing it, is that the item is replaced by two new items.

$$\begin{array}{lll} c & \mathtt{B} ::= es \bullet \mathtt{A} \ ; \ \{\mathtt{<A>;...}\}\mathtt{+}^{n+1} \ \alpha & s \quad i \\ c & \mathtt{B} ::= es \bullet \mathtt{A} \ \mathsf{ENDE}(n+1) \ \alpha & s \quad i \end{array}$$

The number n denotes the number of times the replacement has occurred. This counter is used when the dot is in front of the new symbol $\mathsf{ENDE}(n)$:

$$c \qquad \mathtt{B} ::= es \ e_1 \ \cdots \ e_n \bullet \mathsf{ENDE}(n) \ \alpha \qquad s \qquad i$$

Now a sequence is parsed and the parsed elements of the sequence $e_1 \cdots e_n$ are before the dot. The effect of processing the item is that an LML-list containing the elements $e_1 \cdots e_n$ is constructed:

$$c \qquad \mathtt{B} ::= es \ (e_1.e_2.\cdots.e_n.nil) \bullet \alpha \qquad s \qquad i$$

Besides the replacements of sequence notations, *antiquotescan* may also be applicable when the dot is in front of a sequence element. Recall that the notation `{<A>;...}+` can be compared to a conctype that generates sequences of `A` and that the type is `List(A)`. Thus, the operation antiquotescan may be applicable to the item:

$$c \qquad \mathtt{B} ::= es \bullet \{\mathtt{<A>;...}\}\mathtt{+}^{n} \ \alpha \qquad s \qquad i$$

The requirements are that the next input symbol is an antiquotation e which has a type that is unifiable with `List(A)` and the obtained substitution s' is combinable with s. The result is that the following item is added to the next item set:

$$c \qquad \mathtt{B} ::= es \ e \bullet \mathsf{ENDL}(n) \ \alpha \qquad ss' \qquad i$$

The difference between the symbols $\mathsf{ENDE}(n)$ and $\mathsf{ENDL}(n)$ is that $\mathsf{ENDL}(n)$ regard the last element in the sequence of parsed elements as a list instead of an element of a list. Thus, the result of processing an item:

$$c \qquad \mathtt{B} ::= es \ e_1 \cdots e_n \bullet \mathsf{ENDL}(n) \ \alpha \qquad s \qquad i$$

is that it is replaced by the following item:

$$c \quad \texttt{B} ::= es\ (e_1.e_2.\cdots.e_n) \bullet \alpha \quad s \quad i$$

The notation for possibly empty sequences `{<A>;...}*` corresponds to:

$$\begin{array}{lcl} As' & ::= & \epsilon \\ & | & As \\ As & ::= & A\ ;\ As \\ & | & A \end{array}$$

The nonterminal As' defines possibly empty sequences of A separated by semicolon. Note that the nonterminal As defines, as before, nonempty sequences of A separated by semicolon. Thus, the effect of processing an item:

$$c \quad \texttt{B} ::= es \bullet \{\texttt{<A>;...}\}\texttt{*}\ \alpha \quad s \quad i$$

is that it is replaced by the two new items:

$$\begin{array}{lll} c & \texttt{B} ::= es\ nil \bullet \alpha & s \quad i \\ c & \texttt{B} ::= es \bullet \{\texttt{<A>;...}\}\texttt{+}^0\ \alpha & s \quad i \end{array}$$

The first item corresponds to the epsilon production $As' ::= \epsilon$ and the second to the production $As' ::= As$.

3.4 Parsing with precedences

To handle precedence rules, as described in section 2.2, new components are added to the items in Earley's algorithm, and the algorithm is augmented to use these components to throw away undesirable parse trees. How this is done is described in part V of this thesis.

4 Implementation

An implementation of the conctypes as described in this paper has been done in the functional language LML [AJ87, AJ89], and is now available in the LML-distribution[5] . All examples in this paper work in the implementation.

5 Benchmarks

The compilation (but not the execution) of an LML-program containing conctypes and quotations is slower than the corresponding program containing ordinary datatypes and values. With the corresponding program we mean a program where each conctype is replaced by an ordinary type and each quotation is replaced by its corresponding element. The difference is greater the more and longer quotations that is used. In the table below, we give compilation times for some example programs. In each example we give the time for a program using conctypes and a corresponding program without conctypes. The compilation is done on a SUN sparcstation and times are in seconds.

[5] Available by anonymous ftp from ftp@cs.chalmers.se.

	Program	Conctype version	Normal version
1	SubPascal + quicksort element	20	5
2	SubPascal + interpreter	41	24
3	SubPascal + interpreter + quicksort element	60	28
4	Conctype `expr` + interpreter + fac element	11	5

Examples 1-3 are variants on the SubPascal-program given in appendix A. Example 4 is the conctype `expr` given in part I of this thesis together with the interpreter and the quotation containing the fac-function. All named functions are explicitly typed. Compiling a program without explicit typing takes a little bit longer time. Example 2 without all unnecessary typing takes 50 seconds to compile.

Comparing a conctype version with a "normal" version is a bit unfair if the program contains a big quotation. No one would for example dare to use an ordinary element corresponding to the quotation containing the quicksort procedure. (Our example program is automatically generated.) A parser must be used and the time for compiling a parser is not included.

The parser for quotations is an extension of Earley's algorithm and Earley's algorithm is not the most efficient parsing method. By using another parsing method, the compilation may be more efficient. Also, a more sophisticated lexical analyzer may improve the compilation times.

Appendix A

A conctype for a subset of Pascal

1 Introduction

In this appendix we define a subset of Pascal [Wir71] as a conctype. We call this subset of Pascal SubPascal. SubPascal is about of the same size as the subset of Pascal given in appendix A in the "dragon" book [ASU86], and allows nontrivial programs to be expressed. As an example, we give a quotation containing a quicksort procedure. We also give an interpreter for SubPascal in which we use pattern matching on the concrete syntax.

2 SubPascal

A program in SubPascal consists of a sequence of variable declarations, a sequence of procedure and function declarations, and a single compound statement. There are only two standard types, integer and boolean, and one type constructor, array, in which only standard types are allowable component types. Parameters to a function are only allowed to be of standard type and are passed by value. Parameters of standard type to a procedure are either passed by value or by reference and parameters of array type are passed by reference. The statements are the usual ones: assignment, procedure call, conditional statements and loops. Expressions are build from numbers, identifiers, several operators, array indexing, and function calls. The value of an expression can only be of standard type.

3 SubPascal as conctypes

Below we give conctypes that defines SubPascal. The conctype Program is startsymbol. We assume that the conctype `Id`, with elements like `[|read|]`, `[|j|]` and `[|quicksort|]` as well as the conctype `Number` with elements `[|0|]`, `[|1|]`, ... are already defined. Precedences are used to disambiguate the conctype `Expr`. The escape character `\` is used to give the next character its literal meaning.

```
and conctype Program
          = [|program <Id>({<Id>,...}+);
                <Declaration>{<Subprogramdeclaration>...}*<Compoundstatement>.
            |]
```

```
and conctype Declaration
              = [|var {<Dec>; ...}+; |]
              + [||]

and conctype Dec
              = [|{<Id>,...}+:<Type>|]

and conctype Type
              = [|<Standardtype>|]
              + [|array[<Number>..<Number>] of <Standardtype>|]

and conctype Standardtype
              = [|integer|]
              + [|boolean|]

and conctype Subprogramdeclaration
              = [|<Subprogramhead><Declaration><Compoundstatement>; |]

and conctype Subprogramhead
              = [|function <Id><Arguments> : <Standardtype>; |]
              + [|procedure <Id><Arguments>; |]

and conctype Arguments
              = [|({<ArgDec>;...}+)|]
              + [||]

and conctype ArgDec
              = [|var <Dec>|]                          -- call-by-reference
              + [|<Dec>|]                              -- call-by-value

and conctype Compoundstatement
              = [|begin {<Statement>; ...}+ end|]

and conctype Statement
              = [|<Id>:=<Expr>|]                       -- assignment
              + [|<Id>[<Expr>]:=<Expr>|]               -- assignment, array
              + [|<Id>|]                               -- procedure call, no arguments
              + [|<Id>({<Expr>,...}+)|]                -- procedure call
              + [|<Compoundstatement>|]
              + [|if <Expr> then <Statement> else <Statement>|]
              + [|if <Expr> then <Statement>|]
              + [|while <Expr> do <Statement>|]
              + [|repeat {<Statement>; ...}+ until <Expr>|]
```

```
and conctype Expr
              =             [|<Number>|]
              +             [|<Id>|]
              +             [||(<Expr>)||]
              +             [|<Id>[<Expr>]|]          -- array indexing
              +             [|<Id>({<Expr>,...}+)|]   -- function call
              + <           [|not <Expr>|]
              + < leftassoc [|<Expr> and <Expr>|]
              + =           [|<Expr> mod <Expr>|]
              + =           [|<Expr> div <Expr>|]
              + =           [|<Expr>*<Expr>|]
              + < leftassoc [|<Expr> or <Expr>|]
              + =           [|<Expr>+<Expr>|]
              + =           [|<Expr>-<Expr>|]
              + =           [|-<Expr>|]
              + < nonassoc  [|<Expr>\><Expr>|]
              + =           [|<Expr>\>=<Expr>|]
              + =           [|<Expr>\<<Expr>|]
              + =           [|<Expr>\<=<Expr>|]
              + =           [|<Expr>\<><Expr>|]
              + =           [|<Expr>=<Expr>|]
```

4 An example program

As an example of an element of the conctype `Program` we give a SubPascal program that reads ten numbers, stores them in an array, sorts the array using quicksort and prints the sorted numbers. The quicksort procedure is taken from [SW89]. The SubPascal program is a correct Pascal program and the text inside the brackets can be compiled by a Pascal compiler.

```
[|program p(input,output);
     var A:array[0..10] of integer;
         j:integer;

     procedure swap(var k,j:integer);
        var tmp:integer;
     begin
       tmp:=k;
       k:=j;
       j:=tmp
     end;
```

```
      procedure quicksort(l,r:integer);
         var j,k:integer;
      begin
       if r>l then begin
         if A[l]>A[r] then swap(A[l],A[r]);
         j:=l; k:=r;
         repeat
             repeat j:=j+1 until A[j]>=A[l];
             repeat k:=k-1 until A[l]>=A[k];
             if k>j then swap(A[j],A[k])
         until j>k;
         swap(A[l],A[k]);
         quicksort(l,k-1);
         quicksort(k+1,r)
       end
      end;

  begin
    j:=0;
    repeat
      read(A[j]);
      j:=j+1
    until j>10;
    quicksort(0,10);
    write(A[0],A[1],A[2],A[3],A[4],A[5],A[6],A[7],A[8],A[9],A[10])
  end.
|]
```

5 An interpreter

In this section we give an interpreter for SubPascal. The interpreter is an implementation of a denotational description of SubPascal. The functions in the interpreter are written using pattern matching on the concrete syntax.

Almost all functions are explicitly typed for readability and for making the compilation a bit faster. The only necessary typing is

- `(t:Type)` in `parKindArgDec`
- `(idl:List Id)` in `Prog`
- the first argument to `allocDec`

In the two first cases, the variables `t` and `idl` are not used and therefore the type checker cannot deduce a type. The third case must be typed since it cannot be deduced from the context whether the pattern denotes an element of type `Dec` or `ArgDec`.

We assume that the operators `&&`, `%%` etc are declared as infix and that the function `toInt :: Number → Int` is defined.

```
let rec type Value = I Int + B Bool

and type Location == Int
and type Memory == Location -> Value
and type State == (Memory # (List Int) # (List Int) # Location)
            --    Memory,  Input      ,  Output    , Highest not used Location

and type ParameterKind = callbyREF + callbyVAL

and type Parameter = REF Location    --  call-by-reference
                   + VAL Value       --  call-by-value

and type Kindofid = SIMPLE Location
                  + ARRAY Int Int Location
                        -- first and last index, first Location
                  + FUN   (List Parameter -> State -> Value)
                  + FUNID (List Parameter -> State -> Value) Location
                        -- in function body
                  + PROC  (List ParameterKind)
                          (List Parameter -> State -> State)

and type Environment == (Id -> Kindofid)

and initstate = ((\x.fail "Not defined "),[],[],0) : State
and putinput (m,_,os,h) is = (m,is,os,h) : State
and getoutput   (_,_,os,_) = os
and upd f x v = \y.if x=y then v else f y

and valof_S :: State -> Location -> Value
and valof_S  (m,_,_,_) loc = m loc

and upd_S :: State -> Location -> Value -> State
and upd_S (m,is,os,h) l v = (upd m l v,is,os,h)

and listof a = a.listof a

and initenv :: Environment
and initenv [|read|]  = PROC (listof callbyREF)
                             (revitlist (\(REF l).\(m,i.is,os,h).
                                                  (upd m l (I i),is,os,h)))
||  initenv [|write|] = PROC (listof callbyVAL)
                             (\vs.\(m,is,os,h).
                                let ns = map (\(VAL (I n)).n) vs
                                in (m,is,ns@os,h) )
||  initenv  _        = fail "Not defined "
```

```
and parKindArgDec :: ArgDec -> List ParameterKind
and parKindArgDec [|var ^ids :^(t:Type)|] = rept (length ids) callbyREF
||  parKindArgDec [|^ids :^(t:Type)|]     = rept (length ids) callbyVAL

and parKindArguments :: Arguments -> List ParameterKind
and parKindArguments [|(^args)|] = concmap parKindArgDec args
||  parKindArguments [||]        = []

and allocDec :: Dec -> ((List Parameter) # Environment # State) ->
                       ((List Parameter) # Environment # State)
/* Dec is either a variable declaration or a formal parameter declaration.
   The list of Parameters is nil for variable declarations.
   Returns an Environment and a State in which the variable declared
   identifiers are bound to new locations and storage for them are
   reserved and the parameterpassing is performed for parameter declarations.*/
and allocDec [|^ids:^t|] s =
      case t in
         [|array[^n1..^n2] of ^t1|]
             : let i1 = toInt n1 in
               let i2 = toInt n2 in
               let allocA id (REF loc.ps,env,w) =
                         (ps,upd env id (ARRAY i1 i2 loc), w)
                || allocA id ([],env,(m,is,os,h)) =
                         ([],upd env id (ARRAY i1 i2 h), (m,is,os,h+i2-i1+1))
               in
                   revitlist allocA ids s
      || [|^st|]
             : let allocS id (VAL v.ps,env,(m,is,os,h)) =
                         (ps,upd env id (SIMPLE h), (upd m h v,is,os,h+1) )
                || allocS id (REF loc.ps,env,w) =
                         (ps,upd env id (SIMPLE loc), w)
                || allocS id ([],env,(m,is,os,h)) =
                         ([],upd env id (SIMPLE h), (m,is,os,h+1) )
               in
                   revitlist allocS ids s
      end

and allocArgDec :: ArgDec -> ((List Parameter) # Environment # State) ->
                             ((List Parameter) # Environment # State)
and allocArgDec [|var ^dec|] s = allocDec dec s
||  allocArgDec [|^dec|] s     = allocDec dec s
```

```
and allocArg :: Arguments -> List Parameter -> Environment -> State ->
                (Environment # State)
/* the updated Environment and State after parameterpassing */
and allocArg [|(^argdecs)|] ps env w  =
                let (_,env1,w1) = revitlist allocArgDec argdecs (ps,env,w)
                in
                    (env1,w1)
||  allocArg [||] ps env w      = (env,w)

and D :: Declaration -> Environment -> State -> (Environment # State)
/* the updated Environment and State in which the declared identifiers
   are bound to new locations and storage for them are reserved */
and D [|var ^decs; |] env w  =
          let (_,env1,w1) =  revitlist allocDec decs ([],env,w)
          in
             (env1,w1)
||  D [||] env w = (env,w)

and C :: Statement -> Environment-> State -> State
/* the resulting State from the execution of a Statement in an
   Environment and a State */
and C [|^v:=^e|] env w = let loc = case env v in
                                      SIMPLE loc   : loc
                                   || FUNID f loc  : loc
                                   end
                         in
                             upd_S w loc (E e env w)

||  C [|^v[^e1]:=^e2|] env w = let (ARRAY i1 i2 l1) = env v in
                               let I n              = E e1 env w in
                               let nv               = E e2 env w in
                                  if n<i1 | n>i2
                                  then fail "index out of range (assign)"
                                  else upd_S w (l1+n-i1) nv

||  C [|^q|] env w = let PROC [] p = env q     -- procedure call
                     in
                        p [] w
```

```
||  C [|^q(^ps)|] env w =                        -- procedure call
          (let (PROC pK p) = env q
           in
              p (map2 f pK ps) w
                where f callbyREF [|^id|]
                          = REF (case env id in
                                    SIMPLE loc      : loc
                                  || ARRAY i1 i2 loc : loc
                                  end )
                      || f callbyREF [|^id[^e]|]
                          = REF(let (ARRAY i1 i2 l1) = env id in
                                let I n              = E e env w in
                                   if n<i1 | n>i2
                                   then fail "index out of range (proccall) "
                                   else (l1+n-i1)
                               )
                      || f callbyVAL e
                          = VAL (E e env w)
          )

||  C [|^cc|] env w = CC cc env w

||  C [|if ^(e1) then ^(c1) else ^c2|] env w =
          case E e1 env w in
             B true  : C c1 env w
          || B false : C c2 env w
          end

||  C [|if ^(e1) then ^(c1)|] env w =
          case E e1 env w in
             B true  : C c1 env w
          || B false : w
          end

||  C [|while ^(e) do ^c|] env w =
        let rec while w1 = case E e env w1 in
                              B true  : while (C c env w1)
                           || B false : w1
                           end
        in
            while w
```

```
||  C [|repeat ^(cs) until ^e|] env w =
          let rec repeat w1 = let w2 = revitlist (\c.C c env) cs w1
                              in case E e env w2 in
                                    B true  : w2
                                 || B false : repeat w2
                                 end
          in
              repeat w

and CC :: Compoundstatement -> Environment -> State -> State
and CC [|begin ^(cs) end|] env w = revitlist (\c.C c env) cs w

and SD :: Subprogramdeclaration -> Environment -> Environment
/* an updated Environment in which a function or procedure identifier
   is bound to its meaning */
and SD [|function ^fid^(args) : ^t; ^d ^cc; |] env =
         let rec f ps (m,is,os,h) =
                   let env1 = upd env fid (FUNID f h) in
                   let (env2,w2) = allocArg args ps env1 (m,is,os,h+1) in
                   let (env3,w3) = D d env2 w2 in
                   let w4 = CC cc env3 w3 in
                      valof_S w4 h
         in
             upd env fid (FUN f)

||  SD [|procedure ^pid^args; ^d ^cc; |] env =
         let pK = parKindArguments args in
         let rec p ps w =
              let (env1,w1) = allocArg args ps env w  in
              let (env2,w2) = D d env1 w1 in
                 CC cc (upd env2 pid (PROC pK p)) w2
         in
             upd env pid (PROC pK p)

and (B b1) &&  (B b2) = B (b1 & b2)
and (I n1) %%  (I n2) = I (n1 % n2)
and (I n1) //  (I n2) = I (n1 / n2)
and (I n1) **  (I n2) = I (n1 * n2)
and (B b1) ||| (B b2) = B (b1 | b2)
and (I n1) ++  (I n2) = I (n1 + n2)
and (I n1) sub (I n2) = I (n1 - n2)
and (I n1) >>  (I n2) = B (n1 > n2)
and (I n1) >>= (I n2) = B (n1 >= n2)
and (I n1) <<  (I n2) = B (n1 < n2)
and (I n1) <<= (I n2) = B (n1 <= n2)
and neg (I n) = (I (-n))
```

```
and E :: Expr -> Environment -> State -> Value
/* the Value of an Expr in an Environment and a State */
and E [|^n|] env w = I (toInt n)

||  E [|^v|] env w = case env v in
                        SIMPLE loc  : valof_S w loc
                     || FUNID f loc : f [] w
                     || FUN f       : f [] w
                     end

||  E [|^v[^e]|] env w = let (ARRAY i1 i2 l1) = env v in
                           case E e env w in
                             (I n) : if n<i1 | n>i2
                                     then fail "index out of range "
                                     else valof_S w (l1+n-i1)
                           end

 -- function call
||  E [|^v(^ps)|] env w = let vs = map (\e.VAL (E e env w)) ps
                          in
                             case env v in
                                FUN f     :  f vs w
                             || FUNID f h :  f vs w
                             end

||  E [|not ^e|]        env w = let B b = E e env w in B (not b)
||  E [|^(b1) and ^b2|] env w = (E b1 env w) &&  (E b2 env w)
||  E [|^(e1) mod ^e2|] env w = (E e1 env w) %%  (E e2 env w)
||  E [|^(e1) div ^e2|] env w = (E e1 env w) //  (E e2 env w)
||  E [|^e1*^e2|]       env w = (E e1 env w) **  (E e2 env w)
||  E [|^(b1) or ^b2|]  env w = (E b1 env w) ||| (E b2 env w)
||  E [|^e1+^e2|]       env w = (E e1 env w) ++  (E e2 env w)
||  E [|^e1-^e2|]       env w = (E e1 env w) sub (E e2 env w)
||  E [|-^e|]           env w = neg (E e env w)
||  E [|^e1>^e2|]       env w = (E e1 env w) >>  (E e2 env w)
||  E [|^e1>=^e2|]      env w = (E e1 env w) >>= (E e2 env w)
||  E [|^e1\<^e2|]      env w = (E e1 env w) <<  (E e2 env w)
||  E [|^e1\<=^e2|]     env w = (E e1 env w) <<= (E e2 env w)
||  E [|^e1\<>^e2|]     env w = B ((E e1 env w) ~= (E e2 env w))
||  E [|^e1=^e2|]       env w = B ((E e1 env w) = (E e2 env w))
```

```
and Prog :: Program -> List Int -> List Int
and Prog [|program ^pid(^(idl:List Id)); ^d^sds^cc. |] is =
          let (env1,w1) = D d initenv (putinput initstate is) in
          let env2 = revitlist SD sds env1 in
             getoutput (CC cc env2 w1)
```

The result of applying the function `Prog` to the element in section 4 and the LML list `[10;9;8;7;6;5;4;3;2;1;0]` is `[0;1;2;3;4;5;6;7;8;9;10]`.

Part IV

Precedences in Specifications and Implementations of Programming Languages

Precedences in Specifications and Implementations of Programming Languages[1]

Annika Aasa

1 Introduction

Precedences are used in many language descriptions to resolve ambiguities. The reason for resolving ambiguities with precedences, instead of using an unambiguous grammar, is that the language description often becomes shorter and more readable. An unambiguous grammar which reflects different precedences of operators usually contains a lot of nonterminals and single productions. Consider for example an ambiguous grammar for simple arithmetic expressions and the unambiguous alternative.

```
E ::= E + E
    | E - E
    | E * E
    | E / E
    | int
    | ( E )
```

```
E ::= E + T
    | E - T
    | T

T ::= T * F
    | T / F
    | F

F ::= int
    | ( E )
```

If the language contains also prefix and postfix operators, then the unambiguous grammar will be surprisingly large.

If a language has user defined operators, as for example ML [Mil84] and PROLOG [SS86] it is also convenient to use precedences. When a new operator is introduced, the grammar is augmented with a new production, and it is hard to imagine how a user would be able to indicate where to place this production in an unambiguous grammar with different nonterminals.

When dealing with precedences, at least two questions arise. First, although precedences are used in many situations, there is no adequate definition of what it means for a production in a grammar to have higher precedence than another production. Precedences are only used to guide which steps a parser would take when there is an ambiguity in the grammar [AJU75, Ear75, Sha88, Wha76]. It is not always easy, given an ambiguous grammar and a set of disambiguating precedence rules, to decide if a parse tree belongs

[1] **This paper is an extension of a paper published in Proceedings of the Third international Symposium on Programming Language Implementation and Logic Programming, Lecture Notes in Computer Science, Volume 528, 1991**

to the language. The second question is if it is possible to transform a grammar with precedence rules to an ordinary context-free grammar. This is surprisingly complicated for grammars containing prefix and postfix operators of different precedences.

For a subclass of context-free grammars, we will give a parser independent definition of precedences and an algorithm which transforms a grammar with precedences to an unambiguous context-free grammar.

2 Distfix Grammars and Precedence

Let us first define what kind of grammars we will consider. In the definition `op` stands for an arbitrary operator word, in analogy with `int` and `id`.

Definition 1 *A **distfix grammar** is a grammar of the form*

E	::=	E `op` E	$\cdots$	E `op` $E \cdots$ `op` E	*infix distfix operators*
	$\mid$	`op` E	$\cdots$	`op` $E \cdots$ `op` E	*prefix distfix operators*
	$\mid$	E `op`	$\cdots$	E `op` $\cdots E$ `op`	*postfix distfix operators*
	$\mid$	`op`	$\cdots$	`op` E `op` $\cdots E$ `op`	*closed distfix operators*
	$\mid$	*int*			
	$\mid$	*id*			

where an initial operator word does not work also as a subsequent operator word and no whole sequence of operator words are an initial sequence of operator words of another operator.

We can divide distfix operators into five kinds: left associative infix distfix, right associative infix distfix, prefix distfix, postfix distfix and closed distfix. We will sometimes use AE as a shorthand for all atomic expressions such as integers and identifiers.

An example of a prefix distfix operator is `if` $-$ `then` $-$ `else`. As examples of what the extra requirements imply we consider which productions are allowed if the following production is already in the grammar:

E ::= `if` E `then` E `else` E

The following productions are then illegal:

E ::= `if` E `then` E
E ::= `olle` E `if` E `erik`

We will here concentrate on the special case with infix, prefix and postfix operators but the ideas can easily be extended to include distfix operators, and we will indicate how that can be done. The requirements on the operators when we only consider infix, prefix and postfix operators mean that all operators must be distinct.[2]

The requirement that distfix grammars only have one nonterminal is not as hard as it seems. In many language descriptions, precedences are used to resolve ambiguity in just one part of the language and that part can be described by a grammar with only one nonterminal. The same ideas of defining precedences can also be extended to more general grammars as shown in part V of this thesis.

[2]To allow both unary and binary minus in a language we may assume that the lexical analyzer translates them to different operators.

Definition 2 *A **precedence grammar** is a distfix grammar together with precedence rules.*

With precedence rules we mean both precedence and associativity rules. We will denote precedence grammars as follows.

E	::=	`$` E	3	
	|	E `+` E	2	left associative
	|	`#` E	1	
	|	`int`		

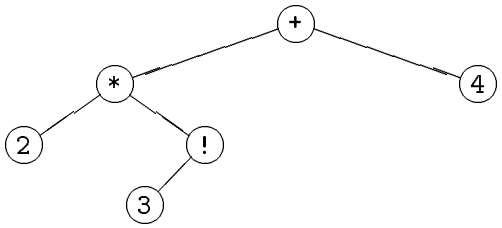
The precedences are given as numbers together with the productions. For these simple grammars we can just as well say that it is the operators that have precedence. The precedence of an operator *op* will be denoted $\mathsf{P}(op)$. Operators of different kinds are not allowed to have the same precedence. We do not for example allow a prefix operator to have the same precedence than a postfix operator. We let the variable H range over all precedence grammars which satisfy the requirements above.

We use the convention that a production with higher precedence has less binding power than one with lower precedence. Thus, for the usual arithmetic operators the addition operator `+` has higher precedence than the multiplication operator `*`. This convention is used for example in PROLOG [SS86] and OBJ [FGJM85]. This convention is unusual, most other languages use the opposite convention, but we have chosen it to make the algorithm in section 4.1 and the proof of it more clearer.

Since precedences have to do with structure we have to consider parse trees or syntax trees instead of strings when we talk about which language a precedence grammar defines. We will use syntax trees and we will for example picture the derivation

$$E \to E\texttt{+}E \to E\texttt{*}E\texttt{+}E \to E\texttt{*}E\texttt{!+}E \to^* \texttt{2*3!+4}$$

as



Note that the sentence can easily be obtained by flatting the syntax tree. A *syntax tree for an operator* is a syntax tree with that operator as root.

3 Definition of Precedence and Associativity

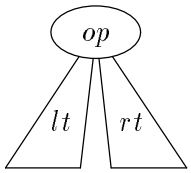
One obvious question to ask is which language we define with a precedence grammar. The language is of course a subset of the language generated by the ambiguous grammar without precedence rules. The precedence rules throw away some parse trees. We will call the parse trees we keep *precedence correct*.

It is unsatisfactory to define the precedence correct trees in terms of a specific parsing method. A specification of a language should not involve a method to recognize it, because

if the language is defined by one parsing method it could be hard to see if a parser which uses another method is correct.

We will define a predicate Pc_H which given a precedence grammar H defines the precedence correct trees. So, $\mathsf{Pc}_H(t)$ holds if and only if the syntax tree t is correct according to the disambiguating rules in the grammar H. The predicate is defined in such a way that syntax trees built by an operator precedence parser [ASU86, Flo63] are precedence correct. This and the converse, i.e. that every precedence correct tree can be recognized by an operator precedence parser is proved in a later section.

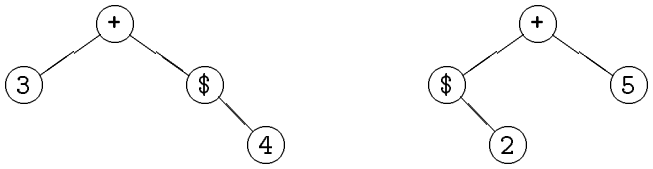
Let us first make some reflections. A syntax tree with an infix operator as root has the following form:



If it would be precedence correct, both the subtrees lt and rt must of course be precedence correct. Furthermore, there must be some requirements involving the precedence of the root operator. For languages with only infix operators it is enough to look at the precedences of the roots of the subtrees. They must be less than the precedence of the root. This is however not enough if the language contains also prefix and postfix operators. Consider the precedence grammar:

E	::=	`$` E	3	
	|	E `+` E	2	left associative
	|	`#` E	1	
	|	`int`		

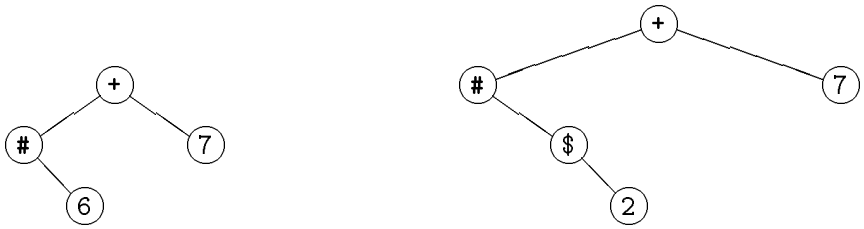
Are the following syntax trees precedence correct?



3+$4 $2+5

We want to consider the left syntax tree as precedence correct but not the syntax tree to the right. This illustrates that prefix operators with higher precedence than an infix operator must be allowed to occur in the right subtree.

Furthermore, consider the two syntax trees below, generated from the same grammar:

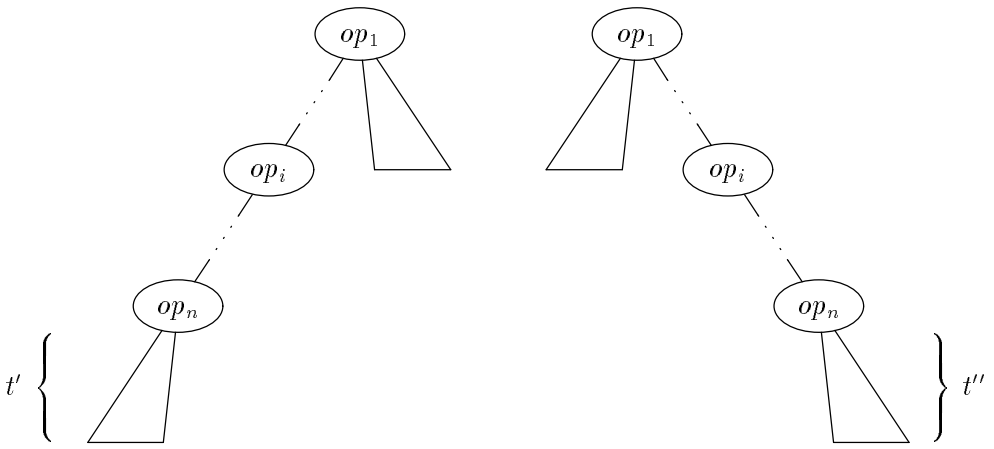


We want to consider the left syntax tree as precedence correct but not the syntax tree to the right. This illustrates that even if the precedence of the root operator of a subtree is less than the precedence of the whole tree, the syntax tree need not be precedence correct. To solve this problem we introduce two different kinds of precedence weights of a syntax tree, the left weight, Lw, and the right weight, Rw. Prefix operators have precedence only to the right, postfix operators only to the left and infix operators in both directions. The weights depend both on the root operator and the weights of the subtrees, and we define them as follows.

Definition 3

$Lw(AE) = 0$	$Rw(AE) = 0$
$Lw(t\ op) = max(P(op), Lw(t))$	$Rw(t\ op) = 0$
$Lw(op\ t) = 0$	$Rw(op\ t) = max(P(op), Rw(t))$
$Lw(lt\ op\ rt) = max(P(op), Lw(lt))$	$Rw(lt\ op\ rt) = max(P(op), Rw(rt))$

It is easy to realize that the right weight of a syntax tree is the maximal precedence of the infix and prefix operators in the chain to the right, and the left weight of a syntax tree is the maximal precedence of the infix and postfix operators in the chain to the left as pictured below. The tree t' is either atomic or a tree for a prefix operator, and the tree t'' is either atomic or a tree for a postfix operator:



We can now give the definition of the predicate Pc_H that defines the precedence correct syntax trees.

Definition 4 *Given a precedence grammar H, the following rules define the predicate Pc_H, where Left, Right, Pre and Post respectively, denote the set of left associative infix operators, right associative infix operators, prefix operators and postfix operators.*

atomic expressions:

$$\mathsf{Pc}_H(AE)$$

left associative infix operators:

$$\frac{op \in Left \quad \mathsf{Pc}_H(lt) \quad \mathsf{Pc}_H(rt) \quad \mathsf{Rw}(lt) \leq \mathsf{P}(op) \quad \mathsf{Lw}(rt) < \mathsf{P}(op)}{\mathsf{Pc}_H(lt\ op\ rt)}$$

right associative infix operators:

$$\frac{op \in Right \quad \mathsf{Pc}_H(lt) \quad \mathsf{Pc}_H(rt) \quad \mathsf{Rw}(lt) < \mathsf{P}(op) \quad \mathsf{Lw}(rt) \leq \mathsf{P}(op)}{\mathsf{Pc}_H(lt\ op\ rt)}$$

prefix operators:

$$\frac{op \in Pre \quad \mathsf{Pc}_H(t) \quad \mathsf{Lw}(t) < \mathsf{P}(op)}{\mathsf{Pc}_H(op\ t)}$$

postfix operators:

$$\frac{op \in Post \quad \mathsf{Pc}_H(t) \quad \mathsf{Rw}(t) < \mathsf{P}(op)}{\mathsf{Pc}_H(t\ op)}$$

In the rest of this paper, a *precedence correct* tree is assumed to be precedence correct according to this definition. The definition can easily be extended to distfix operators. We just notice that the subtrees between operator words of the same operator are allowed to have arbitrary precedence weights as long as they are precedence correct. The precedence weights of the subtrees outside the leftmost and rightmost operator word must satisfy the same conditions as infix, prefix and postfix operators. If we let op denote a complete distfix operator while $op_1, \cdots, op_n$ denote the operator words in op, then for example the rule for infix distfix can be written as follows:

left associative infix distfix operators:

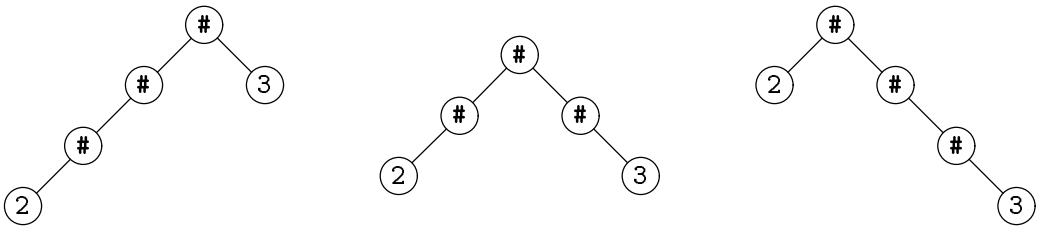
$$\frac{op \in Left \quad \mathsf{Pc}_H(t_0) \cdots \mathsf{Pc}_H(t_n) \quad \mathsf{Rw}(t_0) \leq \mathsf{P}(op) \quad \mathsf{Lw}(t_n) < \mathsf{P}(op)}{\mathsf{Pc}_H(t_0\ op_1\ t_1\ \cdots\ t_{n-1}\ op_n\ t_n)}$$

The only requirement for a closed distfix operator is that the subtrees are precedence correct:

closed distfix operators:

$$\frac{op \in Closed \quad \mathsf{Pc}_H(t_1) \cdots \mathsf{Pc}_H(t_{n-1})}{\mathsf{Pc}_H(op_1\ t_1\ \cdots\ t_{n-1}\ op_n)}$$

The requirement that the operators are distinct is important. Assume that we have an operator `#` that is both a prefix, postfix and infix operator. Consider the sentence `2###3` and the three possible trees:

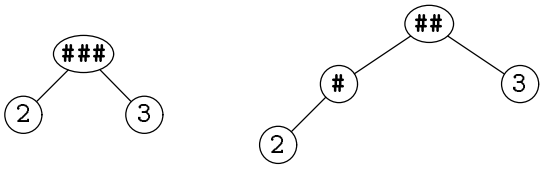


All of them are precedence correct regardless of which precedences we give to the productions. This arise from the fact that we could not know for each occurrence of the operator `#` if it is a prefix, postfix or infix operator. If we annotate each occurrence of the operator with which kind it is, then there is only one syntax tree for the sentence.

Another ambiguity problem can arise if we have operators with different length of the same character. Consider for example the following grammar.

$$\begin{array}{lcl} E & ::= & E\ \texttt{\#\#\#}\ E \\ & | & E\ \texttt{\#\#}\ E \\ & | & E\ \texttt{\#} \\ & | & \texttt{int} \end{array}$$

The sentence `2###3` has two different syntax trees:



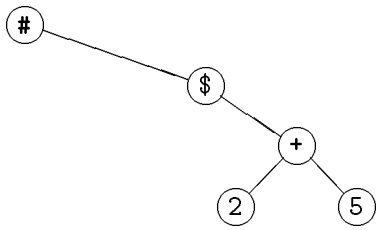
Both are precedence correct regardless of which precedences we give to the productions. We think that this restriction should be taken care of in the lexical analyzer. A lexical analyzer usually finds the longest possible token.

An alternative way to define the precedence correct trees is to define which operators are allowed to occur in each subtree. To say that, we need a new definition which we will also use later.

Definition 5 *An occurrence of an operator in a syntax tree t is **covered** if it occurs in a subtree of an operator with higher precedence than itself. An occurrence of an operator is **uncovered** if it is not covered.*

It is possible for an operator with higher precedence to occur in a subtree of an operator with lower precedence if it is covered. An example of this is the precedence correct syntax tree below generated from the same precedence grammar as on page 70:

E	::=	`$` E	3	
	|	E `+` E	2	left associative
	|	`#` E	1	
	|	`int`		



`#$2+3`

The prefix operator `$` *covers* the infix operator `+`. Postfix operators can be in the left subtree of an infix operator node independently of their precedence but not in the right subtree. Analogously, prefix operators can be in the right subtree of an infix operator node independently of their precedence but not in the left subtree. The conclusion of this is that if a syntax tree $lt\ op\ rt$ (where op is left associative) must be precedence correct both lt and rt must be precedence correct, all infix and prefix operators in lt with higher precedence than op must be covered and all infix and postfix operators in rt with higher or equal precedence than op must be covered.

3.1 "Correctness" of the definition

That the definition is sensible is motivated by three theorems. The first one states that there is exactly one precedence correct tree for each sentence generated by a distfix grammar. This is desirable since we want to use precedences to throw away some syntax trees but not all. Note that this imply that a precedence grammar is unambiguous. The other two theorems motivate that it is the "correct" syntax tree that is precedence correct. They state that an operator precedence parser [ASU86, Flo63] gives as result exactly the precedence correct trees.

3.1.1 Uniqueness of precedence correct trees

We will prove that there is exactly one precedence correct tree for each sentence generated by a distfix grammar. This is theorem 12. To prove the theorem we need some definitions. The first two can be compared to the definition of covering.

Definition 6 *An operator, op, is* ***postfix captured*** *in a sentence if there is a postfix operator to the right of op with higher precedence.*

$$\cdots op \cdots postop \cdots \qquad P(op) < P(postop)$$

Definition 7 *An operator, op, is* ***prefix captured*** *in a sentence if there is a prefix operator to the left of op with higher precedence.*

$$\cdots preop \cdots op \cdots \qquad P(op) < P(preop)$$

The next definition characterizes the operator in a sentence which would be the root in a precedence correct syntax tree.

Definition 8 *A* ***top operator*** *in a sentence generated by a distfix grammar is either*

1. *A postfix operator, postop, such that*
   (a) *there are not any operators to the right of postop, and*
   (b) *all infix and prefix operators in the subsentence w' to postop with higher precedence than postop are postfix captured in w'.*

   or

2. *A prefix operator, preop, such that*
   (a) *there are not any operators to the left of preop, and*
   (b) *all infix and postfix operators in the subsentence w' to preop with higher precedence than preop are prefix captured in w'.*

   or

3. *A left associative infix operator, inl, such that*
   (a) *all infix and prefix operators in the left subsentence w' to inl with higher precedence than inl are postfix captured in w'.*
   (b) *all infix and postfix operators in the right subsentence w'' to inl with higher or equal precedence than inl are prefix captured in w''.*

   or

4. *A right associative infix operator, inr, such that*
   (a) *all infix and prefix operators in the left subsentence w' to inr with higher or equal precedence than inr are postfix captured in w'.*
   (b) *all infix and postfix operators in the right subsentence w'' to inr with higher precedence than inr are prefix captured in w''.*

The definition can be used also for distfix operators if we regard all operator words of an operator and the enclosed expressions as a whole. In for example the expression `if` E `then` E `else` E, we regard `if` E `then` E `else` as a whole and thus the top operator is either the `if-then-else` operator or can be found in the E outside the operator. In the proof of lemma 9 there is an algorithm that finds the top operator in a sentence.

Using the definition of top operator and the three lemmas 9, 10 and 11 below we can prove theorem 12. The proofs of the lemmas are given in appendix A.

Lemma 9 *Every sentence generated by a distfix grammar with at least one operator has one and only one top operator.*

Lemma 10 *A syntax tree is precedence correct if it*

- *is without operators.*
- *has the top operator as root and precedence correct subtrees.*

Lemma 11 *A syntax tree in which the root operator is not the top operator, is not precedence correct.*

Theorem 12 *There is exactly one precedence correct tree for each sentence generated by a distfix grammar.*

Proof: The proof is by induction on the structure of a sentence w.

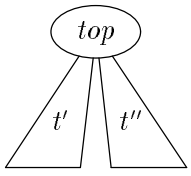
Base:
There are not any operators in w. Clearly there is exactly one precedence correct syntax tree for w.

Induction step:
We assume that all subsentences of w, belonging to the same language as w, have exactly one precedence correct syntax tree and show that w has exactly one precedence correct syntax tree.

Lemma 9 gives that w contains one and only one top operator, *top*. Let us first assume that *top* is an infix operator and therefore appear somewhere in the middle of w.

$$\underbrace{w' \quad top \quad w''}_{w} \tag{1}$$

The two subsentences w' and w'' belong to the same language as w, and thus the induction assumption gives that there is exactly one precedence correct syntax tree for w' and w'' respectively. We call these syntax trees t' and t''. Lemma 10 gives, since *top* is a top operator in w and both t' and t'' are precedence correct, that the syntax tree below is precedence correct:



There cannot be any other precedence correct tree as follows from lemma 11 and thus there is exactly one precedence correct tree for w. If we instead assume that *top* is a prefix or postfix operator, we only get one subsentence. Otherwise, the reasoning is the same. □

Note that if we extend precedence grammars to include also nonassociative infix operators then theorem 12 no longer holds, since there can be sentences which do not have a precedence correct tree. Take for example the usually nonassociative operator `=`. There is no precedence correct tree for the sentence `1=2=3`. A weaker formulation of theorem 12, that each sentence has at most one precedence correct tree can be shown for precedence grammars including nonassociative infix operators.

3.1.2 Comparison with operator precedence parsing

It is easy to translate a precedence grammar to an operator precedence table used in operator precedence parsing. An algorithm is given in the "dragon" book by Aho, Sethi and Ullman [ASU86, chapter 4.6].

Theorem 13 *Parsing a sentence generated from a precedence grammar with an operator precedence parser gives a precedence correct tree as result.*

Proof: The proof is by induction on the number of reductions in the parsing process.

Base:
No reductions are used. Trivial.

Induction step:
We show that if we have done n reductions and the trees built from these reductions are precedence correct then the resulting tree after one more reduction is precedence correct.

Case analysis: on the possible handles in an operator precedence parser.

$\lessdot E\ post \gtrdot$	To show that the tree after the reduction is precedence correct we must, according to definition 4, show that $\mathsf{Rw}(E) < \mathsf{P}(post)$. We know that $\mathsf{Rw}(E)$ is the maximal precedence of the chain of prefix and infix operators to the right. Assume that *op* is the one with highest precedence. It is not possible that *op* has higher precedence than *post* because then we would have reached the configuration $\cdots\ op \lessdot E'\ post \gtrdot \cdots$ some time earlier in the parsing process. In such a configuration, $E'\ post$ would have been chosen for reduction and we would never reach the configuration $\cdots \lessdot E\ post \gtrdot \cdots$.
$\lessdot pre\ E \gtrdot$	Analogous to the postfix case.
$\lessdot E_1\ inl\ E_2 \gtrdot$	The proof of $\mathsf{Rw}(E_1) \leq \mathsf{P}(inl)$ is analogous with the postfix case. The proof of $\mathsf{Lw}(E_2) < \mathsf{P}(inl)$ is analogous with the prefix case.

□

Theorem 14 *An operator precedence parser can give all precedence correct trees as result.*

Proof: Take an arbitrary precedence grammar H and an arbitrary precedence correct syntax tree t generated by H. We will prove that t can be a result from an operator precedence parse. Call the sentence of t for w. Parsing w by an operator precedence parser does not give raise to a syntax error since w is a correct sentence so the result of the parsing is a syntax tree t'. Since we have shown in theorem 13 that operator precedence parsers only gives precedence correct trees as result then t' must be precedence correct. Theorem 12 says that two different syntax tres for the same sentence not both can be precedence correct. This means that t' must be equal to t and thus, since t was arbitrary, an operator precedence parser can generate all precedence correct trees. □

4 Transformation to an Unambiguous Grammar

Besides the theoretical interest of knowing whether a precedence grammar can be transformed to an unambiguous context-free grammar, such an algorithm is sometimes needed in practice. For example, if we want to describe a language with a precedence grammar but parse the language with a method that cannot handle precedence rules, then the algorithm is definitely needed. One such commonly used parsing method is recursive descent [DM81], and another is DCG [PW83]. It is neither obvious how to use precedence rules in Earley's algorithm [Ear70] even if it is possible as shown in part V of this thesis.

For grammars with only infix operators, there is a well-known algorithm [ASU86, chapter 2.2] that transforms them to ordinary unambiguous context-free grammars by introducing one nonterminal for each precedence level. But if the language contains also prefix and postfix operators, this method does not work. Consider the precedence grammar:

E	::=	E `?`	4	
	|	E `+` E	3	left associative
	|	E `!`	2	
	|	E `*` E	1	left associative
	|	`int`		

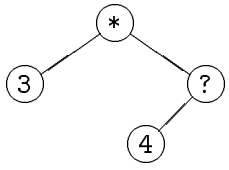
For this grammar the method of introducing one nonterminal for each precedence level does not work. Using the method naïvely would give the grammar:

$E(4)$	::=	$E(4)$ `?`	|	$E(3)$
$E(3)$	::=	$E(3)$ `+` $E(2)$	|	$E(2)$
$E(2)$	::=	$E(2)$ `!`	|	$E(1)$
$E(1)$	::=	$E(1)$ `*` $E(0)$	|	$E(0)$
$E(0)$	::=	`int`		

But this grammar is incorrect since it does not generate *all* precedence correct syntax trees. It does not generate all sentences as the original grammar, for example are not `7?+8`, `3?!` and `9+6?*8` derivable. There exists an unambiguous grammar which generates the same sentences as the precedence grammar above:

$E(3)$	::=	$E(3)$ `+` $E(1)$	|	$E(1)$		
$E(1)$	::=	$E(1)$ `*` $E(0)$	|	$E(0)$		
$E(0)$	::=	`int`	|	$E(0)$ `!`	|	$E(0)$ `?`

But this grammar is not correct since it generates syntax trees which are *not* precedence correct, for example:



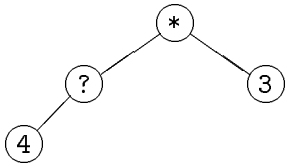
`3*4?`

Another attempt to construct a grammar from which precisely the precedence correct syntax trees are derivable is:

$$
\begin{array}{lcl}
E(4) & ::= & E(3) \\
E(3) & ::= & E(3)\ \texttt{+}\ E(2) \ \mid\ E(2) \\
E(2) & ::= & E(1) \\
E(1) & ::= & E(1)\ \texttt{*}\ E(0) \ \mid\ E(0) \\
E(0) & ::= & \texttt{int} \ \mid\ E(2)\ \texttt{!} \ \mid\ E(4)\ \texttt{?}
\end{array}
$$

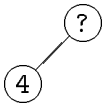
Here we have tried to incorporate the idea that a postfix operator forms a closed expression. This grammar is also incorrect since it is ambiguous and derives both precedence correct syntax trees and incorrect ones. This illustrates that we must construct the grammar in such a way that, for every production $E ::= E_l * E_r$, it is not possible to derive a syntax tree for a postfix operator with higher precedence than * from E_r. Syntax trees for postfix operators with lower precedence than * must of course be derivable from E_r.

Let us now make some reflections about the syntax trees which must be derivable from the nonterminal E_l in the production $E ::= E_l * E_r$. This is harder because we sometimes want syntax trees for postfix operators with higher precedence than * to be derivable from E_l and sometimes not. Consider the syntax tree:

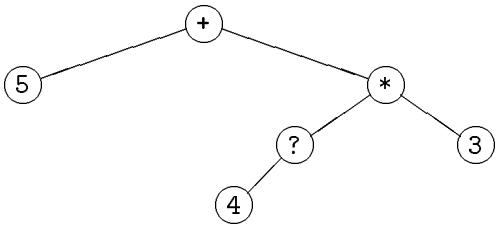


4?*3

It is precedence correct and has as left subtree:

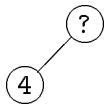


Therefore, we must ensure that the subtree is derivable from E_l in a production $E ::= E_l * E_r$. We have shown that there must be at least one production $E ::= E_l * E_r$ such that we can derive syntax trees for postfix operators with higher precedence than * from E_l. We will now show that we must also have productions $E ::= E_l * E_r$ such that we cannot derive syntax trees for postfix operators with higher precedence than * from E_l. Consider the following syntax tree in which the syntax tree above is a subtree:

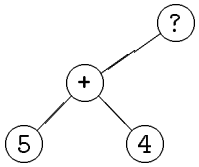


5+4?*3

This syntax tree is not precedence correct since `?` has higher precedence than `+`. So in this case we must ensure that the syntax tree below is *not* derivable from E_l:

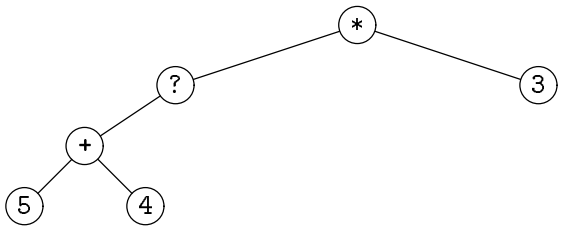


Note that the occurrence of `+` is covered in the syntax tree:



`5+4?`

This syntax tree must be derivable from E_l, since the precedence correct syntax tree for the sentence `5+4?*3` is:



That is, if a postfix operator is allowed to occur in a syntax tree derived from a nonterminal, it can cover also other operators. The reasoning for the case where prefix operators are allowed to occur is analogous.

We have indicated that we must have more than one nonterminal from which it is possible to derive a syntax tree with a specific infix operator as root. In the nonterminals there must be information about which postfix and prefix operators are allowed to occur in the left (right) subtree. If a postfix (prefix) operator is allowed then it can cover also other operators which otherwise are not allowed to occur in the left (right) subtree. So, the number of different nonterminals we need for each infix operator depends on how many postfix and prefix operators with higher precedence there are in the precedence grammar.

4.1 The algorithm $\mathcal{M}$

In this section we transform our observations to an algorithm that generates a context-free grammar where the precedence rules in a precedence grammar are incorporated. For simplicity the algorithm handles exactly one operator on each precedence level. This is not a severe restriction. We could easily extend the algorithm to allow several operators on each level or extend the resulting grammar with more operators. Another restriction is that we do not handle distfix operators but only infix, prefix and postfix operators. Nor is this a severe restriction. We could easily extend the algorithm to allow distfix operators.

Our algorithm generates a grammar with nonterminals of the form $E(n,p,q)$ where the indices are natural numbers and show which operators are allowed to occur in the syntax trees derived from the nonterminal. Before giving the algorithm we introduce some notation.

inl_i	the i:th left associative infix operator
inr_i	the i:th right associative infix operator
pre_i	the i:th prefix operator
$post_i$	the i:th postfix operator
$\mathsf{P}_{\mathsf{pre}}(n,p)$	the precedence of the p:th prefix operator with higher precedence than n
$\mathsf{P}_{\mathsf{post}}(n,q)$	the precedence of the q:th postfix operator with higher precedence than n.

Given a grammar, left associative infix operators, right associative infix operators, prefix operators and postfix operators are numbered separately in increasing precedence order. We define $\mathsf{P}_{\mathsf{post}}(x,0) = x$ and $\mathsf{P}_{\mathsf{pre}}(x,0) = x$. Examples are given in section 4.2. Now we can define which operators must not occur in a syntax tree derived from $E(n,p,q)$.

1. An operator *op* for which $\mathsf{P}(op) > max(\mathsf{P}_{\mathsf{pre}}(n,p), \mathsf{P}_{\mathsf{post}}(n,q))$.
2. An uncovered prefix operator *op* for which $\mathsf{P}(op) > \mathsf{P}_{\mathsf{pre}}(n,p)$.
3. An uncovered postfix operator *op* for which $\mathsf{P}(op) > \mathsf{P}_{\mathsf{post}}(n,q)$.
4. An uncovered infix operator *op* for which $\mathsf{P}(op) > n$.

As mentioned earlier there must be more than one production for each operator, the number depends on the number of prefix and postfix operators with higher precedence. The algorithm generates the grammar by five rules which introduce the nonterminals $E(n,p,q)$ and the productions for them.

1. The rule for left associative infix operators.

$$\begin{array}{lcl} E(\mathsf{P}(inl_i),p,q) & ::= & E(\mathsf{P}(inl_i),0,q)\ \ inl_i\ \ E(\mathsf{P}(inl_i)-1,p,0) \\ & | & E(\mathsf{P}(inl_i)-1,p,q) \end{array}$$

where $1 \le i \le$ number of left associative infix operators
$0 \le p \le$ number of prefix operators with higher precedence than inl_i
$0 \le q \le$ number of postfix operators with higher precedence than inl_i

2. The rule for right associative infix operators.

$$\begin{array}{lcl} E(\mathsf{P}(inr_i),p,q) & ::= & E(\mathsf{P}(inr_i)-1,0,q)\ \ inr_i\ \ E(\mathsf{P}(inr_i),p,0) \\ & | & E(\mathsf{P}(inr_i)-1,p,q) \end{array}$$

where $1 \le i \le$ number of right associative infix operators
$0 \le p \le$ number of prefix operators with higher precedence than inr_i
$0 \le q \le$ number of postfix operators with higher precedence than inr_i

3. The rule for prefix operators.

$$E(\mathsf{P}(pre_i), p, q) \quad ::= \quad E(\mathsf{P}(pre_i) - 1, p + 1, q)$$

where $1 \leq i \leq$ number of prefix operators
$0 \leq p \leq$ number of prefix operators with higher precedence than pre_i
$0 \leq q \leq$ number of postfix operators with higher precedence than pre_i

4. The rule for postfix operators.

$$E(\mathsf{P}(post_i), p, q) \quad ::= \quad E(\mathsf{P}(post_i) - 1, p, q + 1)$$

where $1 \leq i \leq$ number of postfix operators
$0 \leq p \leq$ number of prefix operators with higher precedence than $post_i$
$0 \leq q \leq$ number of postfix operators with higher precedence than $post_i$

5. The A-rule.

$$\begin{array}{lcll} E(0,p,q) & ::= & AE & \\ & | & pre_i \;\; E(\mathsf{P}(pre_i), p-i, 0) & \text{where } 1 \leq i \leq p \\ & | & E(\mathsf{P}(post_j), 0, q-j) \;\; post_j & \text{where } 1 \leq j \leq q \end{array}$$

where $0 \leq p \leq$ number of prefix operators
$0 \leq q \leq$ number of postfix operators

The start symbol in the resulting grammar is the nonterminal $E(m, 0, 0)$ where m is the highest precedence.

4.2 Example

Let us use the method to construct an unambiguous grammar for the language generated by the precedence grammar:

E	::=	E `?`	4	
	|	E `+` E	3	left associative
	|	E `!`	2	
	|	E `*` E	1	left associative
	|	`int`		

For this grammar we have:

$inl_1 =$ `*`	$post_1 =$ `!`	$\mathsf{P}_{\mathsf{post}}(1,2) =$ `?`
$inl_2 =$ `+`	$post_2 =$ `?`	$\mathsf{P}_{\mathsf{post}}(2,1) =$ `?`

The rule for left associative infix operators yields the following productions since $\mathsf{P}(\texttt{+}) = 3$ and there is one postfix operator with higher precedence than `+` but no prefix operators:

$$E(3,0,0) \quad ::= \quad E(3,0,0) \;\texttt{+}\; E(2,0,0) \quad | \quad E(2,0,0)$$

$$E(3,0,1) \quad ::= \quad E(3,0,1) \;\texttt{+}\; E(2,0,0) \quad | \quad E(2,0,1)$$

The rule for left associative infix operators yields also the following productions since $\mathsf{P}(*) = 1$ and there are two postfix operators with higher precedence than * but no prefix operators:

$$E(1,0,0) \ ::= \ E(1,0,0) \ * \ E(0,0,0) \ \mid \ E(0,0,0)$$

$$E(1,0,1) \ ::= \ E(1,0,1) \ * \ E(0,0,0) \ \mid \ E(0,0,1)$$

$$E(1,0,2) \ ::= \ E(1,0,2) \ * \ E(0,0,0) \ \mid \ E(0,0,2)$$

The rule for postfix operators yields the following productions:

$$E(4,0,0) \ ::= \ E(3,0,1)$$

$$E(2,0,0) \ ::= \ E(1,0,1)$$

$$E(2,0,1) \ ::= \ E(1,0,2)$$

The first production arise since $\mathsf{P}(?) = 4$ and there are neither prefix operators nor postfix operators with higher precedence than `?`. The last two productions arise since $\mathsf{P}(!) = 2$ and there is one postfix operator with higher precedence than `!` but no prefix operators.

Finally the A-rule yields the following productions since we have two postfix operators, `!` and `?`:

$$E(0,0,0) \ ::= \ \texttt{int}$$

$$E(0,0,1) \ ::= \ \texttt{int} \ \mid \ E(2,0,0) \ \texttt{!}$$

$$E(0,0,2) \ ::= \ \texttt{int} \ \mid \ E(2,0,1) \ \texttt{!} \ \mid \ E(4,0,0) \ \texttt{?}$$

The resulting grammar contains some useless[3] nonterminals and a lot of single productions. These could easily be eliminated and algorithms for that is for example given by Grune and Jacobs [GJ90]. We can eliminate 8 of the 19 productions from the grammar above.

If we augment the grammar with a prefix operator having greater precedence than all other operators then the unambiguous grammar will consists of 42 productions and even if we eliminate all useless and all single productions there is still 26 left.

4.3 Correctness of algorithm $\mathcal{M}$

The correctness of algorithm $\mathcal{M}$ is shown by proving that every precedence grammar, H, generates the same language (the same set of syntax trees) as the grammar we obtain by applying the algorithm to H. We cannot consider the set of strings the two grammars generate since we are interested in the structure of the expressions. Neither could we consider parse trees since the nonterminals in the parse trees have different names and there are chains of single productions in the parse trees for the grammars that the algorithm produces. The correctness is formulated by the following theorem.

Theorem 15 *If $\mathcal{T}(\mathcal{M}(H))$ is the generated language of syntax trees from the grammar $\mathcal{M}(H)$ and $\mathcal{T}(H) = \{t : \mathsf{Pc}_H(t)\}$ then, for every precedence grammar H, the language $\mathcal{T}(H)$ is equal to the language $\mathcal{T}(\mathcal{M}(H))$.*

[3] A nonterminal is useless if it does not appear in any derivation of any sentence.

Proof: In the proof we use induction on the precedence grammar. By 'induction on a precedence grammar' we mean that we show a statement for a grammar consisting of zero operators, and under the assumption that a statement holds for a grammar consisting of m operators, we show that it holds if we extend the grammar with one more operator. The operators are introduced in increasing precedence order. We use the notation H_m for a precedence grammar where the highest precedence of the operators is m. Let H_m be equal to H_{m-1} plus the production for a new operator with precedence m. Then we have two important properties:

1. $\mathcal{T}(H_{m-1}) \subseteq \mathcal{T}(H_m)$
2. $\mathcal{P}(\mathcal{M}(H_{m-1})) \subseteq \mathcal{P}(\mathcal{M}(H_m))$
   where $\mathcal{P}(G)$ denote the set of productions in the grammar G

The proof of (theorem 15)

$$\mathcal{T}(H) = \mathcal{T}(\mathcal{M}(H))$$

is divided into two parts:

1. $\mathcal{T}(H) \subseteq \mathcal{T}(\mathcal{M}(H))$
2. $\mathcal{T}(\mathcal{M}(H)) \subseteq \mathcal{T}(H)$

In the first part we show that if a syntax tree is precedence correct, that is, if it can be generated from a precedence grammar H, then it can be generated from the grammar we obtain by applying the precedence removing algorithm $\mathcal{M}$ on H. In the second part we show that if a syntax tree is generated from a grammar we have obtained by applying the algorithm on a precedence grammar, then the syntax tree is precedence correct.

In both parts we use a predicate $Q_H(t, n, p, q)$ which informally holds if the syntax tree t is precedence correct and some operators given by the natural numbers n, p and q do not occur in t. We will define Q precisely later. In each part we show one direction of

$$Q_H(t, n, p, q) \iff E(n, p, q) \rightarrow^* t \tag{2}$$

We define Q in such a way that:

$$t \in \mathcal{T}(H_m) \iff Q_{H_m}(t, m, 0, 0) \tag{3}$$

Since $E(m, 0, 0)$ is the start symbol in the grammar $\mathcal{M}(H_m)$ we have:

$$E(m, 0, 0) \rightarrow^* t \iff t \in \mathcal{T}(\mathcal{M}(H_m)) \tag{4}$$

So, from 3 and 4 it follows that if we prove 2 we have then shown:

$$t \in \mathcal{T}(H_m) \iff t \in \mathcal{T}(\mathcal{M}(H_m)) \tag{5}$$

From 5, the theorem follows immediately.

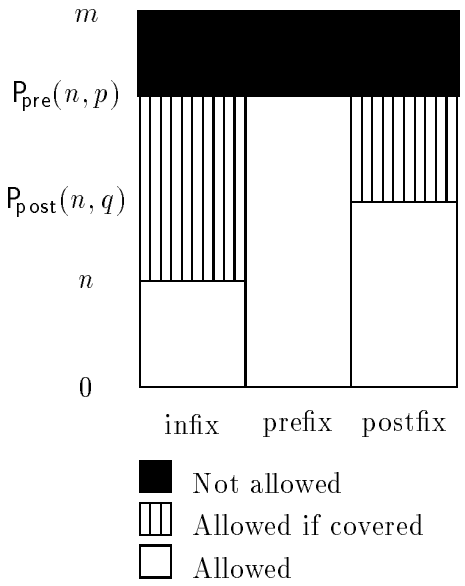
Let us turn to the definition of the predicate Q.

Definition 16 *$Q_H(t, n, p, q)$ holds if and only if*

1. *$t \in \mathcal{T}(H)$*
2. *The following operators do not occur in the syntax tree t.*
   (a) *An operator op for which $P(op) > max(P_{pre}(n, p), P_{post}(n, q))$.*
   (b) *An uncovered prefix operator op for which $P(op) > P_{pre}(n, p)$.*
   (c) *An uncovered postfix operator op for which $P(op) > P_{post}(n, q)$.*
   (d) *An uncovered infix operator op for which $P(op) > n$.*

Recall that an occurrence of an operator in a syntax tree t is uncovered if it does not occur in a subtree of an operator with higher precedence than itself.

Example 17: We illustrate which operators are allowed in a syntax tree t if $Q_{H_m}(t, n, p, q)$ would hold with the following picture:



□

Clearly $Q_{H_m}(t, m, 0, 0)$ holds if and only if $t \in \mathcal{T}(H_m)$.

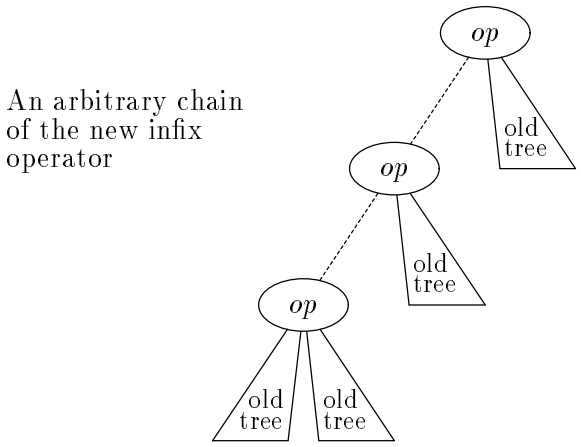
Proof of $\mathcal{T}(H) \subseteq \mathcal{T}(\mathcal{M}(H))$
Here, we only give an overview of the proof. The whole proof is found in appendix B.

First we use induction on the precedence grammar H.

Base:
There are no operators in H. Trivial.

Induction step:
Under the assumption that the transformation is correct for a precedence grammar with $m-1$ operators we show that it is correct if we extend the grammar with one new operator. We prove this by case analysis on the new operator.

1. **left associative infix operator**
   All trees have the following form:



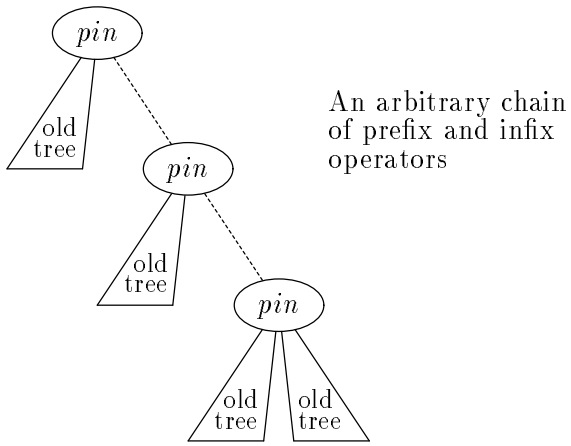
   We use induction on the length of the chain of new operator.

   Base:
   The length is zero, that is the tree is an old one.

   Induction step:
   Under the assumption that we can derive every tree with l occurrences of the new infix operator, we show that we can derive every tree with $l + 1$ occurrences of the new infix operator.

2. **right associative infix operator**
   Analogous with a left associative infix operator.

3. **prefix operator**
   All trees have the following form, where *pin* is either a prefix or infix operator:



We use induction on the length of the chain of infix and prefix operators.

Base:
The length is zero, that is the tree is an old one.

Induction step:
Under the assumption that we can derive every tree with a chain of l occurrences of infix and prefix operators, we show that we can derive every tree with a chain of $l+1$ occurrences of infix and prefix operators.

We show the two cases that the tree has an infix operator as root and a prefix operator as root.

4. **postfix operator**
   Analogous with a prefix operator.

□

Proof of $\mathcal{T}(\mathcal{M}(H)) \subseteq \mathcal{T}(H)$
We again only sketch the proof here. The whole proof is found in appendix B. To prove

$$\forall t.\forall(n,p,q).\ E(n,p,q) \rightarrow^* t \Longrightarrow Q_H(t,n,p,q) \tag{C6}$$

we use induction on the length of a derivation. Under the induction assumption

$$\forall t.\forall(n,p,q).\ E(n,p,q) \rightarrow^r t \Longrightarrow Q_H(t,n,p,q) \qquad \text{where } 1 \leq r < \gamma \tag{A7}$$

we show

$$\forall t.\forall(n,p,q).\ E(n,p,q) \rightarrow^\gamma t \Longrightarrow Q_H(t,n,p,q) \tag{C8}$$

To show

$$Q_H(t,n,p,q) \tag{C9}$$

for arbitrary t and (n,p,q) and given the assumption

$$E(n,p,q) \rightarrow^\gamma t \tag{A10}$$

we use case analysis on the first step in the derivation

1. $E(n,p,q) \rightarrow E(n-1,p+1,q)$

2. $E(n,p,q) \rightarrow E(n-1,p,q+1)$

3. $E(n,p,q) \rightarrow E(n,0,q)\ \ inop\ \ E(n-1,p,0)$

4. $E(n,p,q) \rightarrow preop_i\ \ E(\mathsf{P}(preop_i),p-i,0)$

5. $E(n,p,q) \rightarrow E(\mathsf{P}(postop_i),0,q-i)\ \ postop_i$

6. $E(n,p,q) \rightarrow E(n-1,p,q)$

□

5 Practical use of algorithm $\mathcal{M}$

We have used the algorithm to implement an experimental language with user defined distfix operators [Aas89], also described in part VI of this thesis. A distfix operator is specified by the operator words and optionally precedence and associativity. The parser is written in ML [MTH90] and uses parser constructors due to Burge [Bur75] and Fairbairn [Fai87] and Kent Petersson and Sören Holmström [Pet85]. Using these parser constructors it is easy to write a parser given a grammar, since there are constructors that recognize terminal symbols, sequences, and alternatives and other constructors that introduce actions during the parsing.

The parser constructors construct a recursive descent parser and therefore the grammar must not be left recursive and it must express precedences of the involved operators.

In the parser for user defined distfix operators we use the rules in the algorithm described above. We have to do some changes in order to remove left recursion, and we never generate the entire grammar with all different nonterminals. Instead we see the rules as production schemas in a way that is similar to the hyper rules in two-level grammars [CU77], and instantiate the rules during the parsing. Hanson [Han85] describes another technique for parsing expressions using recursive descent without introducing additional nonterminals, but this technique does not handle prefix and postfix operators of different precedence as our method.

Appendix A

Complete proofs of some lemmas

Complete proofs of the lemmas used in the proof of theorem 12 are given here. We use two lemmas in the proofs of lemma 10 and 11.

- If a prefix or infix operator, *pin*, appears in a syntax tree, t, and *pin* is not covered by a postfix operator then either
  - t is not precedence correct, or
  - $\mathsf{Rw}(t) \geq \mathsf{P}(pin)$

  (lemma 18)
- If a postfix or infix operator, *poin*, appears in a syntax tree, t, and *poin* is not covered by a prefix operator then either
  - t is not precedence correct, or
  - $\mathsf{Lw}(t) \geq \mathsf{P}(poin)$

  (lemma 19)

Lemma 9 *Every sentence generated by a distfix grammar with at least one operator has one and only one top operator.*

Proof: We only consider sentences with one or more operators and show that

there is a top operator in every sentence generated by a distfix grammar (C11)

and

there is at most one top operator in every sentence generated by a distfix grammar (C12)

We show statement C11 by induction on the number of operators in a sentence.

Base:
There is only one operator in the sentence. The operator is a top operator since there are no other operators that can violate the top operator conditions.

Induction step:
Assume that

> there is a top operator in every sentence with less than n operators but at least one operator (A13)

and show that

> there is a top operator in every sentence with n operators (C14)

To show C14 we take an arbitrary sentence w with n operators. In w we consider the operator with highest precedence. If there are several operators with equal (and highest) precedence we consider the leftmost one if the operator is a prefix or right associative infix operator and the rightmost one if the operator is a postfix or left associative infix operator. Clearly there must be one such operator in w.

Case analysis: on the kind of operator with highest precedence

1. infix operator, *in*
   Since *in* has highest precedence in w there cannot be any other operator that violates the conditions for *in* to be a top operator. Note that we have chosen the leftmost occurrence of *in* if *in* is a right associative infix operator and vice versa.

2. postfix operator, *postop*
   If *postop* is the rightmost operator then *postop* is a top operator. This holds since *postop* has the highest precedence in w and thus there cannot be any other operator that violates condition (b). If *postop* is not the rightmost one then we prove that there is a top operator in w as follows. Denote the part of w to the left and included *postop* for w', and the other part w''.

   $$\underbrace{\underbrace{\cdots\cdots postop}_{w'}\underbrace{\cdots\cdots}_{w''}}_{w} \tag{15}$$

   All operators in w' are postfixcaptured by *postop* since *postop* has the highest precedence of all operators in w. Thus, if the sentence, v, we obtain we replacing w' by an atomic expression has a top operator then also w has a top operator.

   $$\underbrace{AE\ w''}_{v} \tag{16}$$

   Clearly v has fewer operators than w and thus the induction assumption A13 gives that there is a top operator in v and thus also in w.

3. prefix operator
   Analogous to postfix operator case.

This concludes the induction and we have proved C11.

To prove statement C12 (there is at most one top operator in every sentence generated by a distfix grammar) we take an arbitrary sentence w and assume that there is a top operator, *top*, in w. We will prove that there cannot be another top operator in w.

Case analysis: on *top*

1. a postfix operator
   Clearly there cannot be any other postfix operator that also is a top operator since at most one can be the rightmost one. We prove by contradiction that there cannot be a prefix or infix operator that also is a top operator so we first assume that

   there is a prefix or infix operator *pin* that also is a top operator (A17)

   We now have two cases

   (a) $\mathsf{P}(pin) < \mathsf{P}(top)$
   In order for *pin* to be a top operator there must be a prefix operator *preop* such that $\mathsf{P}(preop) > \mathsf{P}(top)$ between *pin* and *top*. Let us chose the prefix operator between *pin* and *postop* with highest precedence.

   $$\cdots pin \cdots\cdots preop \cdots\cdots top \tag{18}$$

   But since *top* is a top operator there must be a postfix operator *postop* such that $\mathsf{P}(postop) > \mathsf{P}(preop)$ between *preop* and *top*.

   $$\cdots pin \cdots\cdots preop \cdots postop \cdots top \tag{19}$$

   But now *postop* violates the condition for *pin* to be a top operator. There cannot be a prefix operator with higher precedence than *postop* between *pin* and *postop* since *preop* has highest precedence of all prefix operators between *pin* and *top*. We have a contradiction and *pin* cannot be a top operator. Thus *top* is the only top operator in the sentence.

   (b) $\mathsf{P}(pin) > \mathsf{P}(top)$
   Almost analogous to (a).

2. a prefix operator
   Analogous to postfix operator case.

3. an infix operator
   The reasoning for the impossibility for another operator to be a top operator is either analogous with case 1 or case 2.

□

Lemma 10 *A syntax tree is precedence correct if it*

- *is without operators.*
- *has the top operator as root and precedence correct subtrees.*

Proof: A syntax tree without operators is clearly precedence correct so we assume that a syntax tree has the top operator as root and precedence correct subtrees and show that the syntax tree is precedence correct.

Case analysis: on the syntax tree

1. *t post*
   The rule for postfix operators

$$\frac{op \in Post \quad \mathsf{Pc}_H(t) \quad \mathsf{Rw}(t) < \mathsf{P}(op)}{\mathsf{Pc}_H(t\ op)}$$

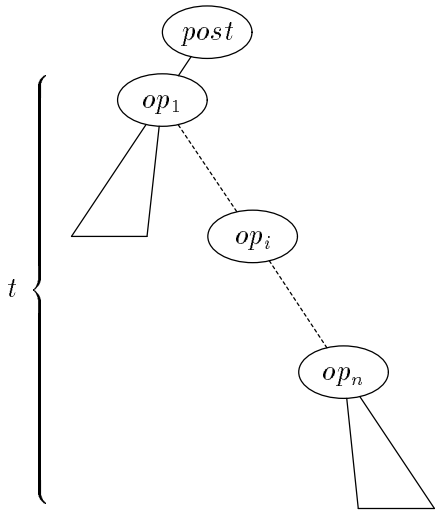
   gives that in order to show that *t post* is precedence correct given that *t* is precedence correct we must show that

$$\mathsf{Rw}(t) < \mathsf{P}(post) \tag{C20}$$

   The right weight of a syntax tree is defined as

$$\begin{aligned}
&\mathsf{Rw}(AE) = 0\\
&\mathsf{Rw}(t\ op) = 0\\
&\mathsf{Rw}(op\ t) = max(\mathsf{P}(op), \mathsf{Rw}(t))\\
&\mathsf{Rw}(lt\ op\ rt) = max(\mathsf{P}(op), \mathsf{Rw}(rt))
\end{aligned}$$

   From that it follows that if *t* is an atomic tree or a tree for a postfix operator then C20 holds. If *t* is a tree for an infix or prefix operator then $\mathsf{Rw}(t)$ is the maximum of the precedences for the infix and prefix operators $op_1, \ldots, op_n$.



   The subtree of op_n is either an atomic tree or a tree for a postfix operator. Let

$$op_i \text{ be the prefix/infix operator in the chain } op_1, \ldots, op_n \text{ with highest precedence.} \tag{A21}$$

   From A21 we have

$$\mathsf{Rw}(t) = \mathsf{P}(op_i) \tag{22}$$

We will show by contradiction that

$$\mathsf{P}(op_i) < \mathsf{P}(post) \tag{C23}$$

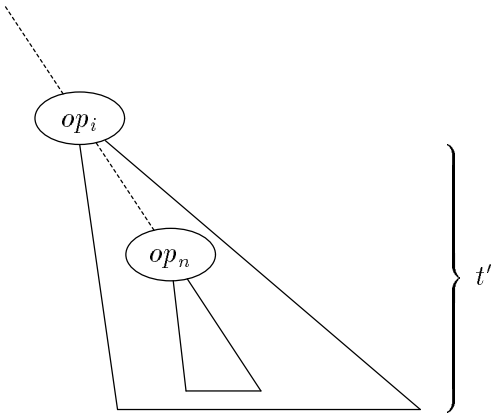
so let us start by assuming the opposite:

$$\mathsf{P}(op_i) > \mathsf{P}(post) \tag{A24}$$

Since *post* is a top operator there is another postfix operator *post*′ with higher precedence than op_i between op_i and *post* in the sentence. Let *post*′ be the postfix operator with highest precedence among those between op_i and *post*.

$$\cdots op_i \cdots post' \cdots post \qquad\qquad \mathsf{P}(post') > \mathsf{P}(op_i) \tag{25}$$

That is, *post*′ must appear in t'.



If *post*′ is not covered by a prefix operator in t' then the precondition for lemma 19 is fulfilled and we get that either

$$t' \text{ is not precedence correct} \tag{26}$$

or

$$\mathsf{Lw}(t') \geq \mathsf{P}(post') \tag{27}$$

Since t is precedence correct and thus all its subtrees, 26 immediately leads to a contradiction. Also 27 leads to a contradiction since

$$\underbrace{\underbrace{\mathsf{Lw}(t') \geq \mathsf{P}(post')}_{27} > \mathsf{P}(op_i)}_{25} \tag{28}$$

Thus, op_i t' cannot be precedence correct which again contradicts the fact all subtrees of t is precedence correct.

If $post'$ is covered by a prefix operator in t' then this contradicts the fact that *post* is a post operator. The contradiction follows from that a prefix operator, pre, which covers $post'$ in t has higher precedence than *post*, and

$$\underbrace{\underbrace{\mathsf{P}(pre) > \mathsf{P}(post')}_{pre \text{ covers } post'} > \underbrace{\mathsf{P}(op_i) > \mathsf{P}(post)}_{\text{A24}}}_{25} \tag{29}$$

Therefore, since *post* is a top operator there must be a postfix operator with higher precedence than *pre* between *pre* and *post* in the sentence. But this is impossible since $post'$ was chosen to have the highest precedence of all postfix operators in t'.

Thus we have proved that assumption A24 leads to a contradiction and thus C23 holds and then also C20, and we have proved the lemma for the postfix operator case.

2. $pre\ t$
   Analogous to $t\ post$.

3. $lt\ inl\ rt$ (left associative infix operator)
   The rule for left associative infix operators

$$\frac{op \in Left \quad \mathsf{Pc}_H(lt) \quad \mathsf{Pc}_H(rt) \quad \mathsf{Rw}(lt) \leq \mathsf{P}(op) \quad \mathsf{Lw}(rt) < \mathsf{P}(op)}{\mathsf{Pc}_H(lt\ op\ rt)}$$

   gives that in order to show that $lt\ inl\ rt$ is precedence correct given that both lt and rt is precedence correct we must show that

$$\mathsf{Rw}(lt) \leq \mathsf{P}(inl) \tag{C30}$$

   and

$$\mathsf{Lw}(rt) < \mathsf{P}(inl) \tag{C31}$$

   The proof of C30 is analogous to the postfix case and the proof of C31 is analogous to the prefix case.

4. $lt\ inr\ rt$ (right associative infix operator)
   Analogous to $lt\ inl\ rt$.

□

Lemma 11 *A syntax tree in which the root operator is not the top operator, is not precedence correct.*

Proof: Take a syntax tree where the top operator is not the root operator. Then there must be another operator that is root and the top operator must appear somewhere in a subtree.

Case analysis: on the syntax tree

1. *t post*
   Since *post* is not the top operator, there is a prefix or infix operator, *pin*, in the left subsentence w, $\mathsf{P}(pin) > \mathsf{P}(post)$ and *pin* is not postfixcaptured in w. That is,

   $$pin \text{ must appear in } t \text{ and is not covered by a postfix operator in } t. \tag{32}$$

   If we use lemma 18 together with 32 we get that either

   $$t \text{ is not precedence correct} \tag{33}$$

   or

   $$\mathsf{Rw}(t) \geq \mathsf{P}(pin) \tag{34}$$

   If 33 holds then of course *t post* is not precedence correct. If 34 holds then *t post* is not precedence correct since $\mathsf{P}(pin) > \mathsf{P}(post)$ and thus $\mathsf{Rw}(t) > \mathsf{P}(post)$ and then the condition in the rule for postfix operator is not fulfilled.

2. *pre t*
   Analogous to *t post*.

3. *lt inl rt*
   Since *inl* is not the top operator, we have two cases.

   (a) There is an infix or prefix operator, *pin*, in the left subsentence, w', to *inl*, $\mathsf{P}(pin > \mathsf{P}(inl)$ and *pin* is not postfixcaptured in w'. This case is analogous to case 1.

   (b) There is an infix or postfix operator, *poin*, in the right subsentence, w'', to *inl*, $\mathsf{P}(pin \geq \mathsf{P}(inl)$ and *poin* is not prefixcaptured in w''. This case is analogous to case 2.

4. *lt inr rt*
   Analogous to *lt inl rt*.

□

Lemma 18 *If a prefix or infix operator, pin, appears in a syntax tree, t, and pin is not covered by a postfix operator then either*

- *t is not precedence correct, or*
- $\mathsf{Rw}(t) \geq \mathsf{P}(pin)$

Proof: The proof is by induction on the number of operators in a syntax tree which satisfies the preconditions.

Base:
A syntax tree with one operator. The operator must be *pin* since *pin* appears in *t*. Then

$\mathsf{Rw}(t) = \mathsf{P}(pin)$ and we have proved the base case.

Induction step:
Assume that the lemma holds for all syntax trees with less than n operators and show it for all syntax trees with n operators.

Case analysis: on the kind of syntax trees

1. *st post*
   The prefix or infix operator *pin* appears somewhere in *st* and since *st* has less than n operators the induction assumption gives that either

   $$st \text{ is not precedence correct} \tag{35}$$

   or

   $$\mathsf{Rw}(st) \geq \mathsf{P}(pin) \tag{36}$$

   If 35 holds then we are ready since then *st post* cannot be precedence correct either. From the conditions in the lemma we have that *pin* is not covered by a postfix operator. Thus,

   $$\mathsf{P}(pin) > \mathsf{P}(post) \tag{37}$$

   So if 36 holds and we combine 36 and 37 we get that the condition in the rule for postfix trees fails. Thus t is not precedence correct.

2. *pre st*
   If $pre = pin$ then we easily prove the lemma since

   $$\mathsf{Rw}(pre\ st) = max(\underbrace{\mathsf{P}(pre)}_{=\mathsf{P}(pin)}, \mathsf{Rw}(st)) \geq \mathsf{P}(pin) \tag{38}$$

   If $pre \neq pin$ then *pin* appears somewhere in *st* and since *st* has less than n operators the induction assumption gives that either

   $$st \text{ is not precedence correct} \tag{39}$$

   or

   $$\mathsf{Rw}(st) \geq \mathsf{P}(pin) \tag{40}$$

   If 39 holds then we are ready since then *pre st* cannot be precedence correct either. If 40 holds then we prove the lemma by stating that

   $$\mathsf{Rw}(pre\ st) = max(\mathsf{P}(pre), \underbrace{\mathsf{Rw}(st)}_{\geq \mathsf{P}(pin)}) \geq \mathsf{P}(pin) \tag{41}$$

3. An infix operator, $lt\ in\ rt$
   If $in = pin$ then

$$\mathsf{Rw}(lt\ in\ rt) = max(\underbrace{\mathsf{P}(in)}_{=\mathsf{P}(pin)}, \mathsf{Rw}(rt)) \geq \mathsf{P}(pin) \tag{42}$$

   Otherwise we have two cases, pin appears in lt or pin appears in rt. The first case is analogous to the postfix operator case and the second case is analogous to the prefix operator case.

□

Lemma 19 *If a postfix or infix operator, poin, appears in a syntax tree, t, and poin is not covered by a prefix operator then either*

- *t is not precedence correct, or*
- $\mathsf{Lw}(t) \geq \mathsf{P}(poin)$

Proof: Analogous to the proof of lemma 18. □

Appendix B

A complete proof of algorithm $\mathcal{M}$

Theorem 15 *If $\mathcal{T}(\mathcal{M}(H))$ is the generated language of syntax trees from the grammar $\mathcal{M}(H)$ and $\mathcal{T}(H) = \{t : \mathsf{Pc}_H(t)\}$ then, for every precedence grammar H, the language $\mathcal{T}(H)$ is equal to the language $\mathcal{T}(\mathcal{M}(H))$.*

Proof: The proof of theorem 15 consists of two parts:

1. $\mathcal{T}(H) \subseteq \mathcal{T}(\mathcal{M}(H))$
2. $\mathcal{T}(\mathcal{M}(H)) \subseteq \mathcal{T}(H)$

In the first part we show that if a syntax tree is precedence correct, that is, if it can be generated from a precedence grammar H, then it can be generated from the grammar we obtain by applying the precedence removing algorithm $\mathcal{M}$ on H. In the second part we show that if a syntax tree is generated from a grammar we have obtained by applying the algorithm on a precedence grammar, then the syntax tree is precedence correct.

1 Proof of $\mathcal{T}(H) \subseteq \mathcal{T}(\mathcal{M}(H))$

We show that:

$$\forall H.\ \forall t.\ \forall (n,p,q).\ E(n,p,q) \in \mathcal{M}(H) \wedge Q_H(t,n,p,q) \Longrightarrow E(n,p,q) \rightarrow^* t \qquad \text{(C43)}$$

The proof is by induction on H.

Base:
There are no operators in H, that is the only syntax tree is an atomic tree. Clearly C43 holds since the only production in $\mathcal{M}(H)$ is:

$$E(0,0,0) \ ::= \ AE \qquad (44)$$

Induction step:
Assume (indhyp 1)

$$\forall t.\ \forall (n,p,q).\ E(n,p,q) \in \mathcal{M}(H_{m-1}) \wedge Q_{H_{m-1}}(t,n,p,q) \Longrightarrow E(n,p,q) \rightarrow^* t \qquad \text{(A45)}$$

Show

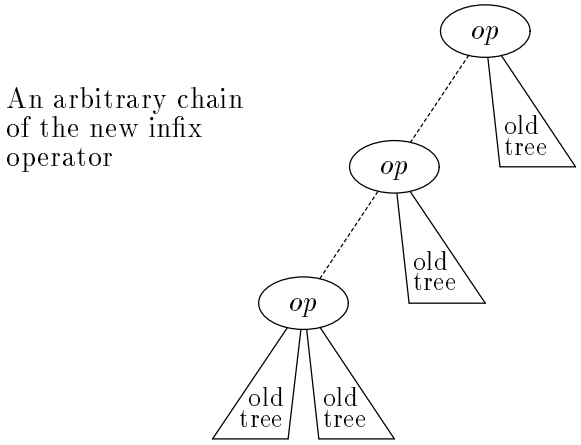
$$\forall t.\ \forall (n,p,q).\ E(n,p,q) \in \mathcal{M}(H_m) \wedge Q_{H_m}(t,n,p,q) \Longrightarrow E(n,p,q) \rightarrow^* t \qquad \text{(C46)}$$

Case analysis: on the introduced operator

1. $op \in Left$
   In this case only one new nonterminal ($E(m,0,0)$) is introduced and the only two new productions in $\mathcal{M}(H_m)$ are:

$$\begin{array}{lcl} E(m,0,0) & ::= & E(m,0,0)\ op\ E(m-1,0,0) \\ E(m,0,0) & ::= & E(m-1,0,0) \end{array} \tag{47}$$

   From lemma 20 it follows that all syntax trees in $\mathcal{T}(H_m)$ have the following form:



   The proof is by induction of the length on the chain of the new operator.
   (t^l denotes a syntax tree with a chain of length l)

   Base:
   The length is zero.
   Show

$$\forall(n,p,q).\ E(n,p,q) \in \mathcal{M}(H_m) \wedge Q_{H_m}(t^0,n,p,q) \Longrightarrow E(n,p,q) \rightarrow^* t^0 \tag{C48}$$

   Take an arbitrary triple (n,p,q).

   Assume

$$E(n,p,q) \in \mathcal{M}(H_m) \wedge Q_{H_m}(t^0,n,p,q) \tag{A49}$$

   and then show

$$E(n,p,q) \rightarrow^* t^0 \tag{C50}$$

   We have two cases:

   1. $E(n,p,q) \in \mathcal{M}(H_{m-1})$
      Since the new infix operator does not occur in t^0, it follows from assumption A49 and property 1 (page 84) that:

$$Q_{H_{m-1}}(t^0,n,p,q) \tag{51}$$

Statement C50 then follows from indhyp 1 (A45) together with 51 and the assumption that $E(n,p,q) \in \mathcal{M}(H_{m-1})$

2. $E(n,p,q) = E(m,0,0)$
Assumption A49 becomes in this case:

$$Q_{H_m}(t^0, m, 0, 0) \tag{52}$$

and since *op* does not occur in t^0 we have

$$Q_{H_{m-1}}(t^0, m-1, 0, 0) \tag{53}$$

From indhyp 1 (A45) and 53 and the fact that $E(m-1,0,0) \in \mathcal{M}(H_{m-1})$ it follows that:

$$E(m-1,0,0) \rightarrow^* t^0 \tag{54}$$

Since we know from 47 that there is a production

$$E(m,0,0) \ ::= \ E(m-1,0,0) \tag{55}$$

in $\mathcal{M}(H_m)$ we get that:

$$E(m,0,0) \rightarrow^* t^0 \tag{56}$$

That is, we have shown C50 for the second case.

This concludes the base (C48) in the induction on the chain of new infix operators.

Induction step:
Assume (indhyp 2)

$$\forall(n,p,q).\ E(n,p,q) \in \mathcal{M}(H_m) \wedge Q_{H_m}(t^l, n, p, q) \Longrightarrow E(n,p,q) \rightarrow^* t^l \tag{A57}$$

Show

$$\begin{array}{l}\forall(n,p,q).\ E(n,p,q) \in \mathcal{M}(H_m) \wedge Q_{H_m}(t^{l+1}, n, p, q) \Longrightarrow \\ E(n,p,q) \rightarrow^* t^{l+1}\end{array} \tag{C58}$$

It suffices to show the statement for $(m,0,0)$ since Q does not hold otherwise. (We know that $t^{l+1} = t^l \ op \ t'$ and $\mathsf{P}(op) = m$.)
That is we have to show:

$$Q_{H_m}(t^{l+1}, m, 0, 0) \Longrightarrow E(m,0,0) \rightarrow^* t^{l+1} \tag{C59}$$

Assume

$$Q_{H_m}(t^{l+1}, m, 0, 0) \tag{A60}$$

and then show

$$E(m,0,0) \rightarrow^* t^{l+1} \tag{C61}$$

Statement C61 follows from the following facts:

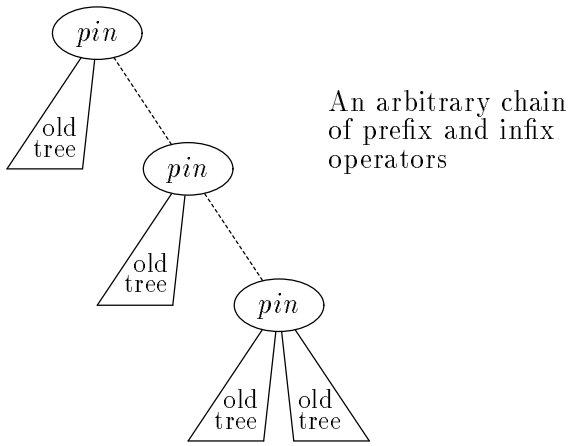
- There is at least one *op* in the chain, so $t^{l+1} = t^l\ op\ t'$
- There is a production:
  $E(m,0,0)\ ::=\ E(m,0,0)\ \ op\ \ E(m-1,0,0)$. (follows from 47)
- t^l has a chain of *op*:s of length l and can therefore be derived from $E(m,0,0)$. (indhyp 2, A57)
- $t' \in \mathcal{T}(H_{m-1})$ and can therefore be derived from $E(m-1,0,0)$. (indhyp 1, A45)

This concludes the induction step and we have shown C46 for the case that the last introduced operator is a left associative infix operator.

2. $op \in Right$
   Analogous with $op \in Left$

3. $op \in Pre$
   In this case a lot of new nonterminals and productions are introduced.

   From lemma 21 it follows that the syntax trees have the following form, where *pin* either denote an infix operator or a prefix operator. The left subtrees are only present in case of infix operators. The prefix operators in the chain can either be the new operator or some old one.



   We show C46 by induction on the length of the chain of infix and prefix operators.

   Base:
   The length is zero, that is t is either an atmic tree or $t = t'\ post_j$, for some j, and $t \in \mathcal{T}(H_{m-1})$.
   If t is an atomic tree then C46 follows from lemma 22 which tells us that an atomic tree can be derived from all nonterminals. So let us consider the case where $t = t'\ post_j$. Show

$$\begin{array}{l}\forall(n,p,q).\ E(n,p,q) \in \mathcal{M}(H_m) \wedge Q_{H_m}(t'\ post_j, n, p, q) \Longrightarrow \\ E(n,p,q) \rightarrow^*\ t'\ post_j\end{array} \qquad \text{(C62)}$$

   Take an arbitrary triple (n,p,q).

Assume

$$E(n,p,q) \in \mathcal{M}(H_m) \wedge Q_{H_m}(t'\ post_j, n, p, q) \tag{A63}$$

and then show

$$E(n,p,q) \rightarrow^* t'\ post_j \tag{C64}$$

From lemma 23 we get

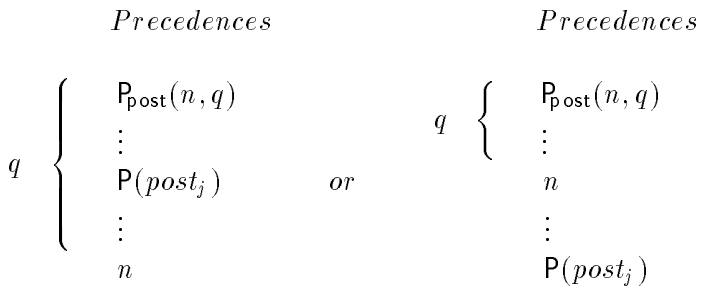
$$E(n,p,q) \rightarrow^* E(0, p+p', q+q') \tag{65}$$

where p' and q' are the number of prefix and postfix operators respectively between 0 and n.

Since $post_j$ of course occurs uncovered in $t'\ post_j$ it follows from assumption A63 that

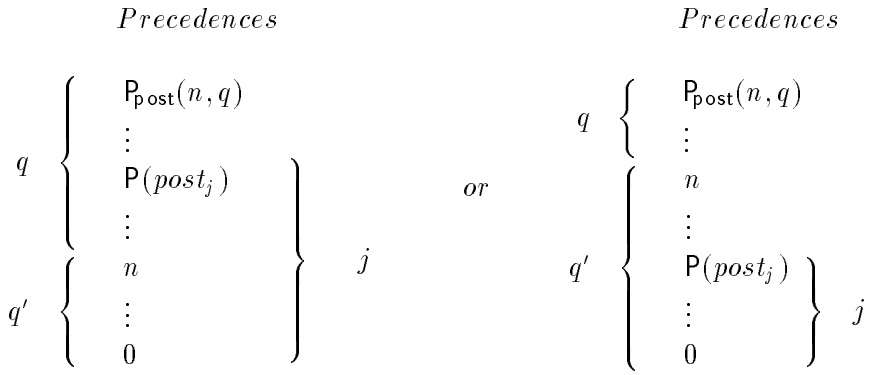
$$\mathsf{P}(post_j) \leq \mathsf{P}_{\mathsf{post}}(n,q) \tag{66}$$

Thus:



In this picture (and in following similar ones) the precedences are given in decreasing precedence order, that is, for the left picture we have $\mathsf{P}_{\mathsf{post}}(n,p) > \mathsf{P}(post_j) > n$ and q is the number of postfix operators between n and $\mathsf{P}_{\mathsf{post}}(n,q)$. Compare also with example 17.

Since we know from 65 that there are q' postfix operators between 0 and n and from the definition of $post_j$ that there are j postfix operators between 0 and $\mathsf{P}(post_j)$ we get:



that is

$$1 \leq j \leq q + q' \tag{67}$$

From the A-rule:

$$\begin{array}{l} E(0,p,q) \ ::= \ E(\mathsf{P}(post_j),0,q-j) \ \ post_j \qquad \text{where } 1 \leq j \leq q \\ \text{where } 0 \leq p \leq \text{number of prefix operators} \\ \qquad 0 \leq q \leq \text{number of postfix operators} \end{array} \tag{68}$$

we see that there exists a production

$$E(0,p+p',q+q') \ ::= \ E(\mathsf{P}(post_j),0,q+q'-j) \ \ post_j \tag{69}$$

since $1 \leq j \leq q + q'$. (67)

Now it remains to show

$$E(\mathsf{P}(post_j),0,q+q'-j) \rightarrow^* t' \tag{C70}$$

We would like to use indhyp 1 (A45) to show this so we start by proving the premisses. We know that $post_j \in H_{m-1}$ and that there are at least $q + q' - j$ postfix operators in H_{m-1} with higher precedence than $\mathsf{P}(post_j)$. (from the rules and A63) From the rule for postfix operators

$$\begin{array}{l} E(\mathsf{P}(post_i),p,q) \ ::= \ E(\mathsf{P}(post_i)-1,p,q+1) \\ \text{where } 1 \leq i \leq \text{number of postfix operators} \\ \qquad 0 \leq p \leq \text{number of prefix operators with higher precedence than } \mathsf{P}(post_i) \\ \qquad 0 \leq q \leq \text{number of postfix operators with higher precedence than } \mathsf{P}(post_i) \end{array} \tag{71}$$

it follows

$$E(\mathsf{P}(post_j),0,q+q'-j) \in \mathcal{M}(H_{m-1}) \tag{72}$$

Now we show that the desired instance of Q holds, that is

$$Q_{H_{m-1}}(t',\mathsf{P}(post_j),0,q+q'-j) \tag{C73}$$

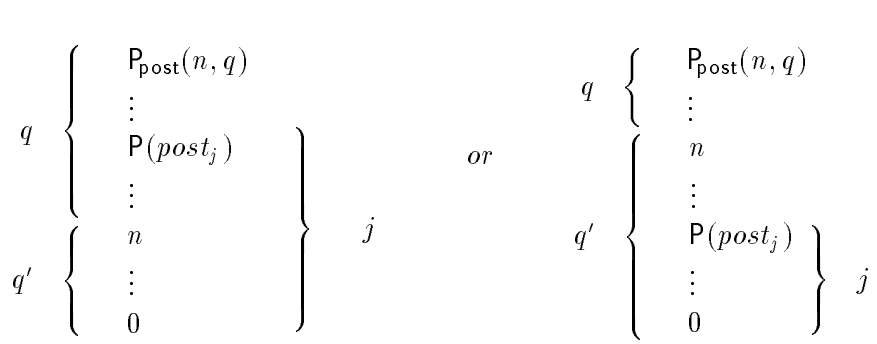
1. Since $t' \in \mathcal{T}(H_m)$ and the new prefix operator does not occur in t' it follows that

$$t' \in \mathcal{T}(H_{m-1}) \tag{74}$$

2. We know that there are q' postfix operators between 0 and n and that there are j postfix operators between 0 and $\mathsf{P}(post_j)$. So

$$\mathsf{P}_{\mathsf{post}}(\mathsf{P}(post_j),q+q'-j) = \mathsf{P}_{\mathsf{post}}(n,q) \tag{75}$$

This can also be seen from the following picture:



Furthermore we have

$$\begin{aligned} max(\mathsf{P}_{\mathsf{pre}}(\mathsf{P}(post_j),0),\mathsf{P}_{\mathsf{post}}(\mathsf{P}(post_j,q+q'-j)) &= \\ max(\mathsf{P}(post_j),\mathsf{P}_{\mathsf{post}}(\mathsf{P}(post_j,q+q'-j)) &= \\ \mathsf{P}_{\mathsf{post}}(\mathsf{P}(post_j,q+q'-j)) &= \\ \mathsf{P}_{\mathsf{post}}(n,q) & \end{aligned} \tag{76}$$

and

$$\mathsf{P}_{\mathsf{pre}}(\mathsf{P}(post_j),0) = \mathsf{P}(post_j) \tag{77}$$

We know the following about the operators in t':

- An uncovered postfix operator with higher precedence than $\mathsf{P}_{\mathsf{post}}(n,q)$ does not occur in t' since we know that it does not occur in $t'\ post_j$ (A63).
- An uncovered prefix or infix operator with higher precedence than $\mathsf{P}(post_j)$ does not occur in t' since $t'\ post_j$ is precedence correct.

Now we can say that the following operators do not occur in t':

a. An operator op for which $\mathsf{P}(op) > \mathsf{P}_{\mathsf{post}}(n,q)$. This follows since there do not occur any uncovered operator with higher precedence than $\mathsf{P}_{\mathsf{post}}(n,q)$ and then no operator can cover another.
b. An uncovered prefix operator op for which $\mathsf{P}(op) > \mathsf{P}(post_j)$.
c. An uncovered postfix operator op for which $\mathsf{P}(op) > \mathsf{P}_{\mathsf{post}}(n,q)$.
d. An uncovered infix operator op for which $\mathsf{P}(op) > \mathsf{P}(post_j)$.

That concludes the proof of C73 and we can use indhyp 1 (A45) together with 72 and C73 to get C70.

Statement C64, that is

$$E(n,p,q) \rightarrow^* t'\ post_j$$

follows from the three results

$$\begin{aligned} &(65) \quad E(n,p,q) \rightarrow^* E(0,p+p',q+q') \\ &(69) \quad E(0,p+p',q+q') \ ::= \ E(\mathsf{P}(post_j),0,q+q'-j)\ \ post_j \\ &(\mathrm{C70}) \quad E(\mathsf{P}(post_j),0,q+q'-j) \rightarrow^* t' \end{aligned}$$

Since the triple (n,p,q) was arbitrary, we have shown C62, that is the base in the induction on the length of the chain of infix and prefix operators.

Induction step:
Assume (indhyp 3)

$$\forall(n,p,q).\ E(n,p,q) \in \mathcal{M}(H_m) \wedge Q_{H_m}(t^l,n,p,q) \Longrightarrow E(n,p,q) \rightarrow^* t^l \quad \text{(A78)}$$

Show

$$\begin{array}{l}\forall(n,p,q).\ E(n,p,q) \in \mathcal{M}(H_m) \wedge Q_{H_m}(t^{l+1},n,p,q) \Longrightarrow \\ E(n,p,q) \rightarrow^* t^{l+1}\end{array} \quad \text{(C79)}$$

(t^l denotes a syntax tree with a chain of infix and prefix operators of length l)

Take an arbitrary triple (n,p,q).

Assume

$$E(n,p,q) \in \mathcal{M}(H_m) \wedge Q_{H_m}(t^{l+1},n,p,q) \quad \text{(A80)}$$

and then show

$$E(n,p,q) \rightarrow^* t^{l+1} \quad \text{(C81)}$$

Case analysis: on t^{l+1}

i. $t^{l+1} = pre_i\ t^l$
Show

$$E(n,p,q) \rightarrow^* pre_i\ t^l \quad \text{(C82)}$$

From lemma 23 we get

$$E(n,p,q) \rightarrow^* E(0,p+p',q+q') \quad (83)$$

where p' and q' are the number of prefix and postfix operators respectively between 0 and n.
Since pre_i of course occurs uncovered in $pre_i\ t^l$ it follows from assumption A80 that:

$$\mathsf{P}(pre_i) \leq \mathsf{P}_{\mathsf{pre}}(n,p) \quad (84)$$

Since we know from 83 that there are p' prefix operators between 0 and n and from the definition of pre_i that there are i prefix operators between 0 and $\mathsf{P}(pre_i)$ we get

$$1 \leq i \leq p + p' \quad (85)$$

From the A-rule:

$$\begin{array}{l}E(0,p,q)\ ::=\ pre_i\ \ E(\mathsf{P}(pre_i),p-i,0) \qquad \text{where } 1 \leq i \leq p \\ \text{where } 0 \leq p \leq \text{number of prefix operators} \\ \qquad 0 \leq q \leq \text{number of postfix operators}\end{array} \quad (86)$$

we get that there exists a production

$$E(0, p+p', q+q') \ ::= \ pre_i \ \ E(\mathsf{P}(pre_i), p+p'-i, 0) \tag{87}$$

since $1 \leq i \leq p+p'$. (85)

Now it remains to show

$$E(\mathsf{P}(pre_i), p+p'-i, 0) \rightarrow^* t^l \tag{C88}$$

We will use indhyp 3 (A78) to prove this so we start by proving the premisses. We know that there are $p+p'-i$ prefix operators with higher precedence than $\mathsf{P}(pre_i)$ so from the rule for prefix operators:

$$\begin{array}{l} E(\mathsf{P}(pre_i), p, q) \ ::= \ E(\mathsf{P}(pre_i) - 1, p+1, q) \\ \text{where } 1 \leq i \leq \text{number of prefix operators} \\ \qquad 0 \leq p \leq \text{number of prefix operators with higher precedence than } \mathsf{P}(pre_i) \\ \qquad 0 \leq q \leq \text{number of postfix operators with higher precedence than } \mathsf{P}(pre_i) \end{array} \tag{89}$$

it follows

$$E(\mathsf{P}(pre_i), p+p'-i, 0) \in \mathcal{M}(H_m) \tag{90}$$

Now we show that the desired instance of Q holds, that is

$$Q_{H_m}(t^l, \mathsf{P}(pre_i), p+p'-i, 0) \tag{C91}$$

1. $t^l \in \mathcal{T}(H_m)$ since $t^{l+1} \in \mathcal{T}(H_m)$
2. We know from 83 that there are p' prefix operators between 0 and n and from the definition of pre_i that there are i prefix operators between 0 and $\mathsf{P}(pre_i)$. So

   $$\mathsf{P}_{\mathsf{pre}}(\mathsf{P}(pre_i), p+p'-i) = \mathsf{P}_{\mathsf{pre}}(n, p) \tag{92}$$

   We also have

   $$\begin{array}{rl} max(\mathsf{P}_{\mathsf{pre}}(\mathsf{P}(pre_i), p+p'-i), \mathsf{P}_{\mathsf{post}}(\mathsf{P}(pre_i, 0)) & = \\ max(\mathsf{P}_{\mathsf{pre}}(\mathsf{P}(pre_i), p+p'-i), \mathsf{P}(pre_i, 0) & = \\ \mathsf{P}_{\mathsf{pre}}(\mathsf{P}(pre_i), p+p'-i) & = \\ \mathsf{P}_{\mathsf{pre}}(n, p) & \end{array} \tag{93}$$

   and

   $$\mathsf{P}_{\mathsf{post}}(\mathsf{P}(pre_i), 0) = \mathsf{P}(pre_i) \tag{94}$$

   We know the following about the operators in t^l:

   - An uncovered prefix operator with higher precedence than $\mathsf{P}_{\mathsf{pre}}(n, p)$ does not occur in t^l since we know from assumption A80 that it does not occur in $pre_i \ t^l$.
   - An uncovered postfix or infix operator with higher precedence than $\mathsf{P}(pre_i)$ does not occur in t^l since we know that $pre_i \ t^l$ is precedence correct.

   Now we can say that the following operators do not occur in t^l:

   a. An operator op for which $\mathsf{P}(op) > \mathsf{P}_{\mathsf{pre}}(n, p)$.
   b. An uncovered prefix operator op for which $\mathsf{P}(op) > \mathsf{P}_{\mathsf{pre}}(n, p)$.
   c. An uncovered postfix operator op for which $\mathsf{P}(op) > \mathsf{P}(pre_i)$.
   d. An uncovered infix operator op for which $\mathsf{P}(op) > \mathsf{P}(pre_i)$.

This concludes the proof of C91. We can also see that C88 follows from indhyp 3 (A78) together with 90, C91 and the fact that t^l has a chain of prefix and infix operators of length l.

Statement C82, that is

$$E(n,p,q) \rightarrow^* pre_i\ t^l$$

follows from the three results

$$\begin{array}{rl} (83) & E(n,p,q) \rightarrow^* E(0,p+p',q+q') \\ (87) & E(0,p+p',q+q') \ ::= \ pre_i \ \ E(\mathsf{P}(pre_i),p+p'-i,0) \\ (C88) & E(\mathsf{P}(pre_i),p+p'-i,0) \rightarrow^* t^l \end{array}$$

This concludes the first case in the induction step, that is we have shown C81 for the case that the chain starts with a prefix operator.

ii. $t^{l+1} = t'\ inl_i\ t^l$
Show

$$E(n,p,q) \rightarrow^* t'\ inl_i\ t^l \tag{C95}$$

We just give an overview of the proof. Statement C95 follows from C96, C97, C98 and C99.

$$E(n,p,q) \rightarrow^* E(\mathsf{P}(inl_i),p+p',q+q') \tag{C96}$$

where p' and q' is the number of prefix operators and postfix operators between n and $\mathsf{P}(inl_i)$

$$\begin{array}{l} E(\mathsf{P}(inl_i),p+p',q+q') \ ::= \\ E(\mathsf{P}(inl_i),0,q+q') \ \ inl_i \ \ E(\mathsf{P}(inl_i)+1,p+p',0) \end{array} \tag{C97}$$

$$E(\mathsf{P}(inl_i),0,q+q') \rightarrow^* t' \tag{C98}$$

$$E(\mathsf{P}(inl_i)-1,p+p',0) \rightarrow^* t^l \tag{C99}$$

Statement C96 follows from lemma 23 and C97 follows from the rule for left associative infix operators. To show C98 we use indhyp 1 (A45) and to show C99 we use indhyp 3 (A78).

iii. $t^{l+1} = t'\ inr_i\ t^l$
Analogous with $t^{l+1} = t'\ inl_i\ t^l$.

4. $op \in Post$
Analogous with $op \in Pre$

2 Proof of $\mathcal{T}(\mathcal{M}(H)) \subseteq \mathcal{T}(H)$

We show that

$$\forall H.\forall t.\forall (n,p,q).\ E(n,p,q) \rightarrow^* t \Longrightarrow Q_H(t,n,p,q) \tag{C100}$$

Take an arbitrary H and show

$$\forall t.\forall (n,p,q).\ E(n,p,q) \rightarrow^* t \Longrightarrow Q_H(t,n,p,q) \tag{C101}$$

The proof is by induction on the length of the derivation.

Base:
Show

$$\forall t.\forall(n,p,q).\ E(n,p,q) \rightarrow^1 t \Longrightarrow Q_H(t,n,p,q) \tag{C102}$$

The only syntax tree we can derive in one step is an atomic tree so C102 holds since

$$\forall(n,p,q).Q_H(AE,n,p,q) \tag{103}$$

Induction step:
Assume

$$\forall t.\forall(n,p,q).\ E(n,p,q) \rightarrow^r t \Longrightarrow Q_H(t,n,p,q) \qquad \text{where } 1 \leq r < \gamma \tag{A104}$$

Show

$$\forall t.\forall(n,p,q).\ E(n,p,q) \rightarrow^\gamma t \Longrightarrow Q_H(t,n,p,q) \tag{C105}$$

Take arbitrary t and (n,p,q).
Assume

$$E(n,p,q) \rightarrow^\gamma t \tag{A106}$$

Show

$$Q_H(t,n,p,q) \tag{C107}$$

Case analysis: on the first step in the derivation

1. $E(n,p,q) \rightarrow E(n-1,p+1,q)$
   That is, there is a prefix operator with precedence n. From A106 together with the "case assumption" it follows

   $$E(n-1,p+1,q) \rightarrow^{\gamma-1} t \tag{108}$$

   From assumption A104 together with 108 it follows

   $$Q_H(t,n-1,p+1,q) \tag{109}$$

   that is

   1. $t \in \mathcal{T}(H)$, and $\mathsf{Pc}_H(t)$ holds
   2. The following operators do not occur in t.
      a. An operator *op* for which $\mathsf{P}(op) > max(\mathsf{P}_{\mathsf{pre}}(n-1,p+1), \mathsf{P}_{\mathsf{post}}(n-1,q))$.
      b. An uncovered prefix operator *op* for which $\mathsf{P}(op) > \mathsf{P}_{\mathsf{pre}}(n-1,p+1)$.
      c. An uncovered postfix operator *op* for which $\mathsf{P}(op) > \mathsf{P}_{\mathsf{post}}(n-1,q)$.
      d. An uncovered infix operator *op* for which $\mathsf{P}(op) > n-1$.

Statement C107 follows almost immediately from 109. Since there is a prefix operator with precedence n there cannot be an infix operator with the same precedence. It is therefore the same to say that an infix operator op for which $\mathsf{P}(op) > n$ does not occur in t as to say that an infix operator for which $\mathsf{P}(op) > n-1$ does not occur in t. We also have

$$\begin{aligned} &\mathsf{P}_{\mathsf{pre}}(n-1, p+1) = \mathsf{P}_{\mathsf{pre}}(n, p) \\ &\mathsf{P}_{\mathsf{post}}(n-1, q) = \mathsf{P}_{\mathsf{post}}(n, q) \end{aligned} \tag{110}$$

2. $E(n,p,q) \rightarrow E(n-1,p,q+1)$
That is, there is a postfix operator with precedence n. Analogous with the previous case.

3. $E(n,p,q) \rightarrow E(n,0,q)\ \ inop\ \ E(n-1,p,0)$
That is, $inop$ is a left associative infix operator with precedence n and $t = lt\ \ inop\ \ rt$. We have

$$E(n,0,q) \rightarrow^{r_1} lt \qquad \text{where } r_1 < \gamma \tag{111}$$

and

$$E(n-1,p,0) \rightarrow^{r_2} rt \qquad \text{where } r_2 < \gamma \tag{112}$$

Assumption A104 together with 111 gives

$$Q_H(lt, n, 0, q) \tag{113}$$

that is

$$lt \in \mathcal{T}(H), \quad \text{and} \quad \mathsf{Pc}_H(lt) \quad \text{holds} \tag{114}$$

$$\begin{aligned} &\text{An operator } op \text{ for which} \\ &\mathsf{P}(op) > \underbrace{max(\mathsf{P}_{\mathsf{pre}}(n,0), \mathsf{P}_{\mathsf{post}}(n,q))}_{=\mathsf{P}_{\mathsf{post}}(n,q)} \\ &\text{does not occur in } lt \end{aligned} \tag{115}$$

$$\begin{aligned} &\text{An uncovered prefix operator } op \text{ for which} \\ &\mathsf{P}(op) > \mathsf{P}_{\mathsf{pre}}(n,0) = n = \mathsf{P}(inop) \\ &\text{does not occur in } lt. \end{aligned} \tag{116}$$

$$\begin{aligned} &\text{An uncovered postfix operator } op \text{ for which} \\ &\mathsf{P}(op) > \mathsf{P}_{\mathsf{post}}(n,q) \\ &\text{does not occur in } lt. \end{aligned} \tag{117}$$

$$\begin{aligned} &\text{An uncovered infix operator } op \text{ for which} \\ &\mathsf{P}(op) > n = \mathsf{P}(inop) \\ &\text{does not occur in } lt. \end{aligned} \tag{118}$$

Lemma 24 gives together with 116 and 118

$$\mathsf{Rw}(lt) \leq \mathsf{P}(inop) \tag{119}$$

Assumption A104 together with 112 gives

$$Q_H(rt, n-1, p, 0) \tag{120}$$

that is

$$rt \in \mathcal{T}(H), \quad \text{and} \quad \mathsf{Pc}_H(rt) \quad \text{holds} \tag{121}$$

$$\begin{array}{l}\text{An operator } op \text{ for which} \\ \mathsf{P}(op) > \underbrace{max(\mathsf{P}_{\mathsf{pre}}(n-1,p), \mathsf{P}_{\mathsf{post}}(n-1,0))}_{=\mathsf{P}_{\mathsf{pre}}(n-1,p)=\mathsf{P}_{\mathsf{pre}}(n,p)} \\ \text{does not occur in } rt.\end{array} \tag{122}$$

$$\begin{array}{l}\text{An uncovered prefix operator } op \text{ for which} \\ \mathsf{P}(op) > \mathsf{P}_{\mathsf{pre}}(n-1,p) = \mathsf{P}_{\mathsf{pre}}(n,p) \\ \text{does not occur in } rt.\end{array} \tag{123}$$

$$\begin{array}{l}\text{An uncovered postfix operator } op \text{ for which} \\ \mathsf{P}(op) > \mathsf{P}_{\mathsf{post}}(n-1,0) = n-1 \\ \text{does not occur in } rt.\end{array} \tag{124}$$

$$\begin{array}{l}\text{An uncovered infix operator } op \text{ for which} \\ \mathsf{P}(op) > n-1 \\ \text{does not occur in } rt.\end{array} \tag{125}$$

Lemma 25 gives together with 124 and 125

$$\mathsf{Lw}(rt) \leq n-1 < n = \mathsf{P}(inop) \tag{126}$$

From the Pc-rule for left associative infix operators

$$\frac{op \in Left \quad \mathsf{Pc}_H(lt) \quad \mathsf{Pc}_H(rt) \quad \mathsf{Rw}(lt) \leq \mathsf{P}(inop) \quad \mathsf{Lw}(rt) < \mathsf{P}(inop)}{\mathsf{Pc}_H(lt\ inop\ rt)} \tag{127}$$

together with 114, 121, 119 and 126 it follows that

$$\mathsf{Pc}_H(lt\ inop\ rt) \quad \text{holds, and} \quad t \in \mathcal{T}(H) \tag{128}$$

From 115 and 122 it follows:

$$\begin{array}{l}\text{An operator } op \text{ for which} \\ \mathsf{P}(op) > max(\mathsf{P}_{\mathsf{pre}}(n,p), \mathsf{P}_{\mathsf{post}}(n,q)) \\ \text{does not occur in } lt\ inop\ rt.\end{array} \tag{129}$$

From 116 and 123 it follows:

$$\begin{array}{l}\text{An uncovered prefix operator } op \text{ for which}\\ \mathsf{P}(op) > \mathsf{P}_{\mathsf{pre}}(n,p)\\ \text{does not occur in } lt\ inop\ rt.\end{array} \tag{130}$$

From 117 and 124 it follows:

$$\begin{array}{l}\text{An uncovered postfix operator } op \text{ for which}\\ \mathsf{P}(op) > \mathsf{P}_{\mathsf{post}}(n,q)\\ \text{does not occur in } lt\ inop\ rt.\end{array} \tag{131}$$

From 118 and 125 it follows:

$$\begin{array}{l}\text{An uncovered infix operator } op \text{ for which}\\ \mathsf{P}(op) > n\\ \text{does not occur in } lt\ inop\ rt.\end{array} \tag{132}$$

Since we have 128, 129, 130, 131 and 132 we have proved C107 and since t and (n,p,q) were arbitrary we have proved C105 for the case that the first step in the derivation is $E(n,p,q) \rightarrow E(n,0,q)\ \ inop\ \ E(n-1,p,0)$.

4. $E(n,p,q) \rightarrow preop_i\ \ E(\mathsf{P}(preop_i),p-i,0)$
That is, $t = preop_i\ t'$ and $n = 0$.

From the A-rule

$$\begin{array}{l}E(0,p,q)\ ::=\ pre_i\ \ E(\mathsf{P}(pre_i),p-i,0) \qquad \text{where } 1 \le i \le p\\ \text{where } 0 \le p \le \text{number of prefix operators}\\ \qquad 0 \le q \le \text{number of postfix operators}\end{array} \tag{133}$$

we get

$$i \le p \tag{134}$$

We have

$$E(\mathsf{P}(preop_i),p-i,0) \rightarrow^r t' \qquad \text{where } r < \gamma \tag{135}$$

Assumption A104 together with 135 gives

$$Q_H(t',\mathsf{P}(preop_i),p-i,0) \tag{136}$$

that is

$$t' \in \mathcal{T}(H),\ \text{ and }\ \mathsf{Pc}_H(t')\ \text{ holds} \tag{137}$$

Before we say something about the operators in t' we note that

$$\mathsf{P}_{\mathsf{pre}}(\mathsf{P}(preop_i),p-i) = \mathsf{P}_{\mathsf{pre}}(0,p) \tag{138}$$

This can be seen from the following picture:

$$p \left\{ \begin{array}{c} \left. \begin{array}{c} \mathsf{P}_{\mathsf{pre}}(0,p) = \mathsf{P}_{\mathsf{pre}}(\mathsf{P}(preop_i), p-i) \\ \vdots \end{array} \right\} p-i \\ \left. \begin{array}{c} \mathsf{P}(preop_i) \\ \vdots \\ 0 \end{array} \right\} i \end{array} \right.$$

Now we say which operators do not occur in t'.

An operator op for which
$\mathsf{P}(op) > \underbrace{max(\mathsf{P}_{\mathsf{pre}}(\mathsf{P}(preop_i), p-i), \mathsf{P}_{\mathsf{post}}(\mathsf{P}(preop_i), 0))}_{=\mathsf{P}_{\mathsf{pre}}(0,p)}$ (139)
does not occur in t'.

An uncovered prefix operator op for which
$\mathsf{P}(op) > \underbrace{\mathsf{P}_{\mathsf{pre}}(\mathsf{P}(preop_i), p-i)}_{=\mathsf{P}_{\mathsf{pre}}(0,p)}$ (140)
does not occur in t'.

An uncovered postfix operator op for which
$\mathsf{P}(op) > \underbrace{\mathsf{P}_{\mathsf{post}}(\mathsf{P}(preop_i), 0)}_{=\mathsf{P}(preop_i)}$ (141)
does not occur in t'.

An uncovered infix operator op for which
$\mathsf{P}(op) > \mathsf{P}(preop_i)$ (142)
does not occur in t'.

Lemma 25 gives with 137 and 141 and 142

$$\mathsf{Lw}(t') \leq \mathsf{P}(preop_i) \tag{143}$$

Since there of course is a prefix operator with precedence $\mathsf{P}(preop_i)$ and the left weight of a prefix node is 0 we in fact have

$$\mathsf{Lw}(t') < \mathsf{P}(preop_i) \tag{144}$$

From the Pc-rule for prefix operators

$$\frac{op \in Pre \quad \mathsf{Pc}_H(t) \quad \mathsf{Lw}(t) < \mathsf{P}(op)}{\mathsf{Pc}_H(op\ t)} \tag{145}$$

it follows together with 137 and 144

$$\mathsf{Pc}_H(preop_i\ t') \text{ holds and therefore } t \in \mathcal{T}(H) \tag{146}$$

In the following statements about which operators that does not occur in $preop_i\ t'$ it is important to note that $n = 0$. From 139 it follows

$$\begin{array}{l}\text{An operator } op \text{ for which} \\ \mathsf{P}(op) > max(\mathsf{P}_{\mathsf{pre}}(n,p), \mathsf{P}_{\mathsf{post}}(n,q)) \\ \text{does not occur in } preop_i\ t'.\end{array} \tag{147}$$

From 140 and 134 it follows

$$\begin{array}{l}\text{An uncovered prefix operator } op \text{ for which} \\ \mathsf{P}(op) > \mathsf{P}_{\mathsf{pre}}(n,p) \\ \text{does not occur in } preop_i\ t'.\end{array} \tag{148}$$

From 141 and the fact that an operator with lower precedence than $\mathsf{P}(preop_i)$ is covered by $preop_i$ it follows

$$\begin{array}{l}\text{An uncovered postfix operator } op \text{ for which} \\ \mathsf{P}(op) > \mathsf{P}_{\mathsf{post}}(n,q) \\ \text{does not occur in } preop_i\ t'.\end{array} \tag{149}$$

From 142 and the fact that an operator with lower precedence than $\mathsf{P}(preop_i)$ is covered by $preop_i$ it follows

$$\begin{array}{l}\text{An uncovered infix operator } op \text{ for which} \\ \mathsf{P}(op) > n \\ \text{does not occur in } preop_i\ t'.\end{array} \tag{150}$$

Since we have 146, 147, 148, 149 and 150 we have proved C107 and since t and (n,p,q) were arbitrary we have proved C105 for the case that the first step in the derivation is $E(n,p,q) \rightarrow preop_i\ \ E(\mathsf{P}(preop_i), p-i, 0)$.

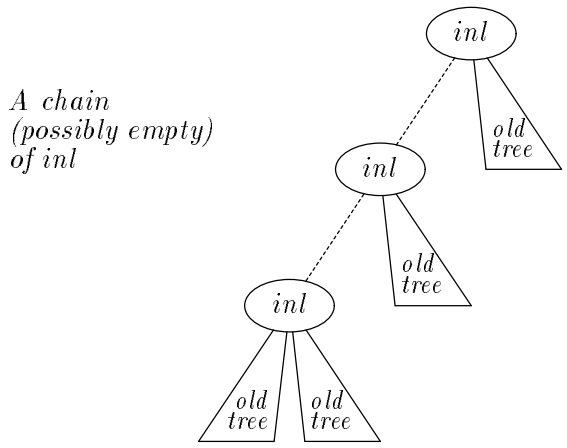
5. $E(n,p,q) \rightarrow E(\mathsf{P}(postop_i), 0, q-i)\ \ postop_i$
   Analogous to the preopcase

6. $E(n,p,q) \rightarrow E(n-1,p,q)$
   Trivial.

This concludes the induction and we have shown C101 and since $H$was arbitrary we have shown C100 and particularly $\mathcal{T}(\mathcal{M}(H)) \subseteq \mathcal{T}(H)$. □

3 Lemmas

Lemma 20 *Let H be a precedence grammar and let inl be a left associative infix operator in H with highest precedence of all operators in H. Then all syntax trees in $\mathcal{T}(H)$ have the following form:*



Proof: An old tree, that is a tree where inl does not occur, of course has the form above (with an empty chain of inl), so let us consider the case where inl is the root operator. Let $t = lt\ inl\ rt$ be a syntax tree in $\mathcal{T}(H)$. We show that it is not possible to build a precedence correct syntax tree where t is a subtree, unless the tree has inl as root operator and t is the left subtree, that is a tree of the form above. Both the left and right weight of t must be $\mathsf{P}(inl)$ since inl has highest precedence in the grammar.

Case analysis: on all ways of building a syntax tree in which t is a subtree.

1. $preop\ t$
   The tree is not precedence correct since $\mathsf{Lw}(t) = \mathsf{P}(inl) \not< \mathsf{P}(preop)$

2. $t\ postop$
   The tree is not precedence correct since $\mathsf{Rw}(t) = \mathsf{P}(inl) \not< \mathsf{P}(postop)$

3. $t\ inop\ t'$ where $inop \neq inl$.
   The tree is not precedence correct since $\mathsf{Rw}(t) = \mathsf{P}(inl) \not\leq \mathsf{P}(inop)$

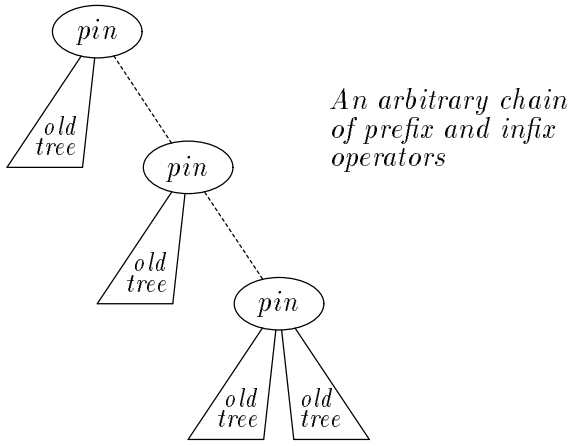
4. $t'\ inop\ t$ where $inop \neq inl$.
   The tree is not precedence correct since $\mathsf{Lw}(t) = \mathsf{P}(inl) \not\leq \mathsf{P}(inop)$

5. $t'\ inl\ t$
   The tree is not precedence correct since $\mathsf{Lw}(t) = \mathsf{P}(inl) \not< \mathsf{P}(inl)$

6. $t\ inl\ t'$
   The tree is precedence correct if $\mathsf{Pc}_H(t')$ and $\mathsf{Lw}(t') < \mathsf{P}(inl)$.
   We have $\mathsf{Rw}(t) = \mathsf{P}(inl) \leq \mathsf{P}(inl)$.

□

Lemma 21 *Let H be a precedence grammar and let pre be a prefix operator in H with the highest precedence of all operators in H. Then all syntax trees in $\mathcal{T}(H)$ have the following form, where pin either denote an infix operator or a prefix operator. The left subtrees are only present in case of infix operators. The prefix operators in the chain can either be pre or another.*



Proof: An old tree of course have the form above (with an empty chain of infix and prefix operators). The simplest case when a tree is not an old one is when pre is the root operator with an old tree as subtree. Such a tree is of the form above (with a chain of one prefix operator). It is also precedence correct (if the old tree is precedence correct) and the right weight of it is $\mathsf{P}(pre)$. We will now show that if a tree t of the form above for which $\mathsf{Rw}(t) = \mathsf{P}(pre)$ is a subtree of another tree t' and $\mathsf{Pc}(t')$ then t' must also be of the form above and $\mathsf{Rw}(t') = \mathsf{P}(pre)$.

Case analysis: on all ways of building a syntax tree t' in which t is a subtree.

1. $t' = preop\ t$ where either $preop = pre$ or $preop \neq pre$
   The tree t' is of the form above and $\mathsf{Rw}(t') = \mathsf{P}(pre)$ since pre has the highest precedence

$$\mathsf{Rw}(preop\ t) = \underbrace{max(\mathsf{P}(preop), \underbrace{\mathsf{Rw}(t)}_{=\mathsf{P}(pre)})}_{=\mathsf{P}(pre)}$$

2. $t' = t\ postop$
   The tree t' is not precedence correct since $\mathsf{Rw}(t) = \mathsf{P}(pre) \not< \mathsf{P}(postop)$.

3. $t' = t\ inop\ rt$
   The tree t' is not precedence correct since $\mathsf{Rw}(t) = \mathsf{P}(pre) \not\leq \mathsf{P}(inop)$.

4. $t' = lt\ inop\ t$
   If lt is an old tree then the tree t' is of the form above and $\mathsf{Rw}(t') = \mathsf{P}(pre)$ since

$$\underbrace{\mathsf{Rw}(lt\ inop\ t) = max(\mathsf{P}(inop), \underbrace{\mathsf{Rw}(t)}_{=\mathsf{P}(pre)})}_{=\mathsf{P}(pre)}$$

   If lt is not an old tree then t' is not precedence correct.

□

Lemma 22 *Let H be a precedence grammar and $E(n,p,q)$ a nonterminal in the grammar $\mathcal{M}(H)$. Then it is possible to derive an atomic tree from $E(n,p,q)$.*

Proof: We show

$$\forall H.\forall n.\forall p.\forall q.E(n,p,q) \in \mathcal{M}(H) \Longrightarrow E(n,p,q) \rightarrow^* AE \tag{C151}$$

Take an arbitrary grammar H and show

$$\forall n.\forall p.\forall q.E(n,p,q) \in \mathcal{M}(H) \Longrightarrow E(n,p,q) \rightarrow^* AE \tag{C152}$$

The proof is by induction on n.

Base:
$n = 0$
Show

$$\forall p.\forall q.E(0,p,q) \in \mathcal{M}(H) \Longrightarrow E(0,p,q) \rightarrow^* AE \tag{C153}$$

Statement C153 follows immediately from the A-rule.

Induction step:
Assume

$$\forall p.\forall q.E(n-1,p,q) \in \mathcal{M}(H) \Longrightarrow E(n-1,p,q) \rightarrow^* AE \tag{A154}$$

Show

$$\forall p.\forall q.E(n,p,q) \in \mathcal{M}(H) \Longrightarrow E(n,p,q) \rightarrow^* AE \tag{C155}$$

Take arbitrary k and q and show

$$E(n,p,q) \in \mathcal{M}(H) \Longrightarrow E(n,p,q) \rightarrow^* AE \tag{C156}$$

Assume

$$E(n,p,q) \in \mathcal{M}(H) \tag{A157}$$

and then show

$$E(n,p,q) \rightarrow^* AE \tag{C158}$$

Case analysis: on the operator with precedence n.

1. infix operator:
   From the rules for infix operators it follows that there is a production

$$E(n,p,q) \ ::= \ E(n-1,p,q) \in \mathcal{M}(H) \tag{159}$$

   From assumption A154 together with 159 it follows that there is a derivation

$$E(n-1,p,q) \rightarrow^* AE \tag{160}$$

   Statement C158 follows from 159 and 160. Since k and q were arbitrary we have shown C155.

2. prefix operator:
   From the rule for prefix operators it follows that there is a production

$$E(n,p,q) \ ::= \ E(n-1,p+1,q) \in \mathcal{M}(H) \tag{161}$$

   The rest is analogous to the infix operator case.

3. postfix operator:
   From the rule for postfix operators it follows that there is a production

$$E(n,p,q) \ ::= \ E(n-1,p,q+1) \in \mathcal{M}(H) \tag{162}$$

   The rest is analogous to the infix operator case.

This concludes the induction and we have shown C152. Since H was arbitrary we have shown C151. □

Lemma 23 *For every precedence grammar H and all natural numbers n_1, n_2, p and q such that $n_1 \geq n_2$ and $E(n_1,p,q) \in \mathcal{M}(H)$ there is a derivation $E(n_1,p,q) \rightarrow^* E(n_2,p+p',q+q')$, where $p'(q')$ is the number of prefix(postfix) operators between n_2 and n_1.*

Proof: We will show

$$\begin{array}{l} \forall H.\ \forall n_1.\ \forall n_2.\ \forall p.\ \forall q.\ n_1 \geq n_2 \wedge E(n_1,p,q) \in \mathcal{M}(H) \Longrightarrow \\ E(n_1,p,q) \rightarrow^* E(n_2,p+p',q+q') \end{array} \tag{C163}$$

Take an arbitrary grammar H and show:

$$\begin{array}{l} \forall n_1.\ \forall n_2.\ \forall p.\ \forall q.\ n_1 \geq n_2 \wedge E(n_1,p,q) \in \mathcal{M}(H) \Longrightarrow \\ E(n,p,q) \rightarrow^* E(n_2,p+p',q+q') \end{array} \tag{C164}$$

We prove this by induction on n_1.

Base:
$n_1 = 0$
Show

$$\forall n_2.\ \forall p.\ \forall q.\ 0 \geq n_2 \wedge E(0,p,q) \in \mathcal{M}(H) \Longrightarrow \\ E(0,p,q) \rightarrow^* E(n_2, p+p', q+q') \tag{C165}$$

In order to satisfy the condition $0 \geq n_2$, n_2 must be 0 and also $p' = 0, q' = 0$. From that C165 follows immediately.

Induction step:
Assume (induction assumption)

$$\forall n_2.\ \forall p.\ \forall q.\ n_1 - 1 \geq n_2 \wedge E(n_1 - 1, p, q) \in \mathcal{M}(H) \Longrightarrow \\ E(n_1 - 1, p, q) \rightarrow^* E(n_2, p+p', q+q') \tag{A166}$$

Show

$$\forall n_2.\ \forall p.\ \forall q.\ n_1 \geq n_2 \wedge E(n_1, p, q) \in \mathcal{M}(H) \Longrightarrow \\ E(n_1, p, q) \rightarrow^* E(n_2, p+p', q+q') \tag{C167}$$

Take arbitrary n_2, p and q.
Assume

$$n_1 \geq n_2 \wedge E(n_1, p, q) \in \mathcal{M}(H) \tag{A168}$$

Show

$$E(n_1, p, q) \rightarrow^* E(n_2, p+p', q+q') \tag{C169}$$

where p' and q' are the number of prefix and postfix operators between n_2 and n_1.

If $n_1 = n_2$ then C169 follows immediately so let us consider the case that $n_1 > n_2$. There is an operator on each precedence level so one of the following productions are in the grammar:

$$\begin{array}{lcll} E(n_1,p,q) & ::= & E(n_1 - 1, p, q) & \textit{infix} \\ E(n_1,p,q) & ::= & E(n_1 - 1, p+1, q) & \textit{prefix} \\ E(n_1,p,q) & ::= & E(n_1 - 1, p, q+1) & \textit{postfix} \end{array} \tag{170}$$

Since $n_1 > n_2$ then $n_1 - 1 \geq n_2$ and we get from assumption A166 that either:

$$\begin{array}{l} E(n_1 - 1, p, q) \rightarrow^* E(n_2, p+p'', q+q'') \\ E(n_1 - 1, p+1, q) \rightarrow^* E(n_2, p+1+p'', q+q'') \\ E(n_1 - 1, p, q+1) \rightarrow^* E(n_2, p+p'', q+1+q'') \end{array} \tag{171}$$

where p'' and q'' are the number of prefix and postfix operators between n_2 and $n_1 - 1$.

In the first case $p''(q'')$ also is the number of prefix(postfix) operators between n_2 and n_1 since there was an infix operator with precedence n_1 and hence neither a prefix nor a postfix operator. In the second case $p'' + 1(q'')$ is the number of prefix(postfix) operators between n_2 and n_1 since there was a prefix operator with precedence n_1. In the third case $p''(q'' + 1)$ is the number of prefix(postfix) operators between n_2 and n_1 since there was a postfix operator with precedence n_1.

Statement C169 follows from this, and since n_2, p and q were arbitrary we have C167. Statement C164 follows with induction since we have C165 and (A166 $\Longrightarrow$ C167). □

Lemma 24 *If* $\mathsf{Pc}(t)$ *and there is no uncovered prefix or infix operator op in t for which* $\mathsf{P}(op) > n$ *then* $\mathsf{Rw}(t) \leq n$.

Proof: The syntax trees have the following form, where *pin* either denote an infix operator or a prefix operator. The left subtrees are only present in case of infix operators. The tree T is either a tree for a postfix operator or an atomic tree.
The proof is by induction on the length of the chain. First we assume that there is no uncovered prefix or infix operator *op* in t, for which $\mathsf{P}(op) > n$ for an arbitrary $n \geq 0$.

Base:
The length is zero.

1. $t = AE$
   The lemma holds since $\mathsf{Rw}(AE) = 0 \leq n$.

2. $t = t'\ postop$
   The lemma holds since $\mathsf{Rw}(t'\ postop) = 0 \leq n$.

Induction step:

1. $t = lt\ inop\ rt$
   From the definition of Rw we have

   $$\mathsf{Rw}(t) = max(\mathsf{P}(inop), \mathsf{Rw}(rt)) \tag{172}$$

   Since *inop* of course occur uncovered in t, the conditions in the lemma give us that

   $$\mathsf{P}(inop) \leq n \tag{173}$$

   and since rt has a shorter chain of infix and prefix operators than $lt\ inop\ rt$ we get from the induction assumption that:

   $$\mathsf{Rw}(rt) \leq n \tag{174}$$

   If we use 173 and 174 in 172 we get

   $$\mathsf{Rw}(t) \leq n \tag{175}$$

   and we have concluded the induction step for the first case.

2. $t = preop\ t'$
   Analogous to the previous case with $t' = rt$.

This conclude the induction and we have proved the lemma. □

Lemma 25 *If* $\mathsf{Pc}(t)$ *and there are no uncovered postfix or infix operator op in t for which* $\mathsf{P}(op) > n$ *then* $\mathsf{Lw}(t) \leq n$.

Proof: Analogous to the proof of lemma 24. □

Part V

Precedences for context-free grammars

Precedences for Context-free Grammars

Annika Aasa

1 Introduction

The syntax of programming languages is often described by context-free grammars. A problem with these descriptions is that it is often hard to make them both concise, readable and unambiguous. Unambiguous context-free grammars often contain a lot of nonterminals and single productions which make them quite hard to read. Consider for example an ambiguous grammar for expressions in a small functional language and an unambiguous alternative where the usual precedences of the constructions are incorporated:

$$
\begin{array}{lcl}
E & ::= & \texttt{if } E \texttt{ then } E \texttt{ else } E \\
 & | & \texttt{let id=}E \texttt{ in } E \\
 & | & E \texttt{ = } E \\
 & | & E \texttt{ + } E \\
 & | & \texttt{int} \\
 & | & \texttt{id} \\
 & | & \texttt{(} E \texttt{)}
\end{array}
$$

$$
\begin{array}{lcl}
E & ::= & T0 \texttt{ = } T1 \\
 & | & T1 \\
T1 & ::= & T0 \texttt{ + } EP \\
 & | & EP \\
T0 & ::= & T0 \texttt{ + } AE \\
 & | & AE \\
EP & ::= & \texttt{if } E \texttt{ then } E \texttt{ else } E \\
 & | & \texttt{let id=}E \texttt{ in } E \\
 & | & AE \\
AE & ::= & \texttt{int} \\
 & | & \texttt{id} \\
 & | & \texttt{(} E \texttt{)}
\end{array}
$$

The reason for the need of two different nonterminals generating parse trees with `+` as root in the unambiguous alternative is explained in part IV of this thesis. A larger example of a syntax description that in some parts is hard to read for a human is the grammar of CAML [CH90, WAL+90] which contains a lot of nonterminals, the only aim of which is to make the grammar unambiguous.

One way to make an ambiguous grammar unambiguous, without rewriting it completely, is to use precedence and associativity rules[1] for disambiguating. It is however not always so easy to see how these rules disambiguate a grammar. In an old description of SML [Mil84] it was stated that the productions were given in decreasing precedence order but it was not explained or obvious what that meant.

For grammars where precedences are given only to infix operators, everyone agrees on which parse trees that should be regarded as correct. For arbitrary context-free grammars, there is however no generally accepted rule but only rules which are dependent on different parsing techniques. Descriptions of how such precedence rules are used in parsers can be found in Aho and Ullman [AJU75] and Earley [Ear75].

[1] From now on we meen both precedence and associativity rules when we say precedence rules.

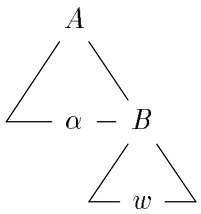
It is unsatisfactory to define the syntax of a language in terms of a specific parsing method. A specification of a language should not involve a method to recognize it, because if the language is defined by one parsing method it could be hard to see if a parser that uses another method is correct. Also, it is a serious disadvantage if understanding of how disambiguation works requires knowledge of how a specific parser works.

The use of precedence rules to resolve ambiguity is common in parsers which are based on LR-techniques [ASU86, Knu65]. In LR-parser generators, for example YACC [Joh75], precedences are used to resolve some shift-reduce conflicts. The user of the parser generator must therefore understand how the parser generator works, at least if she wants to use precedences. The productions given precedences are often not the natural ones. Often a seemingly harmless change in a grammar forces precedence changes since there will be conflicts between other productions. It is also hard to translate LR-precedence-rules to other parsing techniques.

So, although precedences are used in many situations, there is no adequate definition of what it means for a production in a grammar to have higher precedence than another. A parser-independent definition should say which parse trees a grammar generates and not only which sentences, since the structure is important. Given an ambiguous grammar and a set of precedence rules, it should be possible to decide if a parse tree belongs to the language or not. Note that we when dealing with precedences must consider languages as sets of trees and not only as sets of strings as usual. In part IV of this thesis (and also in [Aas91]) there is such a definition for a subclass of context-free grammars. In this part we consider more general grammars.

2 Notation

We assume that the reader is familiar with context-free grammars and we will use ordinary notation conventions. We use letters written in typewriter font for terminals, and capitals together with nt for nonterminals. Greek letters are used for sequences of terminals and nonterminals while u, v, w are used for strings of solely terminals. Grammars are written as sequences of productions and the nonterminal in the left hand side of the first production is assumed to be the start symbol. The notation $\Rightarrow$ is used for a derivation step, $\Rightarrow^+$ for a derivation in one or more steps and we use $\Rightarrow_{lm}$ and $\Rightarrow_{lm}^+$ for leftmost derivations. We will sometimes picture parse trees as follows:

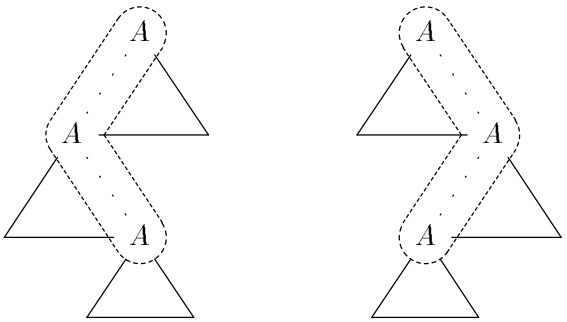


This picture shows a parse tree with a nonterminal A as root. The top half of the tree corresponds to the sentential form $\alpha\ B$. The nonterminal B is root in a sub parse tree for a sentence w.

3 Ambiguity

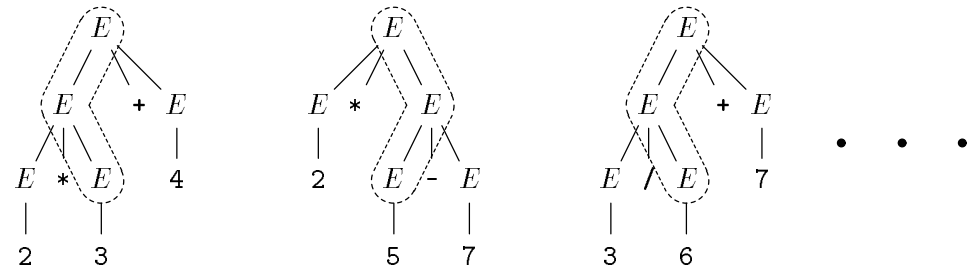
Our goal is to use precedences to resolve ambiguous sentences generated from ambiguous grammars. The first question is if it is possible to resolve all ambiguities with precedences. A grammar is ambiguous if there exist a sentence with more than one parse tree. Unfortunately, the problem of deciding whether an arbitrary context-free grammar is unambiguous is undecidable [AU72, Can62, Flo62], and there are even inherently ambiguous context-free languages [AU72, Par66]. Therefore, the problem of resolving ambiguity with precedences seems to be very hard, if not impossible, to solve in general. We therefore restrict our goal to resolve one kind of ambiguity.

To resolve many kinds of ambiguity there is no other solution than completely rewriting the grammar. However, it is possible to isolate a pattern in the parse trees which we think is suitable to resolve with precedences. This ambiguity is a usual form of ambiguity in grammars for programming languages, and it includes the programming language constructs for which we normally use precedences. The pattern occurs when a parse tree is, or contains a subtree, of one of the following forms:



A more precise definition is given later. We will call such a parse tree an *R-ambiguous* parse tree, and this kind of ambiguity an *R-ambiguity*[2]. Our goal is to resolve R-ambiguities with precedences. Note that the ambiguous grammar for arithmetic expressions generates several R-ambiguous parse trees:

E ::= `int`
| E `*` E
| E `/` E
| E `+` E
| E `-` E



[2]The *R* stands for recursive. We will later see why.

In this case we are used to use precedences to specify which of the R-ambiguous parse trees we want to consider as the correct ones. We usually want to consider the first and third tree as correct and the second as incorrect. To achieve this we must give the productions E ::= E`+`E and E ::= E`-`E higher precedence than the productions E ::= E`*`E and E ::= E`/`E. We use the a bit unusual convention that, for example `+` must be given *higher* precedence than `*` if we would like the usual binding of the arithmetic operators.

A more complicated example is a subset of the grammar for SML as defined in [Mil84] (in later descriptions [MTH90] the grammar is slightly changed):

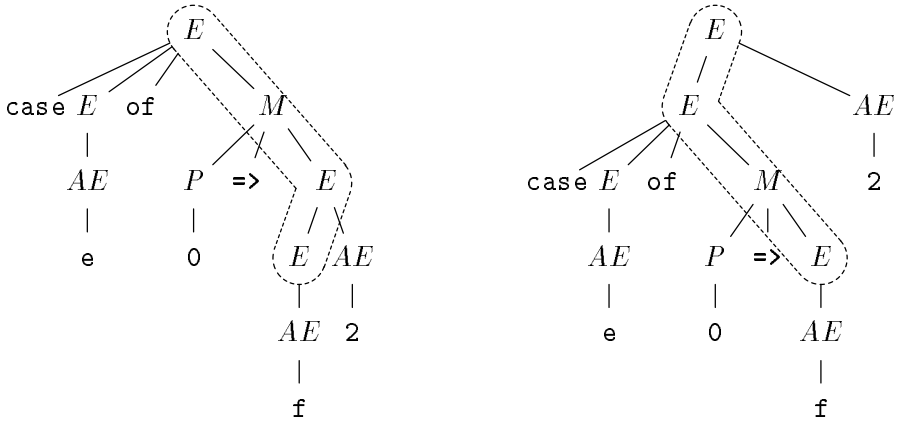
E ::= AE
| E AE
| `case` E `of` M

AE ::= `int`
| `id`

M ::= P `=>` E

P ::= `int`
| `id`

This grammar generates, for example, two R-ambiguous parse trees for the sentence `case e of 0 => f 2`:



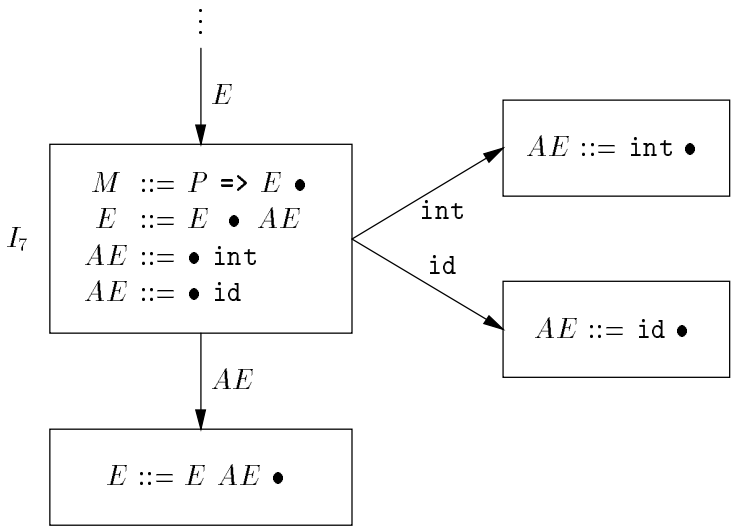
To resolve this ambiguity, the most intuitive way would be to give precedences to the productions:

E ::= E AE
E ::= `case` E `of` M

This is, however, not how the ambiguity is resolved in an LR-parser [ASU86, AJU75, Ear75], where precedences must be given to the productions:

M ::= P `=>` E
AE ::= `int`
AE ::= `id`

To see why, it is often necessary to look at the collection of LR(0) item sets. In the picture below we show some of them:

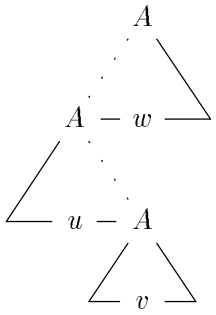


In item set I_7 there is a shift-reduce conflict on input `int` between the two productions M ::= P `=>` E and AE ::= `int` and on input `id` there is a shift-reduce conflict between the two productions M ::= P `=>` E and AE ::= `id`. These conflicts are resolved by giving precedences to the involved productions. This way of resolving ambiguity with precedences is unnatural for most people and it really requires knowledge of how an LR-parser works. The reason why most people want to give precedences to the productions

E ::= E AE
E ::= `case` E `of` M

to resolve the ambiguity, is that the nonterminal E is both left- and right recursive. A nonterminal A is *left recursive* (*right recursive*) if $A \Rightarrow^+ A\alpha$ ($A \Rightarrow^+ \alpha A$).

It is easy to see that if a grammar generates an R-ambiguous parse tree there is at least one *useful*[3] nonterminal which is both left- and right recursive in the grammar. Suppose that a grammar generates the following R-ambiguous (sub)parse tree:



[3] A nonterminal is useful if it appears in a derivation of a sentence. Description of an algorithm that removes useless nonterminals from a grammar can be found in a book by Grune and Jacobs [GJ90].

From the parse tree, we can see that there must be derivations $A \Rightarrow^+ Aw$ and $A \Rightarrow^+ uA$. Thus A is both left- and right (blr) recursive.

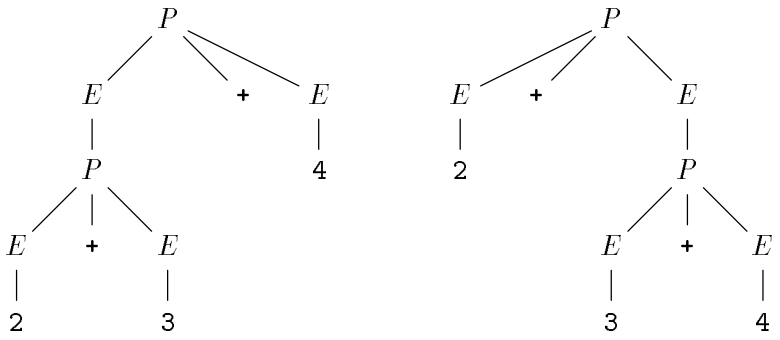
Definition 1 *A **blr-grammar** have at least one useful nonterminal which is both left- and right recursive.*

All blr-grammars are ambiguous. The proof is straightforward. In a blr-grammar there must be derivations $A \Rightarrow^+_{lm} wA$ and $A \Rightarrow^+_{lm} A\beta$. Then the sentential form $wA\beta$ has two leftmost derivations: $A \Rightarrow^+_{lm} wA \Rightarrow^+_{lm} wA\beta$ and $A \Rightarrow^+_{lm} A\beta \Rightarrow^+_{lm} wA\beta$. Thus, the grammar is ambiguous.

Unfortunately, a blr-grammar does not always generate R-ambiguous parse-trees, even if that is the most common case. One example of a blr-grammar for which the ambiguous sentences not always have R-ambiguous parse trees is given below. Note that the nonterminal P is the startsymbol, and that both nonterminals are both left- and right recursive:

$$\begin{array}{lcl} P & ::= & E \texttt{ + } E \\ E & ::= & \texttt{int} \\ & | & P \end{array}$$

This (strange) grammar is ambiguous and the ambiguous sentences sometimes have R-ambiguous parse trees and sometimes not. The sentence `2+3+4` is for example ambiguous but its two different parse trees are not R-ambiguous:

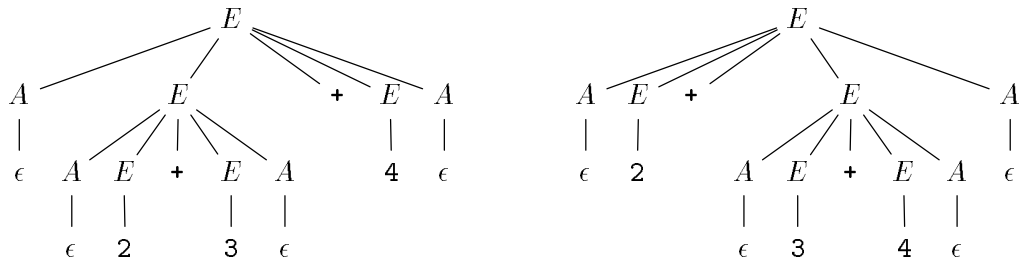


Two non-R-ambiguous parse trees for the sentence `2+3+4`.

Another example is the grammar:

$$\begin{array}{lcl} E & ::= & A\ E \texttt{ + } E\ A \\ & | & \texttt{int} \\ A & ::= & \epsilon \end{array}$$

It, for example, generates two different parse trees for the sentence `2+3+4`, which are not R-ambiguous:



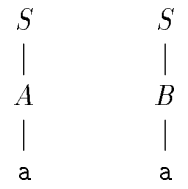
Two non-R-ambiguous parse trees for the sentence `2+3+4`.

In this case, the nonterminal E is both left- and right recursive but the grammar does not generate R-ambiguous parse trees at all.

Also, a sentence generated from a blr-grammar can of course be ambiguous in a way that has nothing to do with the left- and right recursiveness. The following blr-grammar generates the sentence `a`:

$$
\begin{array}{lcl}
S & ::= & A \\
 & | & B \\
A & ::= & A \texttt{ \# } A \\
 & | & \texttt{a} \\
B & ::= & \texttt{a}
\end{array}
$$

The sentence `a` is ambiguous but that has nothing to do with the fact that the nonterminal A is both left- and right recursive:



Two non-R-ambiguous parse trees for the sentence `a`.

For these reasons, we will not resolve all ambiguity in blr-grammars with precedences. However, most blr-grammars generate R-ambiguous parse trees and R-ambiguity has a close correspondence to left- and right recursive nonterminals. Therefore, we propose, in order to resolve R-ambiguity, that precedences must be given to the nonterminals which are both left- and right recursive.

The problem of deciding whether a nonterminal is both left- and right recursive is decidable and is similar to the problem of deciding whether a nonterminal is *accessible*[4] [Bac79]. In practical cases, it is quite easy to see which nonterminals that are both left- and right recursive. Therefore, it should not be any problems for a grammar writer to augment a grammar with precedences.

In the following we will more formally define what we mean by precedences and also define which R-ambiguous parse trees that will be considered as the correct ones. To do this we need to introduce some definitions.

[4] A nonterminal, A, is *accessible* if there is a derivation $S \Rightarrow^* \alpha A \beta$ where S is the startsymbol.

Definition 2 *The **left edge above** a node A in a parse tree is the sequence of nodes given by:*

1. *If A does not have any left siblings then the father of A is first element in the left edge above A.*
2. *If a node B belongs to the left edge above A then the left edge above B is the last part of the left edge above A.*

The order of the nodes in the sequence is that a father follows its child.

*The **right edge above** a node is defined analogously.*

Definition 3 *The **left edge below** a node A in a parse tree is the sequence of nodes given by:*

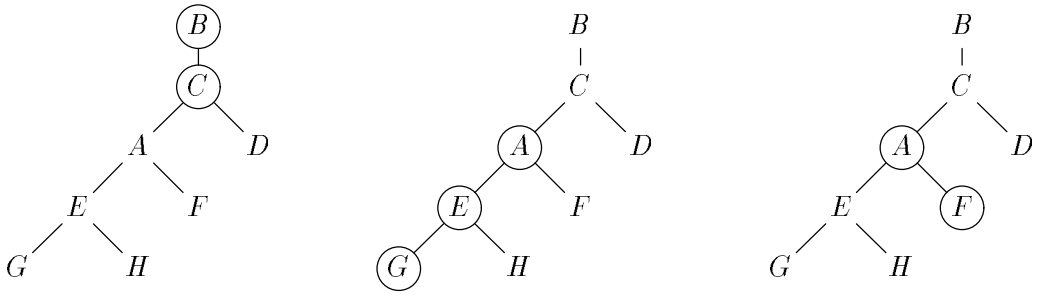
1. *The node A is the first element in the left edge below A.*
2. *If a node B belongs to the left edge below A then the left edge below the leftmost child of B is the last part of the left edge below A.*

The order of the nodes in the sequence is that a child follows its father.

*The **right edge below** a node is defined analogously.*

Note that a node belongs to the left edge below itself but not to the left edge above itself.

Example:



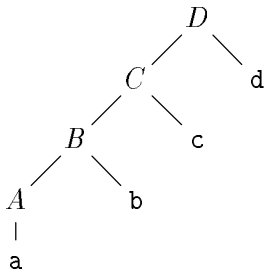
left edge above $A = [C, B]$ left edge below $A = [A, E, G]$ right edge below $A = [A, F]$

No nodes belong to the right edge above A since A has a right sibling. □

In the following, we hope that the distinction between a node and the nonterminal in a node will be clear from the context. With the notation A, A_1, A_2, A_3 we mean different nodes but they all correspond to the same nonterminal.

Definition 4 *A production is said to be* ***used on a left edge above/below a node*** *if it is used in the parse tree with its left hand side nonterminal on the left edge above/below the node.*

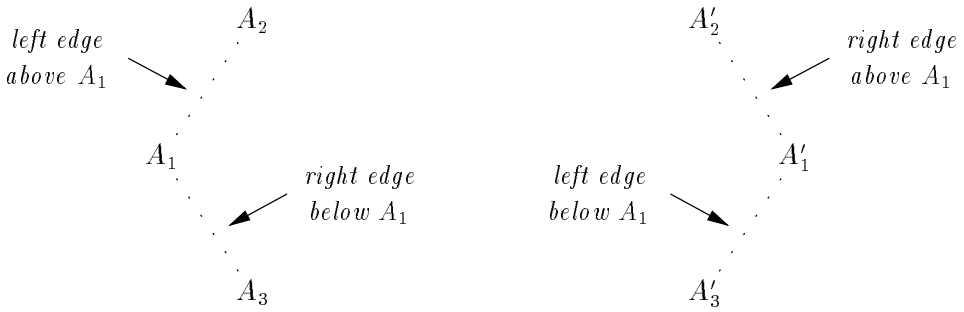
Example: The production $C ::= B$ c is used on the left edge above A:



□

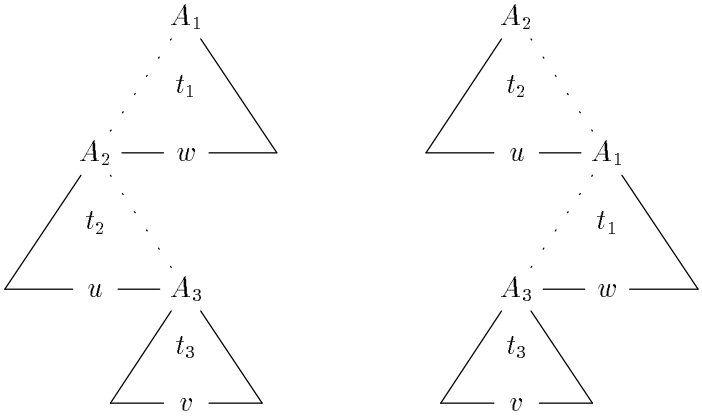
We can now more formally define an R-ambiguous parse tree.

Definition 5 *A parse tree is* ***R-ambiguous*** *if it contains some node A_1 such that there is a node A_2 on the left(right) edge above it and a node A_3 on the right(left) edge below A_1.*

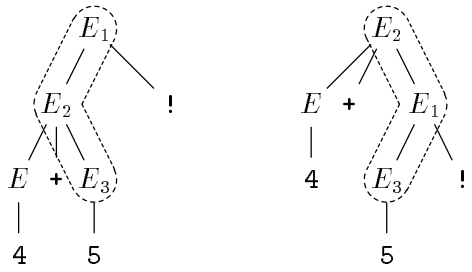


Note that the chosen node A_2 is not necessarily the first with nonterminal A on the left(right) edge above node A_1. There can also be more nodes for A between node A_1 and the chosen node A_3 on the right(left) edge below A_1. Which node we choose does not have any significance when deciding if a parse tree is R-ambiguous, but it matters in the following definition.

Definition 6 *An* ***R-ambiguity-pair*** *is two R-ambiguous parse trees for the same sentence such that the nodes chosen in definition 5 and their (not complete) subtrees, are swapped as shown in the picture:*



Example: An R-ambiguity-pair for the sentence `4+5!`:



□

Every R-ambiguous parse tree for which their exists parse trees for u and v is a parse tree in an R-ambiguity-pair. If the parse trees for u and v are nonexistent, which is only possible if the grammar is cyclic, the same tree is achieved when the nodes are swapped. We use the notion of R-ambiguity-pair in later sections when we discuss how precedences must be given to the productions in a grammar in order to resolve R-ambiguity and when we motivate that the precedence definition is reasonable.

4 Precedence Grammars

Informally, a precedence grammar is a grammar together with precedence rules and in the next section we will define which language (of trees) a precedence grammar generates. First, we more precisely define a precedence grammar.

As mentioned before, it is quite easy to recognize a blr-grammar which almost certainly generates R-ambiguous parse trees. A grammar writer usually have an idea on which structure she wants on the parse trees. Given examples of different parse trees for ambiguous sentences, it is not too hard to translate these ideas into rules of the form that one production must have higher precedence than another, and that one production must be left associative. With these informal rules in mind, a precedence grammar that generates the desired language, can usually easily be constructed.

Definition 7 *In a* ***precedence grammar****, the productions for each both left- and right recursive nonterminal are given in such a way that alternatives that are neither left- nor right recursive come first. The other productions are either preceded by a '<' sign or a '=' sign. The first recursive production is preceded by a '<' sign. A production preceded by a '<' sign has higher precedence than the previous production. A production preceded by a '=' sign has the same precedence than the previous production.*

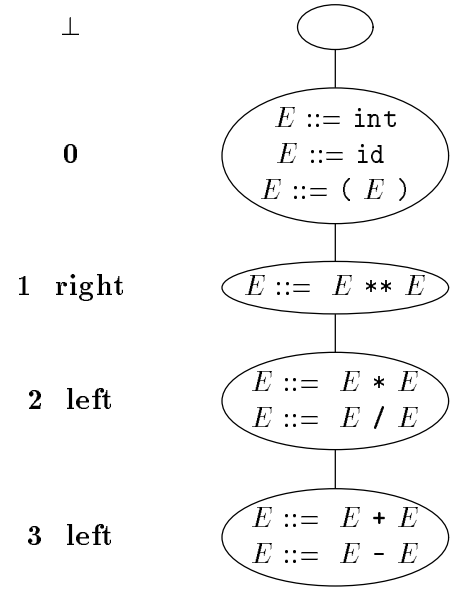
A ***precedence group*** *is a set of productions having the same precedence. If there are both left- and right recursive productions in a precedence group there must be a L, R or a N after the '<' sign indicating the associativity of the precedence group. The letter L stands for left associative, R for right associative, and N for nonassociative.*

Example: A precedence grammar for arithmetic expressions.

E	::=	`int`
	|	`id`
	|	`(` E `)`
$< R$	|	E `**` E
$< L$	|	E `*` E
$=$	|	E `/` E
$< L$	|	E `+` E
$=$	|	E `-` E

□

The precedence groups in a precedence grammar with the precedence relation form a total order. We augment the order with a least element. The total order for the precedence grammar in example 4 can be pictured as follows:



In the next section, when we define which language a precedence grammar defines, we use two operations on the total order. One operation is to take the maximum of two precedence groups and the other is to determine if one group is less than the other. In the description we will code the total order as natural numbers plus $\perp$ and then use the ordinary operations on them, *max* and $<$. In the picture above we have annotated the order with the numbers that will be used in the coding of the order. We will also when appropriate annotate precedence grammars with the natural numbers that code the precedence orders. There is one total order for each nonterminal which is both left- and right recursive. In this example there is only one such nonterminal, E.

Note that the precedence grammars introduced in part IV of this thesis with some minor changes is a special case of the precedence grammars considered here.

5 Definition of Precedence

One obvious question to ask is which language we define with a precedence grammar. According to the precedence rules some parse trees will be illegal and some legal. We will call the legal parse trees *precedence correct* and they will be defined in terms of an attribute grammar. The definition has a lot in common with the definition of precedences for distfix operators given in part IV of this thesis and also in [Aas91]. It is, however, more complicated since we define precedence for more general grammars. In part IV we only considered grammars with one nonterminal.

An attribute grammar [Knu68a, Knu68b] is a context-free grammar in which each node in the parse trees is augmented with a set of attributes and a condition. A parse tree is correct if all conditions are true. The values of the attributes are specified by evaluation rules given together with the productions. Attribute grammars are often used to specify syntax directed translations. Then the conditions are not used. Here, we will use attribute grammars to specify a language and the conditions play a central role.

5.1 Informal description

The main idea in our precedence definition is to define the precedence checks by the conditions in the attribute grammar and use the attributes to let the precedence information be propagated up and down in the parse tree.

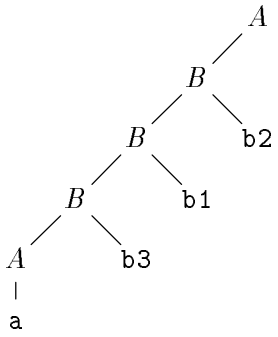
5.1.1 Synthesized attributes

All nodes in the parse trees are given weights, *left weights* and *right weights*. The values of the weights depend on the precedences of the used productions on the left- and right edges below the node. Two synthesized attributes are used for the weights. The left weight for a nonterminal, nt, in a node is the maximum of the precedences of all productions for nt on the left edge below the node, where the precedence for the last production for nt is replaced by zero. The reason to not include the precedence of the last production for nt is that it is not used left recursively.

Example: Consider the precedence grammar:

A	::=	`a`	
	|	B	
B	::=	`b0`	0
$<$	|	B `b1`	1
$<$	|	B `b2`	2
$<$	|	A `b3`	3
$<$	|	`b4` B	4

It generates the parse tree:

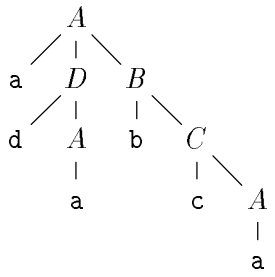


The left weight for the nonterminal B in the root node is 2 despite of the appearance of the production B ::= A `b3` because that production is not used left recursively. □

5.1.2 Inherited attributes

Two inherited attributes are used to let the precedence of a production "flow down" to the node where it is most suitable to define the precedence check. When the precedence of a production A ::= α is inherited it is divided into a *left precedence* and a *right precedence*. The left precedence is propagated down along the left edge below A and the right precedence is propagated along the right edge below A until the father of the next appearance of A. The value of the left precedence for a nonterminal in a node is the precedence for the first used production of this nonterminal on the left edge above the node. If a nonterminal does not appear on the left edge above a node then the value of the left precedence for that nonterminal in the node is $\perp$. The value of the right precedence is analogous.

Example: Consider the (non R-ambiguous) parse tree:



If the precedence of the production $A ::= \mathtt{a}\ D\ B$ is 4 then the right precedence for A in nodes B and C is 4, and the right precedence for A in node D is $\perp$. No node has an inherited left precedence for A. □

Since there can be more than one nonterminal which is both left- and right recursive, all attributes are functions from nonterminals to precedences. The value of the left precedence of a nonterminal in a node indicates how heavy the right weight for the leftmost subtree is allowed to be for the same nonterminal, and vice versa for the right precedence.

5.2 Formal definition

Let us now give a more formal description of the evaluation rules and the conditions in the attribute grammar. In the description we use pattern matching over different forms of productions and we use the following notation.

$\mathsf{P}(A ::= \alpha)$	denote the precedence of the production $A ::= \alpha$.
$f\left[\begin{array}{l} x_1 \mapsto v_1 \\ x_2 \mapsto v_2 \end{array}\right]$	denote the function f updated with the value v_1 for the argument x_1 and then updated with the value v_2 for the argument x_2.

As mentioned in section 4 we use natural numbers plus $\perp$ to represent the precedence order and we now define the following types.

PrecAss	=	Precedence × Associativity
Precedence	=	Nat
Associativity	=	Set of {left, right}

A pattern is typically on the form $A ::= \alpha B$. All productions having a nonterminal as the rightmost symbol on the right hand side matches this pattern, for example $E ::= E\mathtt{+}E$, $E ::= \mathtt{case}\ E\ \mathtt{of}\ M$, and $C ::= D$. All productions in the grammar are matched against all patterns. Notice that a production can match more than one pattern.

5.2.1 Inherited attributes

For precedence checks of indirectly left- or right recursive[5] nonterminals we let the precedences be inherited downwards in the parse tree. We use two inherited attributes, the left precedences, lps, and the right precedences, rps, for this.

[5] A nonterminal, A, is indirectly left recursive if $A \Rightarrow^{+} A\alpha$ in more than one step.

lps: Nonterminal $\rightarrow$ PrecAss $\cup \{\bot\}$
rps: Nonterminal $\rightarrow$ PrecAss $\cup \{\bot\}$

The evaluation rules for these attributes are as follows.

Left precedences

production	*attribute evaluation rules*
$B ::= A\alpha$	$\begin{cases} A.\mathsf{lps} := B.\mathsf{lps} \left[\begin{array}{l} B \mapsto \mathsf{P}(B ::= A\alpha) \\ A \mapsto \bot \end{array} \right] \\ nt.\mathsf{lps} := \lambda x.\bot \quad \text{for all } nt \in \alpha \end{cases}$
$B ::= \mathsf{a}\alpha$	$nt.\mathsf{lps} := \lambda x.\bot \quad \text{for all } nt \in \alpha$

Right precedences

production	*attribute evaluation rules*
$B ::= \alpha A$	$\begin{cases} A.\mathsf{rps} := B.\mathsf{rps} \left[\begin{array}{l} B \mapsto \mathsf{P}(B ::= \alpha A) \\ A \mapsto \bot \end{array} \right] \\ nt.\mathsf{rps} := \lambda x.\bot \quad \text{for all } nt \in \alpha \end{cases}$
$B ::= \alpha\mathsf{a}$	$nt.\mathsf{rps} := \lambda x.\bot \quad \text{for all } nt \in \alpha$

5.2.2 Synthesized attributes

Each node has weights for the nonterminals that belongs to the left or right edge below it. The left(right) weight for a nonterminal, A, in a node is the maximum of the precedences of all productions for A that are used left(right) recursively on the left(right) edge below the node. The left weights, Lws, and right weights, Rws, are two synthesized attributes with the following types.

Lws: Nonterminal $\rightarrow$ Precedence $\cup \{\bot\}$
Rws: Nonterminal $\rightarrow$ Precedence $\cup \{\bot\}$

The attribute evaluation rules for them are as follows.

Left weights

production	*attribute evaluation rules*
$B ::= \mathsf{a}\alpha$	$B.\mathsf{Lws} := (\lambda x.\bot)[B \mapsto 0]$
$B ::= A\alpha$	$B.\mathsf{Lws} := \begin{cases} A.\mathsf{Lws}[B \mapsto max\ \mathsf{lw}\ \mathsf{p}] & \textit{if } \mathsf{lw} \neq \bot \\ A.\mathsf{Lws}[B \mapsto 0] & \textit{otherwise} \end{cases}$ *where* $\mathsf{lw} = (A.\mathsf{Lws})\ B$ $\mathsf{p} = \textit{fst}(\mathsf{P}(B ::= A\alpha))$

Right weights

production	*attribute evaluation rules*
$B ::= \alpha\mathtt{a}$	$B.\mathsf{Rws} := (\lambda x.\bot)[B \mapsto 0]$
$B ::= \alpha A$	$B.\mathsf{Rws} := \begin{cases} A.\mathsf{Rws}[B \mapsto max\ \mathsf{rw}\ \mathsf{p}] & \textit{if } \mathsf{rw} \neq \bot \\ A.\mathsf{Rws}[B \mapsto 0] & \textit{otherwise} \end{cases}$ $\textit{where}\quad \mathsf{rw} = (A.\mathsf{Rws})\ B$ $\mathsf{p} = \mathit{fst}(\mathsf{P}(B ::= \alpha A))$

5.2.3 Conditions

The precedence checks are defined by the conditions in the attribute grammar.

production	*condition*
$A ::= \mathtt{a}\alpha\mathtt{b}$	true
$A ::= \mathtt{a}$	true
$A ::= \alpha B$	$\begin{cases} \text{true} & \textit{if}(\underbrace{A.\mathsf{rps}[A \mapsto \mathsf{P}(A ::= \alpha B)])}_{\mathsf{rp}}\ B = \bot \\ \mathsf{p} > \mathsf{lw} \vee (\mathsf{p} = \mathsf{lw} \wedge \mathsf{right} \in \mathsf{ass}) & \textit{otherwise} \\ \quad \textit{where}\quad \mathsf{lw} = (B.\mathsf{Lws})\ B \\ \quad\quad \mathsf{p} = \mathit{fst}\ \mathsf{rp} \\ \quad\quad \mathsf{ass} = \mathit{snd}\ \mathsf{rp} \end{cases}$
$A ::= B\alpha$	$\begin{cases} \text{true} & \textit{if}(\underbrace{A.\mathsf{lps}[A \mapsto \mathsf{P}(A ::= B\alpha)])}_{\mathsf{lp}}\ B = \bot \\ \mathsf{p} > \mathsf{rw} \vee (\mathsf{p} = \mathsf{rw} \wedge \mathsf{left} \in \mathsf{ass}) & \textit{otherwise} \\ \quad \textit{where}\quad \mathsf{rw} = (B.\mathsf{Rws})\ B \\ \quad\quad \mathsf{p} = \mathit{fst}\ \mathsf{lp} \\ \quad\quad \mathsf{ass} = \mathit{snd}\ \mathsf{lp} \end{cases}$

Notice that a production can match both the two last patterns, for example $E ::= E\mathtt{+}E$. In that case both conditions must be true in order for the whole condition to be true.

5.3 Example

In the grammar below, the nonterminal E is both left- and right recursive and in order to resolve the R-ambiguity, we give precedences to all productions for E which is either left- or right recursive. In this grammar we have to define a precedence relation between $E ::= E\ E$ and $E ::= \mathtt{case}\ E\ \mathtt{of}\ M$. We indicate this in the grammar with a '$<$' sign before the productions. Let us give higher precedence to the case-production since we, for example, want the sentence `case e of 0 => f 2` to be interpreted as `case e of 0 => (f 2)`. We

also indicate with an L that the production $E ::= E\ E$ is left associative.

E	::=	`int`	0
	|	`id`	0
$< L$	|	$E\ E$	1
$<$	|	`case` E `of` M	2
M	::=	P `=>` E	
P	::=	`int`	
	|	`id`	

With our precedence definition we see that the meaning of this precedence grammar grammar is the following attribute grammar. We have simplified the attribute grammar a bit by using the knowledge that it is only productions for E that have precedences, and that the nonterminal P can not derive an E.

$E ::=$ `int` | `id`

$E.\mathsf{Lws} := (\lambda x.\bot)[E \mapsto 0]$
$E.\mathsf{Rws} := (\lambda x.\bot)[E \mapsto 0]$
Condition: true

$E ::= E'\ E''$

$E'.\mathsf{lps} := E.\mathsf{lps}[E \mapsto \bot]$
$E''.\mathsf{lps} := \lambda x.\bot$
$E'.\mathsf{rps} := \lambda x.\bot$
$E''.\mathsf{rps} := E.\mathsf{rps}[E \mapsto \bot]$
$E.\mathsf{Lws} := E'.\mathsf{Lws}[E \mapsto max\ (E'.\mathsf{Lws}\ E)\ 1]$
$E.\mathsf{Rws} := E''.\mathsf{Rws}[E \mapsto max\ (E''.\mathsf{Rws}\ E)\ 1]$
Condition: $1 \geq E'.\mathsf{Rws}\ E\ \wedge\ 1 > E''.\mathsf{Lws}\ E$

$E ::=$ `case` E' `of` M

$E'.\mathsf{lps} := \lambda x.\bot$
$M.\mathsf{lps} := \lambda x.\bot$
$E'.\mathsf{rps} := \lambda x.\bot$
$M.\mathsf{rps} := E.\mathsf{rps}[E \mapsto (2, \{\mathsf{left}, \mathsf{right}\})]$
$E.\mathsf{Lws} := (\lambda x.\bot)[E \mapsto 0]$
$E.\mathsf{Rws} := M.\mathsf{Rws}[E \mapsto max\ (M.\mathsf{Rws}\ E)\ 2]$
Condition: true

$M ::= P$ `=>` E

$E.\mathsf{lps} := \lambda x.\bot$
$E.\mathsf{rps} := M.\mathsf{rps}[E \mapsto \bot]$
$M.\mathsf{Rws} := E.\mathsf{Rws}[M \mapsto 0]$
Condition: $fst\ (M.\mathsf{rps}\ E) \geq E.\mathsf{Lws}\ E$

$P ::=$ `int` | `id`

$P.\mathsf{Lws} := (\lambda x.\bot)[P \mapsto 0]$
$P.\mathsf{Rws} := (\lambda x.\bot)[P \mapsto 0]$
Condition: true

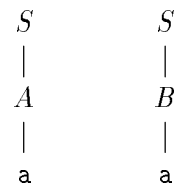
6 Correctness

The reason to add precedences to a grammar is to exclude some parse trees from the generated language. Which parse trees that are excluded are given by the definition of precedence correct parse trees given in section 5. The definition can be said to be correct if it is the undesired parse trees that are excluded. Since our goal was to resolve R-ambiguity, one requirement must be that only R-ambiguous parse trees are thrown away. This is formulated in theorem 8. Another obvious requirement is that all parse trees generated from a grammar without precedences must be correct. This is formulated in theorem 9.

Theorem 10 says that not all parse trees generated from a precedence grammar can be precedence correct. It states that a precedence grammar does not generate any R-ambiguity-pairs. The theorem does, however, not imply that a precedence grammar is unambiguous. As mentioned before, we have restricted ourselves to resolve R-ambiguity and a precedence grammar is not always unambiguous since it may contain other ambiguities than R-ambiguities. A simple example is the precedence grammar:

$$
\begin{array}{lcl}
S & ::= & A \\
 & | & B \\
A & ::= & \texttt{a} \\
B & ::= & \texttt{a}
\end{array}
$$

The sentence `a` has two different parse trees:

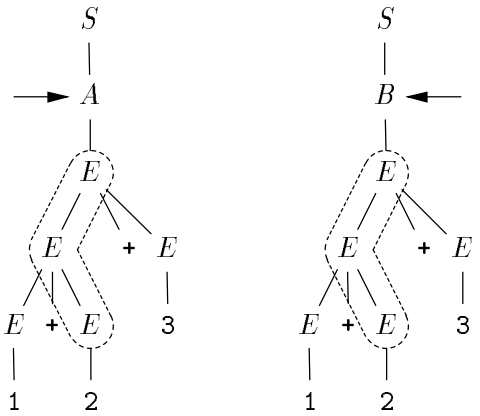


Two non-R-ambiguous parse trees for the sentence `a`.

It is neither true that a sentence generated from a precedence grammar has at most one R-ambiguous parse tree. The same R-ambiguous parse tree can occur as a subtree in two different parse tree as a result of some other ambiguity. The following precedence grammar contains both an R-ambiguity, which we have resolved by precedences, as well as another kind of ambiguity:

$$
\begin{array}{lcl}
S & ::= & A \\
 & | & B \\
A & ::= & E \\
B & ::= & E \\
E & ::= & \texttt{int} \\
> L & | & E \texttt{ + } E
\end{array}
$$

The sentence `1+2+3` has two different parse trees, but this ambiguity is not really an R-ambiguity even if the parse trees are R-ambiguous:

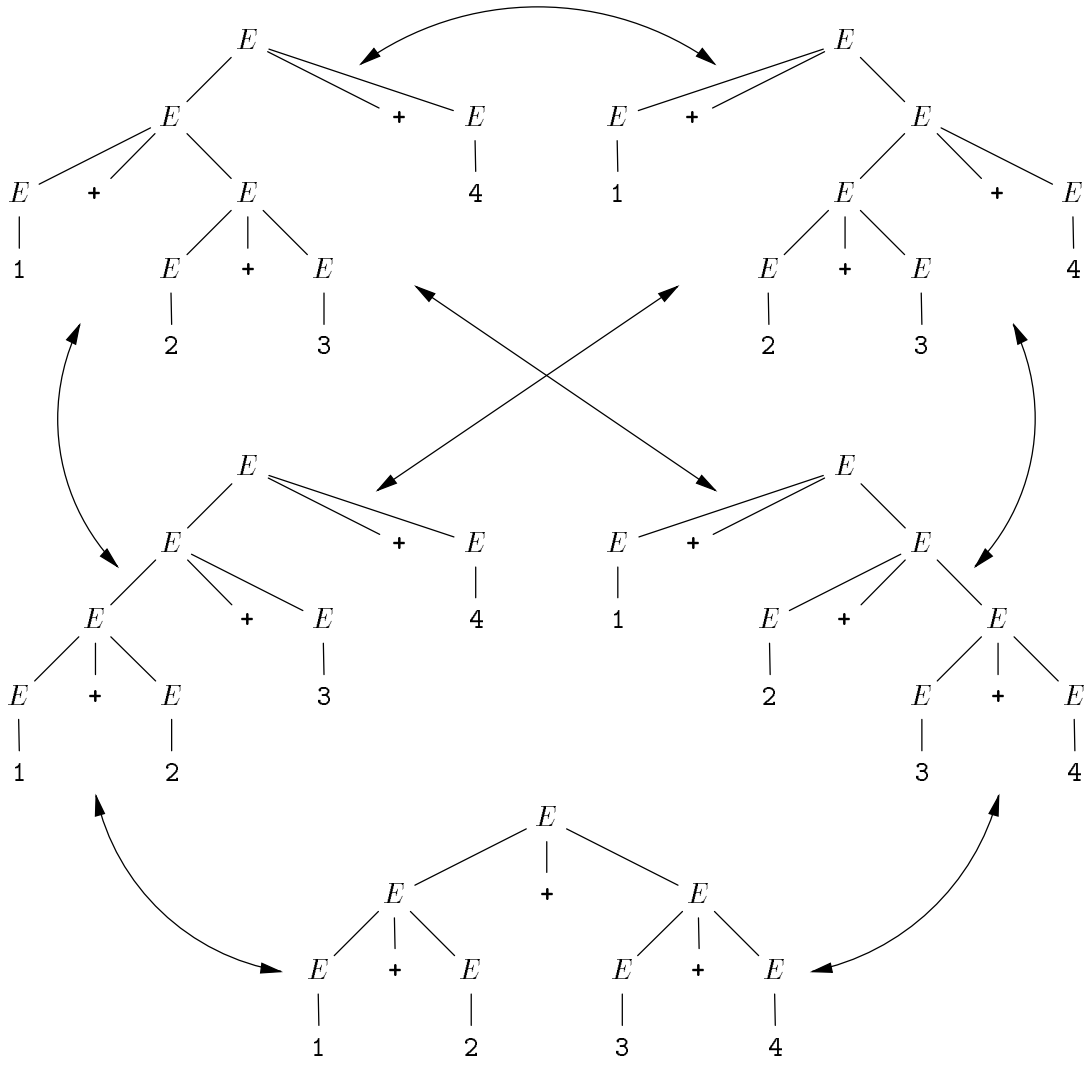


Note that without precedences, the sentence `1+2+3` would have four different parse trees. This example shows that it is not true that a sentence generated from a precedence grammar has at most one R-ambiguous parse tree.

However, theorem 10, that a precedence grammar does not generate any R-ambiguity-pair, is a bit too weak. The following precedence grammar is for example unambiguous, but that does not follow from theorem 10.

$$\begin{array}{lll} E & ::= & \texttt{int} \\ > L & | & E \texttt{ + } E \end{array}$$

The sentence `1+2+3+4` has five different parse trees generated from the grammar without precedences but only one is precedence correct. The R-ambiguity-pairs are indicated by arrows. As seen form the picture, none of the trees form an R-ambiguity-pair with every one of the others. Thus, theorem 10 does not imply that at most one of the trees is precedence correct. Still, the theorem give some indication that the definition is reasonable.

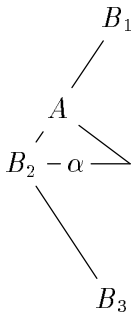


Some lemmas are used in the proofs. These are formulated and proved in appendix A.

Theorem 8 *A precedence incorrect parse tree is R-ambiguous.*

Proof: Suppose that t is a precedence incorrect parse tree. We will show that t is R-ambiguous.

Since t is precedence incorrect, there must be some node in t where the precedence condition is false. The precedence condition rules give that it must either be a node with a nonterminal as leftmost or rightmost child. These two cases are symmetric so let us assume that the condition fails in a node A with a leftmost nonterminal child B, because of the precedence condition rule for the production $A ::= B\alpha$. We will show that there is a nonterminal B on the left edge above B_2 and a nonterminal B on the right edge below B_2 in a node $\neq B_2$ that is, t is R-ambiguous.



If A is the same nonterminal as B then clearly B appears on the left edge above B_2. If A is not the same nonterminal as B then A must have inherited left precedences for B because otherwise the precedence condition is true. Thus B appears on the left edge above B_2 as follows from lemma 13.

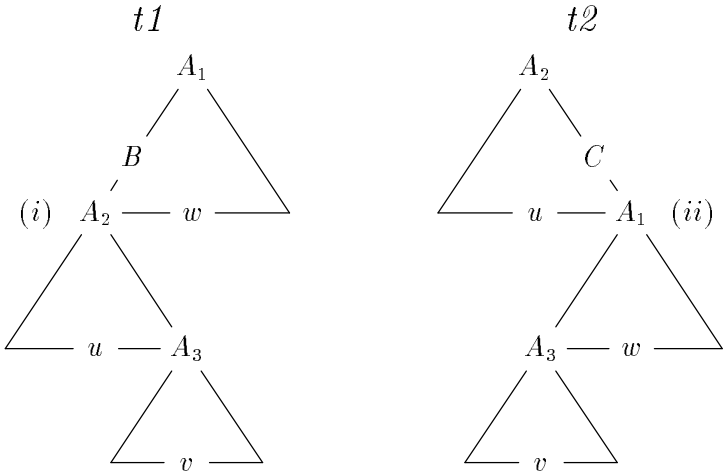
Since the precedence condition is false in node A, lemma 11 gives that the right weight for B in node B_2 is greater than zero and thus B appears on the right edge below B_2 in a node $\neq B_2$ as follows from lemma 15. □

Theorem 9 *A parse tree generated from a grammar in which no precedences are given to the productions is always precedence correct.*

Proof: Follows immediately from lemma 20, which says that if the productions for a nonterminal B is not given precedences then a subtree for B is accepted everywhere. □

Theorem 10 *A precedence grammar does not generate any R-ambiguity-pairs.*

Proof: The theorem says that at most one parse tree in an R-ambiguity-pair is precedence correct. Take an R-ambiguity-pair with parse trees *t1* and *t2*:



We assume that *t1* is precedence correct and show that *t2* is precedence incorrect. If we instead assume that it is *t2* that is precedence correct we get an analogous proof.

Since *t1* is precedence correct, the condition in all nodes is true, and specially is the condition in the first node, B, on the left edge above node (i) true.

Let p1 be the precedence used in the condition in node B. Lemma 16 gives that p1 is the precedence of the first used production for A on the left edge above node (i). We call this production *prod1*.

Let rw be the right weight for A in node (i). Lemma 18 gives that rw is the maximum of the precedences of all used productions for A on the right edge below node (i) where the precedence of the last production for A is replaced by zero.

We will show that the condition in the first node, C, on the right edge above node (ii) in parse tree *t2* is false.

The production *prod1* is used in parse tree *t2* on the left edge below node (ii), and it is not the last one. From that and lemma 19 it follows that the left weight for A, lw, in node (ii) cannot be less than p1.

$$\mathsf{lw} \geq \mathsf{p1} \tag{1}$$

Let p2 be the precedence that will be used in the condition in node C. Lemma 17 gives that p2 is the precedence of the first used production for A on the right edge above node (ii). We call this production *prod2*.

Since the used productions on the right edge above node (ii) in parse tree *t2* is a subset of the used productions on the right edge below node (i) in parse tree *t1*, it follows that the maximum of p2 is rw.

$$\mathsf{p2} \leq \mathsf{rw} \tag{2}$$

We know that *prod1* is left recursive since it it used on the left edge above node (i) and that *prod2* is right recursive since it is used on the right edge above node (ii). From that it follows that if p1 and p2 is equal then they must have the same associativity restriction.

$$\mathsf{p1} = \mathsf{p2} \Longrightarrow \mathsf{ass}_{\mathsf{p1}} = \mathsf{ass}_{\mathsf{p2}} \neq \{\mathsf{left}, \mathsf{right}\} \tag{3}$$

Since the condition in node B is true and $\mathsf{p1} \neq \bot$ the following is true.

$$\mathsf{p1} > \mathsf{rw} \vee (\mathsf{p1} = \mathsf{rw} \wedge \mathsf{left} \in \mathsf{ass}_{\mathsf{p1}}) \tag{4}$$

We will now show that the condition in node C is false. Since $\mathsf{p2} \neq \bot$ we must show that

$$\mathsf{p2} > \mathsf{lw} \vee (\mathsf{p2} = \mathsf{lw} \wedge \mathsf{right} \in \mathsf{ass}_{\mathsf{p2}}) \tag{C5}$$

not holds, that is

$$\mathsf{p2} < \mathsf{lw} \vee (\mathsf{p2} = \mathsf{lw} \wedge \mathsf{right} \notin \mathsf{ass}_{\mathsf{p2}}) \tag{C6}$$

From 4 it follows

$$\mathsf{p1} \geq \mathsf{rw} \tag{7}$$

Combining 1 and 2 with 7 gives

$$\mathsf{p2} \leq \mathsf{rw} \leq \mathsf{p1} \leq \mathsf{lw} \tag{8}$$

and thus

$$\mathsf{p2} \leq \mathsf{lw} \tag{9}$$

We split 9 into two cases, $\mathsf{p2} < \mathsf{lw}$ and $\mathsf{p2} = \mathsf{lw}$. If

$$\mathsf{p2} < \mathsf{lw} \tag{A10}$$

then clearly C6 is true. If instead

$$\mathsf{p2} = \mathsf{lw} \tag{A11}$$

then it follows from 8 that

$$\mathsf{p2} = \mathsf{p1} \tag{12}$$

and

$$\mathsf{rw} = \mathsf{p1} \tag{13}$$

From 13 together with 4 it follows that

$$\mathsf{left} \in \mathsf{ass}_{\mathsf{p1}} \tag{14}$$

and from 3 and 12 together with 14 it follows that

$$\mathsf{ass}_{\mathsf{p2}} = \{\mathsf{left}\} \tag{15}$$

and from that it follows that

$$\mathsf{right} \notin \mathsf{ass}_{\mathsf{p2}} \tag{16}$$

Now C6 follows from A11 and 16 and thus we have proved that the condition in node C is false and moreover that parse tree *t2* is precedence incorrect. This concludes the proof that at most one parse tree in an R-ambiguity-pair is precedence correct.

Note that it is possible that both parse trees in an R-ambiguity-pair is precedence incorrect.

□

7 Parsers and Precedences

Given the definition of precedence correct parse trees it is of course important that it is easy to construct parsers that recognize a language defined by a precedence grammar.

It is always possible to find a parser. We can use a parser constructed from the ambiguous grammar without precedences and let it give all parse trees for a sentence as result. We must of course use a parsing technique that can handle ambiguous grammars. Afterwards, the precedence incorrect parse trees can be filtered out. The filtering is possible since all attribute values and conditions are computable. Furthermore, an evaluation order can be found since our attribute grammars are L-attributed [ASU86, chapter 5.4].

Often, parse trees are not given as a result from a parser. Parse trees are just built implicitly and some other representation of the sentences are given as result. To filter out the precedence incorrect trees, the representation must be parse trees. Thus, we must first give parse trees as result and after (or during) the filtering, translate the parse trees to the desired representation. This process seems unnecessary inefficient and in this section will we investigate how the precedence rules can be incorporated into parsers. That is, the precedence incorrect trees will be thrown away during the parsing.

7.1 Precedences in Earley's algorithm

In this section we will show how precedence rules can be incorporated into Earley's algorithm [Ear70]. Earley's algorithm is a general context-free parsing algorithm. It takes a context-free grammar together with a sentence as input and gives all parse trees for that sentence as output. Our extended version of Earley's algorithm takes a precedence grammar together with a sentence as input and gives one, or sometimes more than one, precedence correct parse tree as result. If the precedence grammar contains other kinds of ambiguity than R-ambiguity then of course the algorithm still can give more than one parse tree as result but it does not give any precedence incorrect trees as result.

The algorithm works by scanning an input string from left to right. A set of items is constructed for each input symbol. Each item represents a possible derivation so far. The item sets are constructed by the operations predict, scan and complete.

The precedence checks are done in the operation complete. An item for which the operation complete is applicable contains a parse tree, or some other representation of the parsed substring. Only those items, in the item set pointed to, which pass the precedence checks for this parse tree, are added as the result of the operation complete. Items which do not pass the precedence checks are not added and incorrect parse trees will not be built.

To facilitate these checks, some information, besides the dotted production and the backward pointer, is added to the items. The extra information corresponds to the four attributes used in the attribute grammar that defines the precedence correct trees, i.e. left inherited precedences, right inherited precedences, left weights and right weights. There is also one component that is the precedence of the production in the item. An item in the extended version will be denoted as

$$lps, rps, p,\ A ::= ts \bullet \alpha, lws, rws, i$$

where	lps	is the inherited left precedences for the left hand side nonterminal,
	rps	the inherited right precedences,
	p	the precedence of the production,
	A	the left hand side nonterminal,
	ts	a sequence of parse trees,
	α	a sequence of terminals and nonterminals,
	lws	the left weights for the left hand side nonterminal,
	rws	the right weights, and
	i	is the backward pointer.

The values of the four new components are calculated in a way that follows the attribute evaluation rules in the attribute grammar that defines the precedence correct trees.

The left and right inherited precedences are added to an item by the operation predict. As in the ordinary algorithm, the operation predict is applicable when the dot is to the left of a nonterminal. If this nonterminal is the leftmost(rightmost) symbol then the left(right) precedences in the predicting item will be inherited to the new item that will be added.

The weight components in an item are the weights for the parse tree with the left hand side nonterminal as root. They are updated when the dot is moved over the leftmost and rightmost symbol respectively. We will in our examples use a question mark to indicate

when the weights have not yet been updated. Updating of the weights can be done both by the operations scan and complete.

Since the result of the operation scan is that the dot is moved over terminals, it is easy to update the weights in that case. When, for example, the dot is moved over a leftmost terminal, then the left weights are updated to be zero for the left hand side nonterminal and $\perp$ for all others.

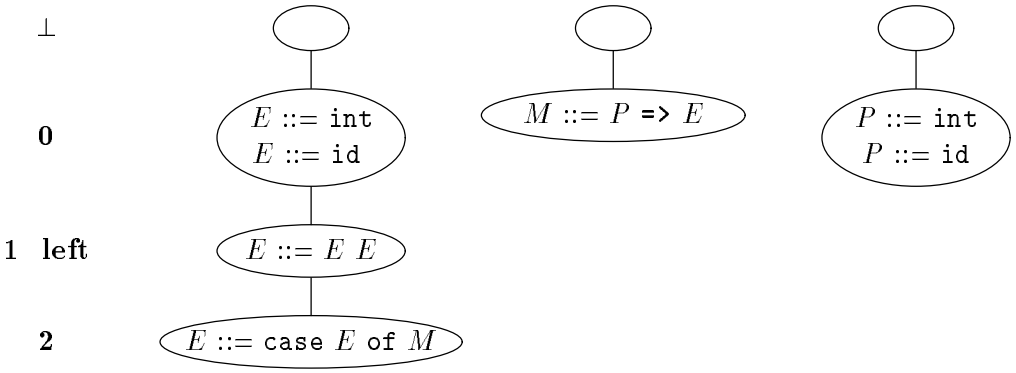
Updating the weights as a result of the operation complete is a bit more complicated since the dot then is moved over a nonterminal, but the way this is done follows closely the attribute evaluation rules for the weights. As mentioned before, the conditions are checked in the operation complete.

7.1.1 Example

Let us use the algorithm to parse the sentence `case e of 0 => f 5` with the precedence grammar:

$$
\begin{array}{lcl}
E & ::= & \texttt{int} \\
 & | & \texttt{id} \\
< L & | & E\ E \\
< & | & \texttt{case}\ E\ \texttt{of}\ M \\
 & & \\
M & ::= & P\ \texttt{=>}\ E \\
 & & \\
P & ::= & \texttt{int} \\
 & | & \texttt{id}
\end{array}
$$

The precedence orders for this grammar can be pictured as below where we have also given the natural numbers that will be used in the coding of the orders:



Note that there is one precedence order for each nonterminal and if there is no precedence relation between the productions then all productions will be in the same precedence group, We will indicate the precedence for productions that is left associative with an L. In the description we will subscript the precedences with the nonterminals. In this case it is only the nonterminal E that is interesting. We will not always present all items in all item sets, and if so we indicate with dots that there are more.

As in the ordinary algorithm, we first augment the grammar with a new start symbol and an endmark and get the following items in the first item set.

Item set 0

$$\begin{array}{l}
\bot\ ,\ \bot\ ,\ 0\ ,\ S ::= \bullet E \dashv \qquad\qquad ,\ ?\ ,\ ?\ ,\ 0 \\
\bot\ ,\ \bot\ ,\ 1\ ,\ E ::= \bullet E\ E \qquad\qquad ,\ ?\ ,\ ?\ ,\ 0 \quad (i) \\
\qquad\qquad\qquad\qquad \vdots \\
\bot\ ,\ \bot\ ,\ 2\ ,\ E ::= \bullet \texttt{case}\ E\ \texttt{of}\ M\ ,\ ?\ ,\ ?\ ,\ 0 \quad (ii)
\end{array}$$

The operation scan is applicable to item (ii) and since the terminal `case` is the leftmost symbol, we update the left weights when moving the item to item set 1.

Item set 1

$$\bot\ ,\ \bot\ ,\ 2\ ,\ E ::= \texttt{case} \bullet\ E\ \texttt{of}\ M\ ,\ 0_E\ ,\ ?\ ,\ 0 \quad (ii)$$

Now the operation predict is applicable. Since the nonterminal is neither the leftmost nor the rightmost symbol, no precedences are inherited.

Item set 1

$$\begin{array}{l}
\qquad\qquad \vdots \\
\bot\ ,\ \bot\ ,\ 0\ ,\ E ::= \bullet \texttt{id}\ ,\ ?\ ,\ ?\ ,\ 1 \\
\qquad\qquad \vdots
\end{array}$$

After a sequence of steps we get (ii) in item set 3. The dot is moved over three symbols.

Item set 3

$$\bot\ ,\ \bot\ ,\ 2\ ,\ E ::= \texttt{case}\ E\ \texttt{of}\ \bullet M\ ,\ 0_E\ ,\ ?\ ,\ 0 \quad (ii)$$

The operation predict is applicable to this item and since M is the rightmost symbol, the right precedences for E is inherited to the new item.

Item set 3

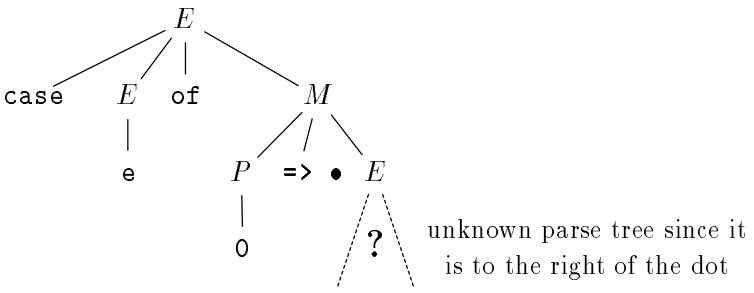
$$\begin{array}{l}
\qquad\qquad \vdots \\
\bot\ ,\ 2_E\ ,\ 0\ ,\ M ::= \bullet P\ \texttt{=>}\ E\ ,\ ?\ ,\ ?\ ,\ 3 \quad (iii) \\
\qquad\qquad \vdots
\end{array}$$

After some more steps we come to item set 5.

Item set 5

$$\bot\ ,\ 2_E\ ,\ 0\ ,\ M ::= P\ \texttt{=>}\ \bullet E\ ,\ 0_P\ 0_M\ ,\ ?\ ,\ 3 \quad (iii)$$

Let us take a closer look at item (iii) and what it represents. It is predicted in item set 3 by the item $\bot, \bot, 2,\ E ::= \texttt{case}\ E\ \texttt{of} \bullet M, 0_E, ?, 0$ and at the stage of item set 5 these represent the following parse:



That the inherited left precedences in item set (iii) is $\perp$ for all nonterminals means that the left edge above node M is empty, i.e. the node M has a left sibling, in this case the terminal `of`. That the inherited right precedence for E is 2 means that the first production for E on the right edge above M has precedence 2. The left weights in item (iii) are updated since the dot has been moved over the leftmost symbol and is zero for both nonterminal P and M since these occur once on the left edge below M. Since the dot is not yet moved over the rightmost symbol, the right weights are not yet updated and this is indicated by a question mark in the item.

Going back to the parsing process we note that the operation predict is applicable to item (iii) and the following items are added to item set 5.

Item set 5

$$\vdots$$
$$\perp \ , \ 0_M \ , \ 0 \ \ , \ E ::= \bullet\texttt{id} \ \ , \ ? \ , \ ? \ , \ 5 \quad (iv)$$
$$\perp \ , \ 0_M \ , \ 1^L \ , \ E ::= \bullet E \ E \ , \ ? \ , \ ? \ , \ 5 \quad (v)$$
$$\vdots$$

The operation scan is applicable to item (iv), and that item is moved to item set I_6 with the dot moved over the terminal `id`.

Item set 6

$$\perp \ , \ 0_M \ , \ 0 \ , \ E ::= \texttt{id}\bullet \ , \ 0_E \ , \ 0_E \ , \ 5 \quad (iv)$$

As a result of applying the operation complete to item (iv), two items from item set I_5 are added. Item (v) gets E twice on the left edge so in that case the weight for E is something else than zero.

Item set 6

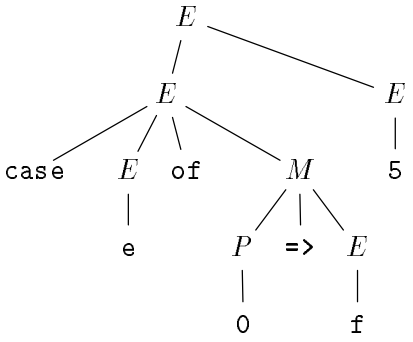
$$\vdots$$
$$\perp \ , \ 2_E \ \ , \ 0 \ \ , \ M ::= P \ \texttt{=>} \ E\bullet \ , \ 0_P \ 0_M \ , \ 0_E \ 0_M \ , \ 3 \quad (iii)$$
$$\perp \ , \ 0_M \ , \ 1^L \ , \ E ::= E \bullet E \ \ \ , \ 1_E \ \ \ , \ ? \ \ \ , \ 5 \quad (v)$$

The operation complete is applicable to item (iii) and as a result of that we add the following item to item set 6.

Item set 6

$$\vdots$$
$$\perp \ , \ \perp \ , \ 2 \ , \ E ::= \texttt{case} \ E \ \texttt{of} \ \bullet M \ , \ 0_E \ , \ 2_E \ 0_M \ , \ 0 \quad (ii)$$
$$\vdots$$

The dot is last in item (ii) and precedence checks are done for item (i) in item set 0. But the precedence check will not be okey this time. The right weight for the nonterminal E in the built parse tree is 2 and thus the parse tree is not accepted in the place of a leftmost E in a production with precedence 1. Therefore, the item (i) is not added to item set 6, and the precedence incorrect parse tree below, will not be constructed:



Proceeding to item set 7 where we get item (v).

Item set 7

$$\vdots$$

$$\perp\ ,\ 0_M\ ,\ 1^L\ ,\ E ::= E\ E\bullet\ ,\ 1_E\ ,\ 1_E\ ,\ 5 \quad (v)$$

The operation complete is applicable to this item and we go back to item set 5 to see if item (iii) will pass the precedence checks. Thus, we check if the left parse tree in the picture below is acceptable at the place of E in the right parse tree:

Item (iii) has a inherited right precedence for E and this is compared to the left weight for E in item (v). The precedence check will be okey since $2 > 1$ and the following item is added to item set 7.

Item set 7

$$\vdots$$

$$\perp\ ,\ 2_E\ ,\ 0\ ,\ M ::= P\ \texttt{=>}\ E\bullet\ ,\ 0_P\ 0_M\ ,\ 1_E\ 0_M\ ,\ 3 \quad (iii)$$

As a result of applying the operation complete to item (iii) in item set 7, item (ii) from item set 3 is added.

Item set 7

$$\perp\ ,\ \perp\ ,\ 2\ ,\ E ::= \texttt{case}\ E\ \texttt{of}\ M\bullet\ ,\ 0_E\ ,\ 2_E\ 0_M\ ,\ 0 \quad (ii)$$

The precedence check causes no problems since there is no precedences defined for the nonterminal M in item (ii). Note that the right weight for E in item (ii) is updated to be the maximum of the precedence of the production, (=2), and the right weight for E in item (iii), (=1).

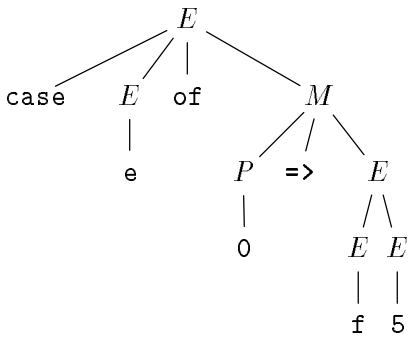
Finally, the item with the new start symbol in item set 0 is added to item set 7 as a result of applying the operation complete to item (ii).

Item set 7

$$\vdots$$

$$\perp\ ,\ \perp\ ,\ 0\ ,\ S ::= E \bullet \dashv\ ,\ 2_E\ 0_S\ ,\ ?\ ,\ 0$$

When the end symbol is scanned we are ready and the parser returns the precedence correct tree:




```
case e of 0 => f 5
```

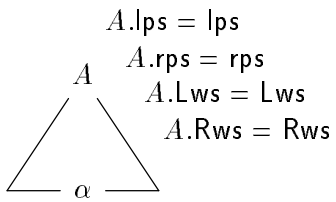
7.1.2 Correctness discussion

In this section we will motivate that our version of Earley's algorithm with precedences is correct. We state that only precedence correct trees are given as result from the parser. We will argue that this is true by showing how close the updating of precedence components are to the attribute evaluation rules. Moreover, the precedence check in the operation complete corresponds exactly to the condition in the attribute grammar.

An item in Earley's algorithm represents a (sub)parse tree. The precedence components are the attributes for the root of the parse tree. Thus, an item:

$$\mathsf{lps}\ ,\ \mathsf{rps}\ ,\ \mathsf{p}\ ,\ A ::= \alpha \bullet\ ,\ \mathsf{Lws}\ ,\ \mathsf{Rws}\ ,\ i$$

represents the parse tree:



We will show that the components: lps, rps, Lws and Rws are updated in such a way that

their values are equal to the values of the attributes in the corresponding annotated parse tree.

Left and right precedences

The components corresponding to the inherited attributes left- and right precedences are updated by the operation predict. The updating corresponds exactly to the attribute evaluation rules for left/right precedences. Let us consider the first rule for left precedences:

production	*attribute evaluation rule*
$B ::= A\alpha$	$A.\mathsf{lps} := B.\mathsf{lps}\left[\begin{array}{l} B \mapsto \mathsf{P}(B ::= A\alpha) \\ A \mapsto \bot \end{array}\right]$ $nt.\mathsf{lps} := \lambda x.\bot \qquad$ for all $nt \in \alpha$

This rule is incorporated into the operation predict and is used when predict is applied to an item:

Item set i

$\mathsf{lps}\ ,\ \mathsf{rps}\ ,\ \mathsf{pb}\ ,\ B ::= \bullet A\ \alpha\ ,\ ?\ ,\ ?\ ,\ i$

When applying the operation predict on this item, the left precedences, lps, is inherited to the new items. For a production $A ::= \gamma$ with precedence pa, we add the following item to item set i.

Item set i

$$\vdots$$

$$\mathsf{lps}\left[\begin{array}{l} B \mapsto \mathsf{pb} \\ A \mapsto \bot \end{array}\right]\ ,\ \bot\ ,\ \mathsf{pa}\ ,\ A ::= \bullet\ \gamma\ ,\ ?\ ,\ ?\ ,\ i$$

The right precedences are updated to $\bot$ which corresponds to the rules for right precedences. (We assume that α is nonempty.) If the dot is to the left of a rightmost nonterminal, the updating of the right precedences corresponds exactly to the first rule for right precedences. For nonterminals in the middle of a right hand side, the left and right precedences are updated to $\bot$ which also correspond to the rules.

Left and right weights

The weight components in the items correspond to the synthesized attributes left- and right weights. These components are updated when the dot is moved over the leftmost or rightmost symbol in the right hand side. Let us first consider the first rule for left weights.

production	*attribute evaluation rule*
$B ::= \mathsf{a}\alpha$	$B.\mathsf{Lws} := (\lambda x.\bot)[B \mapsto 0]$

This rule is incorporated into the scan operation and is used when scan is applied to an item:

Item set k

$\mathsf{lps}\ ,\ \mathsf{rps}\ ,\ \mathsf{p}\ ,\ B ::= \bullet\ \mathsf{a}\ \alpha\ ,\ ?\ ,\ ?\ ,\ i$

The left weights are updated in the new item that is added to the next item set.

$$\begin{array}{l}\underline{\textit{Item set } k+1}\\ \mathsf{lps}\ ,\ \mathsf{rps}\ ,\ \mathsf{p}\ ,\ B ::= \mathsf{a}\ \bullet\ \alpha\ ,\ 0_B\ ,\ ?\ ,\ i\end{array}$$

The second rule for left weights is a bit more complicated.

$$\begin{array}{ll}\underline{\textit{production}} & \underline{\textit{attribute evaluation rule}}\\ B ::= A\alpha & B.\mathsf{Lws} := \begin{cases} A.\mathsf{Lws}[B \mapsto max\ \mathsf{lw}\ \mathsf{p}] & \textit{if}\ \mathsf{lw} \neq \bot\\ A.\mathsf{Lws}[B \mapsto 0] & \textit{otherwise}\end{cases}\\ & \quad \textit{where}\ \ \mathsf{lw} = (A.\mathsf{Lws})\ B\\ & \qquad\qquad \mathsf{p} = \mathit{fst}(\mathsf{P}(B ::= A\alpha))\end{array}$$

This rule is incorporated into the operation complete, since the dot then is moved over a nonterminal. An item for which the operation complete is applicable has updatable left and right weights since the dot is last in such an item.

$$\begin{array}{l}\underline{\textit{Item set } k}\\ \mathsf{lps}'\ ,\ \bot\ ,\ \mathsf{pa}\ ,\ A ::= \gamma\ \bullet\ ,\ \mathsf{Lws}\ ,\ \mathsf{Rws}\ ,\ i\end{array}$$

Assume that this item is predicted by an item:

$$\begin{array}{l}\underline{\textit{Item set } i}\\ \mathsf{lps}\ ,\ \mathsf{rps}\ ,\ \mathsf{pb}\ ,\ B ::= \bullet\ A\ \alpha\ ,\ ?\ ,\ ?\ ,\ i\end{array}$$

If $\mathsf{Lws}\ B \neq \bot$ then the result of the operation complete is that the dot is moved over A and the left weights are updated as follows:

$$\begin{array}{r}\underline{\textit{Item set } k}\\ \mathsf{lps}\ ,\ \mathsf{rps}\ ,\ \mathsf{pb}\ ,\ B ::= A\ \bullet\ \alpha\ ,\ \mathsf{Lws}[B \mapsto max\ (\mathsf{Lws}\ B)\ (\mathit{fst}\ \mathsf{pb})]\ ,\ ?\ ,\ i\end{array}$$

If instead $\mathsf{Lws}\ B = \bot$, the result is:

$$\begin{array}{r}\underline{\textit{Item set } k}\\ \mathsf{lps}\ ,\ \mathsf{rps}\ ,\ \mathsf{pb}\ ,\ B ::= A\ \bullet\ \alpha\ ,\ \mathsf{Lws}[B \mapsto 0]\ ,\ ?\ ,\ i\end{array}$$

Updating of the right weights is analogous.

Precedence check

The precedence check corresponds exactly to the condition in the attribute grammar. Let us consider the last condition rule:

$$\begin{array}{ll}\underline{\textit{production}} & \underline{\textit{condition}}\\ A ::= B\alpha & \begin{cases} \text{true} & \textit{if}\ \underbrace{(A.\mathsf{lps}[A \mapsto \mathsf{P}(A ::= B\alpha)])\ B}_{\mathsf{lp}} = \bot\\ \mathsf{p} > \mathsf{rw} \vee (\mathsf{p} = \mathsf{rw} \wedge \mathsf{left} \in \mathsf{ass}) & \textit{otherwise}\\ \quad \textit{where}\ \ \mathsf{rw} = (B.\mathsf{Rws})\ B & \\ \qquad\quad \mathsf{p} = \mathit{fst}\ \mathsf{lp} & \\ \qquad\quad \mathsf{ass} = \mathit{snd}\ \mathsf{lp} & \end{cases}\end{array}$$

This condition, as the other ones, is incorporated into the operation complete:

Item set k

lps' , $\bot$, pb , $B ::= \gamma \bullet$, Lws , Rws , i

Assume that this item is predicted by the item:

Item set i

lps , rps , pa , $A ::= \bullet\ B\ \alpha$, ? , ? , i

This item will be added to item set k if the precedence check will pass, that is, the condition is true. We let, as in the condition rule, $\mathsf{lp} = \mathsf{lps}[A \mapsto \mathsf{pa}]$, $\mathsf{p} = \mathit{fst}\ \mathsf{lp}$, $\mathsf{ass} = \mathit{snd}\ \mathsf{lp}$, $\mathsf{rw} = \mathsf{Rws}\ B$. If $\mathsf{lp} = \bot$, the condition is true and the item is added to item set k. If $\mathsf{lp} \neq \bot$, the item is added if $\mathsf{p} > \mathsf{rw} \vee (\mathsf{p} = \mathsf{rw} \wedge \mathsf{left} \in \mathsf{ass})$.

Thus, incorrect parse trees will not be built and only precedence correct parse trees will be given as result.

7.2 Precedences in LR-parsing

Precedences (not ours) are widely used in LR-parsing but as shown in section 3, the way the user must use precedences is sometimes unnatural. Precedences are given to the productions which cause conflicts when the parsing table is built from the collections of items sets.

As we have already pointed out, a grammar with precedences that is given as input to an LR-parser generator is not equal to a precedence grammar generating the same language. Since we think that the precedence grammar is more intuitive and natural we will indicate how to generate an LR-parsing table from a precedence grammar.

The idea is to inherit precedences to the item sets where the conflicts occur. This is done when the collection of item sets is constructed. We add an extra component to the items, besides the dotted production. The extra component is a precedence component, a function from nonterminals to precedences. It contains the precedence of the production and possibly inherited precedences from other items. Let us denote a production $A ::= \gamma,\ p$ where p is the precedence of the production. An item is denoted $A ::= \alpha \bullet \beta,\ ip$ where ip is the precedence component. Following the construction of a collection of item sets as described in the "dragon" book [ASU86, chapter 4.7] we must extend the operation *closure*. The operation *goto* need no changes. In the closure operation, we let the precedence component inherit to the new items if the dot is to the left of a nonterminal which is the leftmost or rightmost symbol. Thus, if I is a set of items for a grammar G, then *closure(I)* is the set of items constructed from I by the rules:

1. Initially, every item in I is added to *closure(I)*.
2. If $A ::= \alpha \bullet B\beta,\ ip$ (where $\alpha, \beta \neq \epsilon$) is in *closure(I)* and $B ::= \gamma,\ p$ is a production in G, then add the item $B ::= \bullet\gamma,\ (\lambda x.\bot)[B \mapsto p]$ to I.
3. If $A ::= \bullet B\beta,\ ip$ is in *closure(I)* and $B ::= \gamma,\ p$ is a production in G, then add the item $B ::= \bullet\gamma,\ ip[B \mapsto p]$ to I.
4. If $A ::= \alpha \bullet B,\ ip$ is in *closure(I)* and $B ::= \gamma,\ p$ is a production in G, then add the item $B ::= \bullet\gamma,\ ip[B \mapsto p]$ to I.
5. Apply rules 2-4 until no more items can be added to *closure(I)*.

When a parsing table is constructed from the collection of items sets and there is a conflict, the precedence components are used. The conflict is resolved in favor of the item with lowest precedence.

Let us show how this is done with the grammar from section 3 as example. We give precedences to the nonterminal E since it is the only nonterminal which is both left- and right recursive.

```
E    ::=  AE                0
<     |   E AE               1
<     |   case E of M        2

AE   ::=  int
      |   id

M    ::=  P => E

P    ::=  int
      |   id
```

Below, a big part of the collection of item sets for this grammar is given. For simplicity, precedences of other productions than those for E are omitted. We let $\{E \mapsto p\}$ denote a function which returns the value p for the argument E and $\perp$ for all others. In item set I_4 we have the item:

$$E ::= \texttt{case}\ E\ \texttt{of}\ \bullet\ M\ ,\ \{E \mapsto 2\}$$

This item has the dot to the left of a rightmost nonterminal. Therefore, the precedence component is inherited to the new item that is added by the operation closure. Thus the item

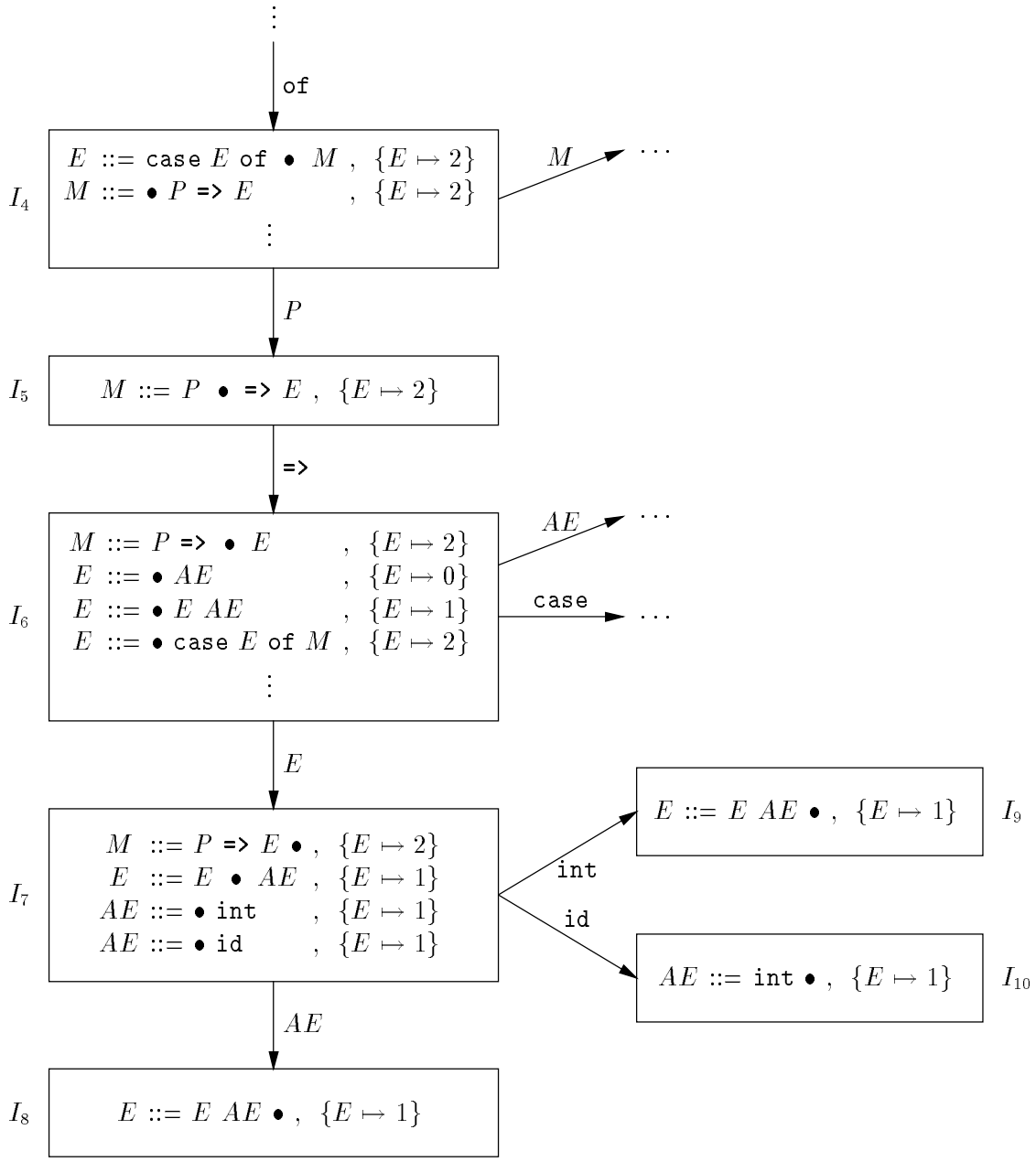
$$M ::= \bullet\ P\ \texttt{=>}\ E\ ,\ \{E \mapsto 2\}$$

is added to item set I_4. As results of the operation *goto*, this item with the dot moved over one symbol at a time appear also in item sets I_5, I_6 and I_7. In item set I_7 there is also an item:

$$E ::= E\ \bullet\ AE\ ,\ \{E \mapsto 1\}$$

The dot is to the left of a rightmost nonterminal and the precedence component is therefore inherited to the new items added by the operation closure.

$$AE ::= \bullet\ \texttt{int}\ ,\ \{E \mapsto 1\}$$
$$AE ::= \bullet\ \texttt{id}\ ,\ \{E \mapsto 1\}$$



When constructing a parsing table from this collection of item sets, item set I_7 generates conflicts. On input `int` and `id` there are conflicts between reduction by M ::= P `=>` E and shift.

These conflicts are resolved by the precedence components. Both items involved in one of these conflicts have precedences for E. Then the action in the parsing table will be the one that the item with lowest precedence gives rise to. Thus, there will be shift actions for inputs `int` and `id`. The action table for state 7 will be:

STATE	*action*				
	`int`	`id`	`case`	`of`	`$`
7	shift 9	shift 10			reduce by $M ::= P$`=>`E

8 Application

We have used the definition of precedences and an Earley parser in the parsing of conctypes [APS88]. How it is used is described in part III of this thesis.

9 Conclusion

We have suggested a way to resolve one kind of ambiguity, *R-ambiguity*, by giving precedences to alternatives of both left- and right recursive nonterminals. All kinds of ambiguity is not resolved but R-ambiguity is common and many programming construct are most easily expressed by R-ambiguous grammars. Therefore, having an intuitive and easy way to resolve R-ambiguity would improve the readability of many programming language descriptions.

The meaning of a grammar with precedences, *a precedence grammar* was given in terms of an attribute grammar. We showed also that it is quite easy to incorporate these precedence rules into parsers.

Appendix A

Lemmas

Lemma 11 *The precedence condition in the rule for productions $A ::= B\alpha$ is always true if the right weight for B in node B is zero.*

Proof: Consider the precedence condition for $A ::= B\alpha$.

production	*condition*	
$A ::= B\alpha$	true	$if(\underbrace{A.\mathsf{lps}[A \mapsto \mathsf{P}(A ::= B\alpha)])\ B}_{\mathsf{lp}} = \bot$
	$\mathsf{p} > \mathsf{rw} \vee (\mathsf{p} = \mathsf{rw} \wedge \mathsf{left} \in \mathsf{ass})$ *where* $\mathsf{rw} = (B.\mathsf{Rws})\ B$, $\mathsf{p} = fst\ \mathsf{lp}$, $\mathsf{ass} = snd\ \mathsf{lp}$	*otherwise*

Suppose that $\mathsf{rw} = 0$ and prove:

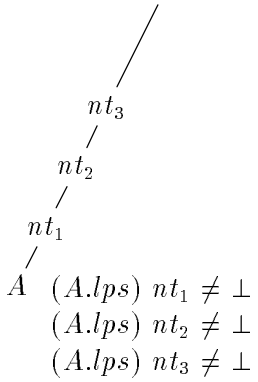
$$\mathsf{p} > \mathsf{rw} \vee (\mathsf{p} = \mathsf{rw} \wedge \mathsf{left} \in \mathsf{ass})$$

If $\mathsf{p} > 0$ then the condition is true since $\mathsf{p} > \mathsf{rw} = 0$. Suppose $\mathsf{p} = 0$. Then $\mathsf{p} = \mathsf{rw}$ and $\mathsf{left} \in \mathsf{ass}$ since if a precedence is zero then there can be no associativity restriction. □

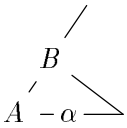
Lemma 12 *The precedence condition in the rule for productions $A ::= \alpha B$ is always true if the left weight for B in node B is zero.*

Proof: Analogous to the proof of lemma 11. □

Lemma 13 *All nonterminals for which the left precedence is not $\bot$ in a node belong to the left edge above the node.*



Proof: The only attribute evaluation rule which updates the left precedences is for productions of the form $B ::= A\ \alpha$. This production can be pictured as:



Thus, if the left precedence of a nonterminal is not $\bot$ it must have been updated by this rule. If the nonterminal is B then it clearly belongs to the left edge above A. If the nonterminal is not B then the left precedence must be hold in $B.\mathsf{lps}$, and we can apply the same argumentation again. □

Lemma 14 *All nonterminals for which the right precedence is not $\bot$ in a node belong to the right edge above the node.*

Proof: Analogous to the proof of lemma 13. □

Lemma 15 *If the right weight, in a node B_1, for the nonterminal B is greater than zero, then the nonterminal B appears twice on the right edge below B_1.*

Proof: From the attribute evaluation rules for right weights we get that if the right weight for B should be greater than zero then we must use the evaluation rule:

$$B.\mathsf{Rws} := A.\mathsf{Rws}[B \mapsto max\ \mathsf{rw}\ \mathsf{p}]$$

This rule is only applicable to productions of the form $B ::= \alpha\ A$ and when $(A.\mathsf{Rws}\ B) \neq \bot$. This means that B belongs to the right edge below A, as can be seen from the evaluation rules for the right weights. Thus, B appears twice on the right edge below B_1, inside B_1 and on the right edge below A. □

Lemma 16 *The precedence* p *in the condition for productions* $A ::= B\alpha$ *is the precedence of the first used production for* B *on the left edge above* B*.*

Proof: Ignoring the associativity, we have that p is

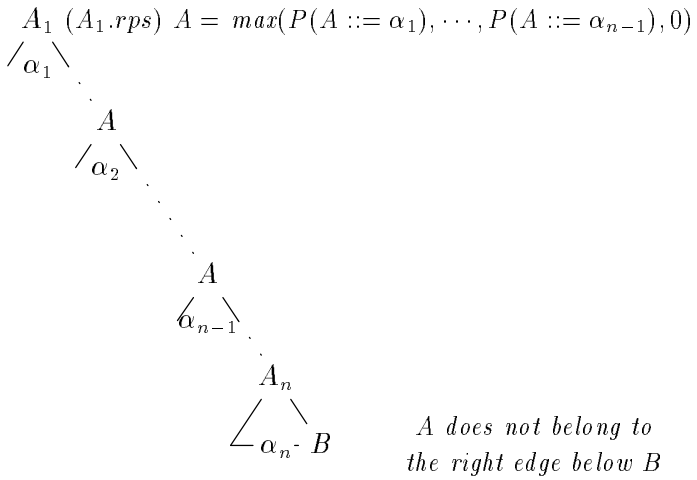
$$A.\mathsf{lps}[A \mapsto \mathsf{P}(A ::= B\alpha)]$$

If the nonterminals A and B are the same, then clearly p is the precedence of the first used production for B on the left edge above B. Since $\mathsf{p} \neq \bot$, it follows from lemma 13 that B belong to the left edge above A. Furthermore, p must be the precedence of the first occurrence of B as follows from the attribute evaluation rules for left precedences. □

Lemma 17 *The precedence* p *in the condition for productions* $A ::= \alpha B$ *is the precedence of the first used production for* B *on the right edge above the node containing* B*.*

Proof: Analogous to the proof of lemma 16. □

Lemma 18 *The right weight for a nonterminal* A *in a node is the maximum of all used productions for* A *on the right edge below the node, where the precedence of the last production for* A *is replaced by zero.*



Proof: Follows from the rules for right weights. The right weight for A in node A_n is zero, since A does not belong to the right edge below B. If A_n has a terminal as rightmost child this follows immediately from the first rule for right weights. Otherwise, there are a sequence of nodes, with other nonterminals than A, on the right edge below A_n and finally a terminal. In the last node the right weight for A will be $\bot$ and this value will be synthesized up to B. Then, it follows from the second rule for right weights that $(A_n.\mathsf{Rws})A = 0$.

The right weight for A in the nodes on the right edge below A_1 is updated every time A occurs. As follows from the second rule, it will each time be updated to the maximum of the precedence of the actual production and the right weight for A.

Thus, the right weight for A in node A_1 is the maximum of all used productions for A on the right edge below A_1 where the precedence of $A_n ::= \alpha_n B$ is replaced by zero.

□

Lemma 19 *The left weight for a nonterminal A in a node is the maximum of all used productions for A on the left edge below the node, where the precedence of the last production for A is replaced by zero.*

Proof: Analogous to the proof of lemma 18. □

Lemma 20 *If the productions for a nonterminal B are not given precedences then a subtree with B as root is accepted everywhere.*

Proof: Since no precedences are given, the weights cannot be more than zero and thus the precedence conditions cannot be violated as follows from the lemmas 11 and 12. □

Part VI

A Recursive Descent Parser for User Defined Distfix Operators

A Recursive Descent Parser for User Defined Distfix Operators

Annika Aasa

1 Introduction

The syntax of a programming language cannot possibly include every notation a user may want to use. It is therefore desirable to give her the possibility to extend the syntax. One way to do this for expressions is to offer user defined distfix operators. A distfix operator [Jon86] is an operator in which the operator name is distributed among its operands. A familiar example in most programming languages is the if-then-else construction with the syntax:

`if` *Expr*
`then` *Expr*
`else` *Expr*

Distfix operators make programs more readable, as can be seen in the following example. The left expression is in SML syntax and the right in LISP syntax[1]:

```
if x < 0                    (if (lt x 0)
then ~1                         -1
else if x > 0                   (if (gt x 0)
     then 1                         1
     else 0                         0))
```

An expression with repeated use of infix operators of type $\alpha \times \alpha \rightarrow \alpha$ is much more readable than the corresponding expression using functions with prefix notation. For example compare the arithmetic expressions:

```
2 + (3 * 4) + 6                 add (add 2 (mul 3 4)) 6
```

Infix, prefix and postfix operators are just special cases of distfix operators. With distfix operators it is possible to distinguish the operands using keywords and not just by position, which is the case if we use functions. This is illustrated below:

```
SUM fn x => x*x FROM 3-7*y TO z-8

sumfromto (fn x => x*x) (3-7*y) (z-8)
```

[1] If you regard the LISP syntax as the most readable you can stop reading now!

The two expressions are both supposed to reflect the mathematical formula:

$$\sum_{x=3-7y}^{z-8} x * x$$

Note that there is a distinction between an operator and a function. An operator is a syntactical construction. We cannot use an operator as an object in the same way as a function. An operator can, for example, not be given as argument to a function or another operator. A disadvantage with operators is that it is not obvious how to apply an operator to only some of its arguments. They cannot be curried in the same way as functions. In Haskell [Hea91] and LML [AJ87, AJ89] an operator is transformed to a function if it is enclosed in parentheses. For example, `(+ 2)` denotes the function `fn x => x+2` and `(+)` denotes the two argument function `fn x => fn y => x+y`.

Several programming languages give the user the possibility to define her own infix operators. Not as many languages provide user defined distfix operators. Two examples are however OBJ [FGJM85] and Hope [BMS80]. Implementations of distfix operators are seldom described in the literature. In [Jon86] there is a description of how distfix operators can be parsed with YACC [Joh75].

2 Introducing Distfix Operators

The usual way of describing the syntax of a programming language is by a context-free grammar. A parser for the language is somehow constructed from the grammar. In a language with user defined distfix operators the user is allowed to extend the syntax, so the parser must then be changed. The user must of course tell the parser which new distfix operators she wants to use in her program. To see how this can be done, let us consider expressions in a language generated by the following grammar:

$$\begin{array}{rcl} E & ::= & E \texttt{ + } E \\ & | & E \texttt{ = } E \\ & | & \texttt{if } E \texttt{ then } E \texttt{ else } E \\ & | & \texttt{int} \\ & | & \texttt{id} \end{array}$$

If a user wants to add a new distfix operator to the language she must give information which corresponds to one more production in the grammar. Since there is only one nonterminal in the grammar, it suffices to give the terminals[2] and mark where the nonterminal must be placed. In our experimental language this information is given in a distfix declaration:

```
distfix SUMNUMFROM _ TO _ ;
```

From the user's point of view, the underscores mark the positions where operands must be placed. From the parser's point of view, the underscores mark the places where the nonterminal E must be placed when the declaration is seen as a production in the grammar

[2]It is the terminals which compose the distfix operator.

above. After the distfix declaration the parser must be able to parse (among all other things) the language generated from the grammar:

$$
\begin{array}{rcl}
E & ::= & E \texttt{ + } E \\
 & | & E \texttt{ = } E \\
 & | & \texttt{if } E \texttt{ then } E \texttt{ else } E \\
 & | & \texttt{int} \\
 & | & \texttt{id} \\
 & | & \texttt{SUMNUMFROM } E \texttt{ TO } E
\end{array}
$$

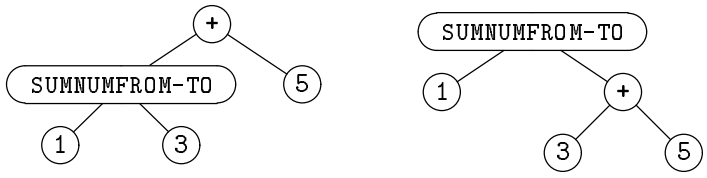
When the user has declared a distfix operator the distfix syntax can be used in expressions:

```
SUMNUMFROM 2 TO 9                    7 + SUMNUMFROM 2+1 TO 3*6
```

In this paper we will only consider *distfix grammars*, defined in part IV of this thesis, (definition 1 in section 2). A distfix grammar only contains one nonterminal and generates a language consisting of distfix operators. We will call the terminals which compose the distfix operator *operator words*. We note that distfix operators can be divided into four different categories:

1. *postfix distfix* where there is an operand to the left of the leftmost operator word but none to the right of the rightmost one. For example `_ A _ B`
2. *prefix distfix* where there is no operand to the left of the leftmost operator word but one to the right of the rightmost one. For example `A _ B _`
3. *infix distfix* where there are operands both to the left of the leftmost operator word and to the right of the rightmost one. For example `_ A _ B _`
4. *closed distfix* where there are no operands to the left of the leftmost operator word nor to the right of the rightmost one. For example `A _ B`

We want the parser to transform the expressions in our programs to syntax trees. Above we said that the effect of a distfix declaration is that a new production is added to a distfix grammar. This causes a problem. The resulting grammar can be ambiguous! How would, for example, the parser know which syntax tree we want for the sentence `SUMNUMFROM 1 TO 3 + 5`? There are two distinct syntax trees for this sentence:



One solution to this problem is to require that the user always parenthesizes an ambiguous expression, but this is not a good solution since many parentheses makes an expression hard to read. Another solution is to completely rewrite the grammar until it becomes unambiguous, but this introduces several new nonterminals, the grammar becomes quite big and unreadable, and it is hard to see how the user would indicate where a new distfix operator production must be placed in such a grammar. A third solution, which is

more suitable in this case, is to use precedence and associativity rules, that throw away undesirable syntax trees.

A definition of which syntax trees a *precedence grammar*, (a distfix grammar together with precedence rules), generates is given in part IV of this thesis (and also in [Aas91]). The generated syntax trees are called *precedence correct*. The definition is formal but here we restrict ourselves to a more informal description of how precedences eliminate ambiguity.

An operator with higher precedence has less binding power than one with lower. Thus, we have for the usual arithmetic operators that the addition operator `+` has higher precedence than the multiplication operator `*`. This notion of precedence is used for example in PROLOG [SS86] and OBJ [FGJM85].

Recall the ambiguous sentence `SUMNUMFROM 1 TO 3 + 5` and the two possible syntax trees for it. Suppose we know the precedence for the predefined infix operator `+`. If we want the sentence to be parsed as the first syntax tree we must give the distfix operator `SUMNUMFROM-TO` lower precedence than `+` and if we want it to be parsed as the second syntax tree we must give the distfix operator higher precedence than `+`. In our experimental language, information about the precedence of an operator is given when it is declared:

```
distfix 5 SUMNUMFROM _ TO _ ;
```

If `+` has precedence 3 this declaration gives that the right syntax tree on page 167 is correct. Note that we need precedence to resolve ambiguity only for the operands which are not enclosed in operator words.

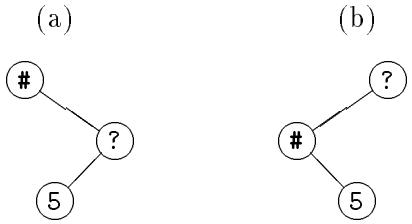
For infix distfix we must also give information to the parser if we want the operator to be left associative or right associative, that is if we for example want the expression `3 & 4 & 5` to be parsed as `(3 & 4) & 5`, (left associative) or `3 & (4 & 5)`, (right associative). In our experimental language an infix distfix operator becomes right associative if we use the word `distfixr` in the declaration:

```
distfixr 8 _ & _ ;
```

It is not necessary to have associativity rules for prefix and postfix distfix operators since there is an operator word on one side of the expression. For closed distfix operators it is not useful to have precedence and associativity rules at all since there are operator words enclosing the expression acting somewhat like parentheses. This, however relies on the fact that we will not accept that different operators use the same operator words. We cannot accept all possible combinations of precedences for the operators, either, since we then still can get ambiguities. An example of this is if we allow the user to define a prefix operator and a postfix operator with equal precedence:

```
distfix 2 _ ? ;
distfix 2 # _ ;
```

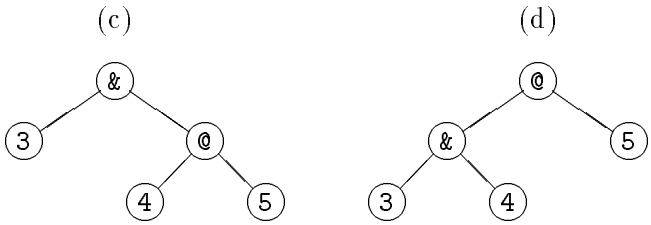
Then there are two possible parsings for the sentence `#5?`:



The precedences do not disambiguate the sentence. A similar example is:

```
distfix  2 _ & _ ;
distfixr 2 _ @ _ ;
```

That is, we define two infix operators with equal precedence but different associativity. The information in the declaration does not disambiguate the two possible parsings of the sentence `3&4@5`:



In YACC [Joh75] one always declares operators with associativity if one gives them precedence. If the operators in the first example are declared left associative then the sentence `#5?` will be parsed as syntax tree (b). That is, the behaviour is as if the postfix operator has higher precedence. Analogously, if they are declared right associative then the behaviour is as if the prefix operator has higher precedence and the sentence will be parsed as syntax tree (a). A third alternative in YACC is to declare the operators as nonassociative. Then there will be a syntax error for the string. Since it is not possible to declare operators with equal precedence and unequal associativity in YACC, the problem of deciding how to parse the string if `#` is left associative and `?` is right associative does not arise. Neither does the problem in the second example.

In SML, where one can define new infix operators, the parsing of the second example will be as syntax tree (d). That is, the behaviour is as if left associative infix operators has lower precedence than right associative ones. In our experimental implementation, we do not allow operators of different kinds to have the same precedence and therefore the problems above never arise.

3 A parser for user defined distfix operators

An experimental parser for user defined distfix operators has been implemented in SML [MTH90]. It is a recursive descent [DM81] parser and uses parser constructors due to Burge [Bur75] and Fairbairn [Fai87] and Kent Petersson and Sören Holmström [Pet85]. Using these parser constructors it is very easy to write a parser given a grammar. However, it is not possible to generate the parser directly from a precedence grammar. First, the precedence rules must be incorporated into the grammar itself. As shown in part IV of this

thesis (and also in [Aas91]) this is quite complicated if the precedence grammar contains infix, prefix and postfix operators of different precedence and not solely infix operators. Furthermore, since the parser is a recursive descent parser, the grammar must not be left recursive. In order to be able to eliminate the left recursion we use a variant of algorithm $\mathcal{M}$ given in part IV of this thesis (and in [Aas91]). Algorithm $\mathcal{M}$ transforms a precedence grammar to an ordinary context-free grammar. First we describe the parser constructors and use them to write parsers for languages containing only infix distfix operators. All program examples are written in SML.

3.1 An example of using parser constructors

A complete description of parser constructors and implementations of them can be found in [Pet85] and [Rea89, chapter 6.6] but for those only interested in an informal description of how our constructors are used, the following example is hopefully enough. So, let us use the parser constructors to write a parser for the language generated from the precedence grammar:

E	::=	E `+` E	2	left associative
	|	E `-` E	2	left associative
	|	E `*` E	1	left associative
	|	E `/` E	1	left associative
	|	`int`		
	|	`id`		

First, we must transform the precedence grammar to an ordinary context-free grammar in which the precedences are incorporated [ASU86, chapter 2.2]. Then, we must eliminate all left recursion [ASU86, chapter 4.3]. After this transformation we obtain the grammar:

E2	::=	*E1 EL2*
EL2	::=	*addop E1 EL2* | ϵ
E1	::=	*E0 EL1*
EL1	::=	*mulop E0 EL1* | ϵ
E0	::=	`int` | `id`
addop	::=	`+` | `-`
mulop	::=	`*` | `/`

We can now use the parser constructors and easily write a parser for the language. A parsing function (one for each nonterminal in the grammar) takes a list of terminal symbols as argument and if an initial part of the list is of correct form, the parsing function succeeds and returns a representation of the initial part together with the rest of the list. If no initial part of the list is of correct form (according to the grammar), the parsing function fails. The type of a parsing function is $(\alpha, \beta)Analyz$ where α is the type of the terminal symbols and β is the type of the representation of the part of the list that has been parsed. We do not have to go into more details about the datatype $(\alpha, \beta)Analyz$. Since the parsing functions are constructed from the parser constructors, we first briefly explain them.

‘ is used to construct a parsing function recognizing terminal symbols. It takes a terminal symbol as argument and gives a parsing function recognizing that terminal symbol as result.
The type is $\alpha \rightarrow (\alpha, \alpha)Analyz$.

`+-+` is used for sequencing. It takes two parsing functions as argument and returns a parsing function as result.
The type is $(\alpha,\beta)Analyz \times (\alpha,\gamma)Analyz \rightarrow (\alpha,(\beta \times \gamma))Analyz$.

`|||` is used for alternatives. It takes two parsing functions as argument and returns a parsing function as result that first tries the first parsing function and if that fails the other.
The type is $(\alpha,\beta)Analyz \times (\alpha,\beta)Analyz \rightarrow (\alpha,\beta)Analyz$.

`alt` is also used for alternatives. It takes a list of parsing functions (of the same type) as argument and returns a parsing function which tries the alternatives in order until one succeeds.
The type is $(\alpha,\beta)Analyz\,list \rightarrow (\alpha,\beta)Analyz$.

`==>` is used to make type transformations during parsing. It takes a parsing function and another function as argument and returns a parsing function recognizing the same thing as the argument parsing function but returns another representation of the parsed sentence. It is often necessary to use this constructor to get a type correct program.
The type is $(\alpha,\beta)Analyz \rightarrow (\beta \rightarrow \gamma) \rightarrow (\alpha,\gamma)Analyz$.

Suppose the parsing function `ID` recognizes an identifier, and the parsing function `INT` recognizes an integer constant, then the following functions define parsers from the languages generated from the nonterminals *E2*, *EL2*, *E1*, *EL1*, *E0*, *addop* and *mulop*. The parsing function `epsilon` succeeds for all input. Note the strong correspondence between the parser and the grammar:

```
fun E2 s    = (E1 +-+ EL2) s

and EL2 s   = (addop +-+ E1 +-+ EL2  |||  epsilon)  s

and E1 s    = (E0 +-+ EL1) s

and EL1 s   = (mulop +-+ E0 +-+ EL1  |||  epsilon) s

and E0 s    = (INT  |||  ID) s

and addop s = (` "+"  |||  ` "-") s

and mulop s = (` "*"  |||  ` "/") s
```

The parser is not supposed to do anything else than recognize sentences in the language. There is, however, one substantial problem with the program — it is not type correct! To make it type correct we must use the parser constructor `==>` which changes the type of a parsing function. This is not a problem in practice since a parser usually anyway translate a sentence to some other representation.

3.2 Translation to syntax trees

Normally a parser does not only recognize a language but does also translate sentences to syntax trees used by the rest of the compiler. Attribute grammars [ASU86, chapter 5] and [Knu68a, Knu68b] is one formalism for specifying translations for programming constructs. An attribute grammar specifies the translation of a construct in terms of *attributes* associated with its syntactic components. An attribute is said to be *synthesized* if its value at a parse tree is determined from attributes values at the children of that node. The value of an *inherited* attribute in a node can be defined in terms of attributes at the parents and/or siblings of that node. The values of the attributes are specified by evaluation rules given together with the productions. Let us turn our example grammar to an attribute grammar by specifying with evaluation rules how each syntactical construction will be translated to a syntax tree:

$$
\begin{array}{rcll}
E & ::= & E_v \texttt{ + } E_h & E.val := INFIX(E_v.val, \texttt{+}, E_h.val) \\
 & | & E_v \texttt{ - } E_h & E.val := INFIX(E_v.val, \texttt{-}, E_h.val) \\
 & | & E_v \texttt{ * } E_h & E.val := INFIX(E_v.val, \texttt{*}, E_h.val) \\
 & | & E_v \texttt{ / } E_h & E.val := INFIX(E_v.val, \texttt{/}, E_h.val) \\
 & | & \texttt{int} & E.val := NUM(\texttt{int}.val) \\
 & | & \texttt{id} & E.val := VAR(\texttt{id}.val)
\end{array}
$$

When eliminating left recursion we must also transform the evaluation rules if the resulting syntax tree would be the same as before. An algorithm for this is given in the "dragon" book [ASU86, chapter 5.5]. Besides the synthesized attributes in the attribute grammar above, we must then also use inherited attributes. The result of the transformation is the attribute grammar:

$$
\begin{array}{rcll}
E2 & ::= & E1 \;\; EL2 & EL2.in := E1.val \\
 & & & E2.val := EL2.val \\
 & & & \\
EL2 & ::= & addop \;\; E1 \;\; EL2_1 & EL2_1.in := INFIX(EL2.in, addop.val, E1.val) \\
 & & & EL2.val := EL2_1.val \\
 & & & \\
EL2 & ::= & \epsilon & EL2.val := EL2.in \\
 & & & \\
E1 & ::= & E0 \;\; EL1 & EL1.in := E0.val \\
 & & & E1.val := EL1.val \\
 & & & \\
EL1 & ::= & mulop \;\; E0 \;\; EL1_1 & EL1_1.in := INFIX(EL1.in, mulop.val, E0.val) \\
 & & & EL1.val := EL1_1.val \\
 & & & \\
EL1 & ::= & \epsilon & EL1.val := EL1.in \\
 & & & \\
E0 & ::= & \texttt{int} & E0.val := NUM(\texttt{int}.val) \\
 & & & \\
E0 & ::= & \texttt{id} & E0.val := VAR(\texttt{id}.val) \\
 & & & \\
addop & ::= & \texttt{+} & addop.val := \texttt{+} \\
 & | & \texttt{-} & addop.val := \texttt{-} \\
 & & & \\
mulop & ::= & \texttt{*} & mulop.val := \texttt{*} \\
 & | & \texttt{/} & mulop.val := \texttt{/}
\end{array}
$$

Here the attribute *in* is inherited while the attribute *val* is, as before, synthesized. When implementing a translation specified by an attribute grammar in SML, inherited attributes are given as arguments to the parsing functions and synthesized attributes are the results from the parsing functions. From the first production, for example, we see that the result (a syntax tree) from the parsing function `E1` must be given as argument to the parsing function `EL2`. This schema often occurs in the evaluation rules and we therefore introduce another parser constructor `+->`, which works almost as `+-+`. That is, it is used for sequencing in the grammar, but the result from the first operand (a parsing function) is given as argument to the second operand. The type of `+->` is:

$$(\alpha, \beta)Analyz \times (\beta \rightarrow (\alpha, \gamma)Analyz) \rightarrow (\alpha, \gamma)Analyz$$

With this new parser constructor we can easily write parsing functions for our language which gives syntax trees as result. For convenience we change the parsing function `epsilon` a bit. It succeeds for all inputs and gives its extra argument as result.

```
type Operator = string;
datatype Tree = INFIX of Tree * Operator * Tree
              | VAR   of string
              | NUM   of int;

fun E2 s     = (E1 +-> EL2) s

and EL2 lt   =
          addop +-+ E1 ==> (fn(aop,rt)=>INFIX(lt,aop,rt)) +-> EL2
     |||  epsilon lt

and E1 s     = (E0 +-> EL1) s

and EL1 lt   =
          mulop +-+ E0 ==> (fn(mop,rt)=>INFIX(lt,mop,rt)) +-> EL1
     |||  epsilon lt

and E0 s     = (INT  |||  ID) s

and addop s  = (` "+"  |||  ` "-") s

and mulop s  = (` "*"  |||  ` "/") s
```

3.3 User defined infix operators

The functions `E1` and `E2` as well as the functions `EL1` and `EL2` are almost identical. If we parameterize the parsing functions with the precedence we just need one function for all `E`*n*s and one for all `EL`*n*s. We must also have a parsing function `inop` which recognizes infix operators with a specific precedence.

```
fun E 0     = AE
 |  E n     = E (n-1) +-> EL n

and EL n lt =
       inop n +-+ E (n-1) ==> (fn(Op,rt)=>INFIX(lt,Op,rt)) +-> EL n
  |||  epsilon lt

 and AE s   = (INT  |||  ID) s

 and inop n =
      alt (map (` o snd)
               (filter (fsteq n) [(2,"+"),(2,"-"),(1,"*"),(1,"/")]))

and fsteq n (p,_) = p=n;
```

Now it is not difficult to see how user defined infix operators can be implemented. Just parse the infix declarations[3] and build up an operator-precedence list. The predefined operators must of course be in this list too. Give the list as argument to a parsing function, `parse`, in which the functions `E`, `EL` and `inop` are contained. The body of parse is a call to `E m`, where `m` is the highest precedence:

```
fun parse operatorlist =
  let fun E 0     = ...
       |  E n     = ...

      and EL n lt = ...

      and AE s    = ...

      and inop n  = alt (map (` o snd) (filter (fsteq n) operatorlist))
  in
     E (maxprec operatorlist)
  end
```

What about function application using juxtaposition? It causes us no problem since it can be seen as an invisible predefined infix operator. We must only decide the precedence of it and we cannot allow other operators to have the same precedence. In most functional languages, function application has the lowest possible precedence. This is not necessary. It is no problem allowing the user the possibility of defining operators which have lower precedence than function application. Still, I think it is most convenient that function application has the lowest precedence.

3.4 User defined infix distfix operators

Let us now see how we can introduce user defined infix distfix operators. Recall that an infix distfix operator is an operator consisting of a number of operator words with operands

[3] with parser constructors of course!

both to the left of the leftmost operator word and to the right of the rightmost operator word. For simplicity we assume that all operators are left associative. With some changes, we can use the same method as we introduced in section 3.3 for parsing infix operators.

An important thing is that the operands between the operator words are seen as belonging to the operator while the operands outside the operator words are not. We will therefore change the parsing function `inop` which only recognizes one operator word (the infix operator) to a parsing function which recognizes a number of operator words and expressions between them. For example, if the following distfix declaration has been made:

```
distfix 9 _ A _ B _ C _  ;
```

Then the parsing function `inop 9` must be able to recognize the sequence:

```
A expr B expr C
```

The parsing function `inop` uses an auxiliary function, `mkopanalyz`, which takes a parsing function recognizing expressions and a list of operator words as argument and returns a parsing function recognizing the sequence of operator words with expressions between them. We have for example that `mkopanalyz expr ["A","B","C"]` returns a parsing function which recognizes the sequence `A expr B expr C`:

```
fun inop n = alt (map (mkopanalyz expr o snd)
                      (filter (fsteq n) operatorlist)
                 )
```

(Here, `expr` is the parsing function recognizing expressions with highest precedence).

The parsing function `EL` must be changed a bit since it must handle the value returned from the new variant of `inop`. This means that the code for transforming the recognized expressions and operators has to be changed in order for the structure to be correct.

3.5 A complete distfix parser

In the complete distfix parser we use a simplified version of algorithm $\mathcal{M}$ given in part IV of this thesis (and in [Aas91]) that transforms a precedence grammar to an unambiguous context-free grammar. The simplified algorithm does not always give an unambiguous grammar as result but a recursive descent parser constructed from such a grammar will always give precedence correct syntax trees.

Note that *inl*, *inr*, *pre*, *post* and *closed* are distfix operators. As an example, *post* is a shorthand for

$$op_1\ E(m,0)\ op_2\ E(m,0)\ \cdots\ E(m,0)\ op_n$$

where $op_1, \ldots, op_n$ are the operator words in *post* and m is the highest precedence of all distfix operators.

3.5.1 A simplified version of algorithm $\mathcal{M}$

1. Give the operators numbers in the same way as is done in the original algorithm.
2. Introduce the nonterminals $E(n,q)$ and the following productions.
   (a) The rule for left associative infix operators.

$$\begin{array}{lcl} E(\mathsf{P}(inl_i),q) & ::= & E(\mathsf{P}(inl_i),q)\ \ inl_i\ \ E(\mathsf{P}(inl_i)-1,0) \\ & | & E(\mathsf{P}(inl_i)-1,q) \end{array}$$

   where $1 \le i \le$ number of left associative infix operators
   $0 \le q \le$ number of postfix operators with higher precedence than inl_i

   (b) The rule for right associative infix operators.

$$\begin{array}{lcl} E(\mathsf{P}(inr_i),q) & ::= & E(\mathsf{P}(inr_i)-1,q)\ \ inr_i\ \ E(\mathsf{P}(inr_i),0) \\ & | & E(\mathsf{P}(inr_i)-1,q) \end{array}$$

   where $1 \le i \le$ number of right associative infix operators
   $0 \le q \le$ number of postfix operators with higher precedence than inr_i

   (c) The rule for prefix operators.

$$E(\mathsf{P}(pre_i),q) \quad ::= \quad E(\mathsf{P}(pre_i)-1,q)$$

   where $1 \le i \le$ number of prefix operators
   $0 \le q \le$ number of postfix operators with higher precedence than pre_i

   (d) The rule for postfix operators.

$$E(\mathsf{P}(post_i),q) \quad ::= \quad E(\mathsf{P}(post_i)-1,q+1)$$

   where $1 \le i \le$ number of postfix operators
   $0 \le q \le$ number of postfix operators with higher precedence than $post_i$

   (e) The A-rule.

$$\begin{array}{lcll} E(0,q) & ::= & AE & \\ & | & E(\mathsf{P}(post_i),q-i)\ \ post_i & \text{where } 1 \le i \le q \end{array}$$

   $0 \le q \le$ number of postfix operators

   (f) The B-rule.

$$\begin{array}{lcll} AE & ::= & \texttt{int} \mid \texttt{id} \mid etc & \\ & | & closed_j & \text{where } 1 \le j \le \text{number of closed operators} \\ & | & pre_i\ \ E(\mathsf{P}(pre_i),0) & \text{where } 1 \le i \le \text{number of prefix operators} \end{array}$$

3. The start symbol in the grammar is the nonterminal $E(m,0)$ where m is the highest precedence of the operators.

The simplification compared to the original algorithm is that the index in the nonterminals telling which uncovered prefix operators that are not allowed to occur are not present.

3.5.2 Elimination of left recursion

Before we try to implement a parser from a grammar constructed from our algorithm, we must eliminate all left recursion, both direct and indirect, [ASU86, chapter 4.3]. We start by eliminating the direct left recursion in the rule for left associative infix operators (rule 2a):

$$\begin{array}{rcl} E(\mathsf{P}(inl_i), q) & ::= & E(\mathsf{P}(inl_i) - 1, q) \;\; EL(\mathsf{P}(inl_i) \\ EL(\mathsf{P}(inl_i)) & ::= & inl_i \;\; E(\mathsf{P}(inl_i) - 1, 0) \;\; EL(\mathsf{P}(inl_i)) \\ & | & \epsilon \end{array}$$

where $1 \leq i \leq$ number of left associative infix operators
$0 \leq q \leq$ number of postfix operators with higher precedence than inl_i

The rule for right associative infix operators is not left recursive. However, left factoring [ASU86, chapter 4.3] makes the further transformation easier and the parser more efficient (rule 2b):

$$\begin{array}{rcl} E(\mathsf{P}(inr_i), q) & ::= & E(\mathsf{P}(inr_i) - 1, q) \;\; ER(\mathsf{P}(inr_i) \\ ER(\mathsf{P}(inr_i)) & ::= & inr_i \;\; E(\mathsf{P}(inr_i), 0) \\ & | & \epsilon \end{array}$$

where $1 \leq i \leq$ number of right associative infix operators
$0 \leq q \leq$ number of postfix operators with higher precedence than inr_i

The rules for prefix and postfix operators are not left recursive but the A-rule contains indirect left recursion. To realize this consider the derivation:

$$\underline{E(0,q)} \rightarrow E(\mathsf{P}(post_i), q - i) \; post_i \rightarrow^* \underline{E(0,q)} \; \cdots \; post_i$$

To eliminate the indirect left recursion we start by transforming the following production until it contains direct left recursion:

$$E(0,q) \;\; ::= \;\; E(\mathsf{P}(post_i), q - i) \;\; post_i$$

In each transforming step we replace the first nonterminal in the production by the right hand side of the production for the nonterminal. After the first step we get the production below since we know there is a postfix operator with precedence $\mathsf{P}(post_i)$:

$$E(0,q) \;\; ::= \;\; E(\mathsf{P}(post_i) - 1, q - i + 1) \;\; post_i$$

In the following steps the production we get depends on the grammar. We make the following reflections. If we are about to replace the nonterminal $E(n,r)$, the result depends on which kind of operator there is with precedence n. There are four possibilities:

- A left associative infix operator.
  Then the nonterminal is replaced by $E(n-1,r) \;\; EL(n)$.
- A right associative infix operator.
  Then the nonterminal is replaced by $E(n-1,r) \;\; ER(n)$.
- A postfix operator.
  Then the nonterminal is replaced by $E(n-1,r+1)$.
- A prefix operator.
  Then the nonterminal is replaced by $E(n-1,r)$.

In each replacement step the first index is decremented by one. When we after n replacements have the nonterminal $E(0,q)$ first, the production has the form:

$$E(0,q) \;::=\; E(0,q)\;\; E[L|R](n')\;\; \cdots\;\; E[L|R](n'')\;\; post_i$$

where $E[L|R](n)$ is either $EL(n)$ or $ER(n)$ depending on if there is a left associative or right associative infix operator with precedence n. n' is the lowest precedence where there are infix operators and n'' is the highest precedence below $\mathsf{P}(post_i)$ where there are infix operators. We can now eliminate the direct left recursion and achieve the non left recursive A-rule.

$$\begin{array}{rcll} E(0,q) & ::= & AE\;\; AEP(q) & \\ AEP(q) & ::= & E[L|R](n')\;\; \cdots\;\; E[L|R](n'')\;\; post_i\;\; AEP(q) & \text{where } 1 \le i \le q \\ & | & \epsilon & \end{array}$$

$0 \le q \le$ number of postfix operators

3.5.3 The parser

We will now explain how the complete distfix parser is constructed. The parser returns a syntax tree but we will not go into details of how this tree is constructed. A main function `parse` is given five lists of distfix operators. One for left associative infix distfix operators, one for right associative infix distfix operators, one for prefix distfix operators, one for postfix distfix operators and one for closed distfix operators.

The function `parse` contains a number of local parsing functions and auxiliary functions. Here we will only explain the most important of them. As usual in recursive descent parsing, there is one parsing function corresponding to each nonterminal in the grammar above. That is, one for $E(n,q)$, one for $EL(n)$, one for $ER(n)$, one for AE and one for $AEP(q)$. For the nonterminal $E(n,q)$ there is a parsing function `E` taking arguments corresponding to the indices n and q. If $n = 0$ the function corresponds to the first production in the A-rule.

$$E(0,q) \;::=\; AE\;\; AEP(q)$$

The parsing function become:

```
fun E(0,q) = AE +-> AEP q
```

If $n > 0$ the function corresponds to the other productions which have E on the left hand side:

- $E(\mathsf{P}(inl_i),q) \;::=\; E(\mathsf{P}(inl_i)-1,q)\;\; EL(\mathsf{P}(inl_i)$
- $E(\mathsf{P}(inr_i),q) \;::=\; E(\mathsf{P}(inr_i)-1,q)\;\; ER(\mathsf{P}(inr_i)$
- $E(\mathsf{P}(pre_i),q) \;::=\; E(\mathsf{P}(pre_i)-1,q)$
- $E(\mathsf{P}(post_i),q) \;::=\; E(\mathsf{P}(post_i)-1,q+1)$

The parsing function `E` must check which kind of operator there is with precedence n. We use a function `kindof` which returns what kind of operator having precedence n:

```
|  E(n,q) = (case kindof n of
               infixL  => E(n-1,q) +-> EL n
             | infixR  => E(n-1,q) +-> ER n
             | postfix => E(n-1,q+1)
             | prefix  => E(n-1,q)
            )
```

For the nonterminal EL(n) there is a parsing function `EL` taking two arguments, an integer corresponding to the index n and a syntax tree. The function `inop n` is a a parsing function recognizing infix distfix operators with precedence `n` (including the expressions between the operator words):

```
fun EL n lt = (inop n +-+ E(n-1,0)
               ==> (fn ((iop,ts),rt)=> INFIX(iop,(lt::ts)@[rt]))
              ) +-> EL n

            ||| epsilon lt
```

The function `EL` corresponds to the following productions:

$$EL(\mathsf{P}(inl_i)) \;::=\; inl_i \;\; E(\mathsf{P}(inl_i)-1,0) \;\; EL(\mathsf{P}(inl_i)) \;\mid\; \epsilon$$

There is also a parsing function `ER`:

```
fun ER n lt =  inop n +-+ E(n,0)
               ==> (fn ((inop,ts),rt)=> INFIX(inop,(lt::ts)@[rt]))

            ||| epsilon lt
```

This function corresponds to the productions:

$$ER(\mathsf{P}(inr_i)) \;::=\; inr_i \;\; E(\mathsf{P}(inr_i),0) \;\mid\; \epsilon$$

For the nonterminal AE from which the atomic expressions, the prefix expressions and the closed distfix expressions are generated there is a parsing function `AE`. The variables `preOps` and `closedOps` respectively are the lists of prefix distfix operators and closed distfix operators:

```
and AE s   =
     alt ( [ INT , ID ]
         @
          (map (fn words => mkopanalyz expr words ==> CLOSED) closedOps)
         @
          (map (fn (n,preops) => (mkopanalyz expr preops) +-+ E(n,0)
                ==> (fn ((preop,ts),t) => PREFIX(preop,ts@[t]))
               ) preOps )
         ) s
```

The function corresponds to the productions:

$$
\begin{array}{lcll}
AE & ::= & \texttt{int} \mid \texttt{id} & \\
 & \mid & closed_j & \text{where } 1 \leq j \leq \text{number of closed operators} \\
 & \mid & pre_i \;\; E(\mathsf{P}(pre_i), 0) & \text{where } 1 \leq i \leq \text{number of prefix operators}
\end{array}
$$

Finally we have productions for the nonterminal $AEP(q)$:

$$
\begin{array}{lcll}
AEP(q) & ::= & E[L|R](n') \;\; \cdots \;\; E[L|R](n'') \;\; post_j \;\; AEP(q) & \text{where } 1 \leq j \leq q \\
 & \mid & \epsilon &
\end{array}
$$

For this nonterminal there is a parsing function `AEP` taking two arguments, one which corresponds to the index q and the other a syntax tree:

```
and AEP q t =
   alt ((map (fn i=>(mkanalyz t (fromto 1 (Ppost i)) +-+ popf (Ppost i)
                     ==> (fn (pt,(postop,ts)) => POSTFIX(postop,pt::ts))
                    ) +-> AEP q )
             (fromto 1 q)
        ) @ [epsilon t] )
```

The function `mkanalyz` takes a syntax tree and a list of precedences as arguments and returns a parsing function recognizing the sequence $E[L|R](n') \;\; \cdots \;\; E[L|R](n'')$, where n' is the lowest precedence in the list for which there is an infix distfix operator and n'' is the highest precedence in the list for which there exists an infix distfix operator. The function `popf n` is a parsing function recognizing distfix postfix operators (including the expressions between the operator words) and `Ppost i` returns the precedence of the i:th postfix operator.

The body of the main function `parse` consists of a single call to the parsing function `E(m,0)` where `m` is the highest precedence of all operators.

4 Summary

We have shown how the expression part of a parser for a language with user defined distfix operators can be implemented using the parsing method recursive descent. To achieve a parser for the whole language we must of course also implement the other parts of the parser, for example the part that recognizes the distfix declarations and builds the operator lists.

Bibliography

Bibliography

[Aas89] Annika Aasa. Recursive Descent Parsing of User Defined Distfix Operators. Licentiate Thesis, Department of Computer Sciences, Chalmers University of Technology, S-412 96 Göteborg, Sweden, May 1989.

[Aas91] Annika Aasa. Precedences in Specifications and Implementations of Programming Languages. In J. Maluszynski and M. Wirsing, editors, *Proceedings of Third International Symposium on Programming Language Implementation and Logic Programming, Lecture Notes in Computer Science*, volume 528, pages 183–194. Springer-Verlag, August 1991.

[AJ87] Lennart Augustsson and Thomas Johnsson. *Lazy ML User's Manual.* Programming Methodology Group, Department of Computer Sciences, Chalmers, S–412 96 Göteborg, Sweden, 1987. To be distributed with the LML compiler.

[AJ89] Lennart Augustsson and Thomas Johnsson. The Chalmers Lazy-ML Compiler. *The Computer Journal*, 32(2):127–141, 1989.

[AJU75] A. V. Aho, S. C. Johnson, and J. D. Ullman. Deterministic Parsing of Ambiguous Grammars. *Communications of the ACM*, 18(8):441–452, August 1975.

[AM89] Andrew W. Appel and David B. MacQueen. A Standard ML Compiler. In *Proceeding of the ACM, Lisp and Functional Programming*, pages 301–324. ACM, 1989.

[APS88] Annika Aasa, Kent Petersson, and Dan Synek. Concrete Syntax for Data Objects in Functional Languages. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 96–105, Snowbird, Utah, 1988.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, Tools.* Addison-Wesley Publishing Company, Reading, Mass., 1986.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling Volume 1: Parsing* . Prentice-Hall, 1972.

[Aug84] Lennart Augustsson. A Compiler for Lazy ML. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 218–227, Austin, 1984.

[Bac60] J. Backus. The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference. In *Proceedings International Conference on Information Processing, June 1959*, pages 125–132, Unesco, Paris, 1960.

[Bac79] Roland Backhouse. *Syntax of Programming Languages, Theory and Practice.* Prentice Hall, 1979.

[BMS80] R. M. Burstall, D. B. McQueen, and D. T. Sannella. Hope: An Experimental Applicative Language. In *Conference Record of the 1980 LISP Conference*, pages 136–143, Stanford, CA, August 1980.

[Bur75] W. H. Burge. *Recursive Programming Techniques.* Addison-Wesley Publishing Company, Reading, Mass., 1975.

[Can62] David G. Cantor. On the Ambiguity Problem of Backus Systems. *Journal of the ACM*, 9(4):477–479, 1962.

[Car87] Luca Cardelli. Basic Polymorphic Typechecking. *Science of Computer Programming*, 8:147–172, 1987.

[CH90] G. Cousineau and G. Huet. The CAML Primer. Technical Report 122, INRIA, France, 1990.

[CU77] J. C. Cleaveland and R. C. Uzgalis. *Grammars for Programming Languages.* Elsevier North-Holland, 1977.

[DGKLM84] V. Donzeau-Gouge, G. Kahn, B. Lang, and B Mélèse. Document Structure and Modularity in Mentor. In *Proceedings of the ACM SIGSOFT/SIGPLAN - Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, 1984. Software Engineering Notes Vol. 9, No 3.

[DM81] A. J. T. Davie and R. Morrison. *Recursive Descent Compiling.* Ellice Horwood Limited, 1981.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9:th ACM Symposium on Principles of Programming Languages*, pages 207–212, Albuquerque, NM, January 1982.

[Ear70] Jay Earley. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

[Ear75] Jay Earley. Ambiguity and Precedence in Syntax Description. *Acta Informatica*, 4(2):183–192, 1975.

[Fai87] Jon Fairbairn. Making Form Follow Function: An Exercise in Functional Programming Style. *Software–Practice and Experience*, 17(6):379–386, 1987.

[FGJM85] K. Futatsugi, J. A. Goguen, J-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proceedings of the 12:th ACM Symposium on Principles of Programming Languages*, pages 52–66, New Orleans, Lousiana, January 1985.

[Flo62] Robert W. Floyd. On Ambiguity in Phrase Structure Languages. *Communications of the ACM*, 5(10):526–534, 1962.

[Flo63] Robert W. Floyd. Syntactic Analysis and Operator Precedence. *Journal of the ACM*, 10(3):316–333, 1963.

[FM88] You-Chin Fuh and Prateek Mishra. Type Inference with Subtypes. In H. Ganzinger, editor, *Proceedings ESOP '88. LNCS vol. 300*, pages 94–114, Nancy, France, 1988. Springer-Verlag.

[Ge83] G. Goos and J. Hartmanis (eds.). *The Programming Language Ada Reference Manual*, volume 155 of *Lecture Notes in Computer Science*. Springer-Verlag, 1983. American National Standards Institute, ANSI/MIL-STD-1815A-1983.

[GJ88] Dick Grune and Ceriel J.H. Jacobs. A Programmer-friendly LL(1) Parser Generator. *Software–Practice and Experience*, 18(1):29–38, 1988.

[GJ90] Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques a practical guide*. Ellis Horwood Limited, 1990.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[Gor79] M. Gordon. *The Denotational Description of Programming Languages*. Springer-Verlag, 1979.

[GTWW77] J. A. Gougen, J. W Thatcher, E. G. Wagner, and J. B. Wright. Initial Algebra Semantics and Continuous Algebras. *JACM*, 24(1):68–95, January 1977.

[Han85] David R. Hanson. Compact Recursive-descent Parsing of Expressions. *Software–Practice and Experience*, 15(12):1205–1212, 1985.

[Hea91] Paul Hudak and Philip Wadler et al. *Report on the Functional Programming Language Haskell*, August 1991. Version 1.1.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *ACM Sigplan Notices*, 24(11):43–75, 1989.

[HKR89] Jan Heering, Paul Klint, and Jan Rekers. Incremental Generation of Parsers. *ACM Sigplan Notices*, 24(7), 1989.

[Hoa75] C. A. R. Hoare. Recursive Data Structures. *International Journal of Computer and Information Sciences*, 4(2):105–132, 1975.

[Hue86] Gérard Huet. Formal Structures for Computation and Deduction. Lecture Notes for International Summer School on Logic Programming and Calculi of Discrete Design, Marktoberdorf, Germany, May 1986.

[INR85] INRIA. The ML Handbook, Version 6.1. Project Formel, Inria, May 1985.

[Joh75] S. C. Johnson. Yacc—Yet Another Compiler Compiler. Technical Report 32, Bell labs, 1975. Also in UNIX Programmer's Manual, Volume 2B.

[Jon86] Simon L. Peyton Jones. Parsing Distfix Operators. *Communications of the ACM*, 29(2):118–122, February 1986.

[Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[Knu64] Donald E. Knuth. Backus-Normal-Form vs Backus-Naur-Form. *Communications of the ACM*, 7(12):735–736, 1964.

[Knu65] Donald E. Knuth. On the Translation of Languages from Left to Right. *Information and Control*, 8(6):607–639, December 1965.

[Knu68a] Donald E. Knuth. Semantics of Context-free Languages. *Math. Systems Theory*, 2:127–145, 1968.

[Knu68b] Donald E. Knuth. Semantics of Context-free Languages: Correction. *Math. Systems Theory*, 5:95–96, 1968.

[Lan64] P. J. Landin. The Mechanical Evaluation of Expressions. *Computer Journal*, 6(4):308–320, 1964.

[LdR81] Wilf R. Lalonde and Jim des Rivieres. Handling Operator Precedence in Aritmetic Expressions with Tree Transformations. *ACM Transactions on Programming Languages and Systems*, 3(1):83–103, January 1981.

[Les75] M.E. Lesk. LEX - a lexical analyzer generator. CSTR 39, Bell Laboratories, Murray Hill, N.J, 1975.

[Mau89] Michel Mauny. Parsers and Printers as Stream Destructors Embedded in Functional Languages. In *In Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 360–370, London, 1989.

[MdR92] Michel Mauny and Daniel de Rauglaudre. Parsers in ML. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 76–85, San Francisco, California, 1992.

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and Systems Sciences*, 17:348–375, 1978.

[Mil84] Robin Milner. Standard ML Proposal. *Polymorphism: The ML/LCF/Hope Newsletter*, 1(3), January 1984.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Nau63] Peter Naur. Revised Report on the algorithmic language Algol 60. *Communications of the ACM*, 6(1):1–17, 1963.

[Par66] R. J. Parikh. On context-free languages. *Journal of the ACM*, 13(4):570–581, 1966.

[Pet85] Kent Petersson. LABORATION: Denotationsemantik i ML, 1985. Department of Computer Sciences, University of Göteborg and Chalmers University of Technology, S-412 96 Göteborg, Sweden.

[PF92a] Mikael Pettersson and Peter Fritzon. DML—a Meta-Language and System for the Generation of Practical and Efficient Compilers from Denotational Specifications. In *Proceedings of 4th IEEE International Conference on Computer Languages, ICCL'92*, 1992.

[PF92b] Mikael Pettersson and Peter Fritzson. A General and Practical Approach to Concrete Syntax Objects within ML. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, pages 17–27, San Francisco, California, 1992.

[PW83] Fernando C. N. Pereira and David H. D. Warren. Parsing as deduction. In *Proceedings of the 21'st Annual Meeting of the Association for Computational Linguistics*, pages 137 – 144, 1983.

[Qui81] Willard Van Orman Quine. *Mathematical Logic*. Harward University Press, 1981.

[Rea89] Chris Reade. *Elements of Functional Programming*. Addison-Wesley Publishing Company, Reading, Mass., 1989.

[San82] David Sandberg. LITHE: A Language Combining a Flexible Syntax and Classes. In *Proceedings of the 9:th ACM Symposium on Principles of Programming Languages*, pages 142–145, Albuquerque, NM, January 1982.

[Sha88] Michael Share. Resolving Ambiguities in the Parsing of Translation Grammars. *ACM Sigplan Notices*, 23:103–109, 1988.

[Sli91] Konrad Slind. Object Language Embedding in Standard ML of New Jersey. In *Proceedings of the SML workshop*, 1991.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, 1986.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, Mass., 1986.

[SW89] Daniel F. Stubbs and Neil W. Webre. *Data Structures with Abstract Data Types and Pascal*. Brooks/Cole Publishing Company, Pacific Grove, California, 1989.

[Udd88] Göran Uddeborg. A Functional Parser Generator. Licentiate Thesis, Department of Computer Sciences, Chalmers University of Technology, S-412 96 Göteborg, Sweden, 1988.

[Voi86] Frédéric Voisin. CIGALE: A Tool for Interactive Grammar Construction and expression parsing. *Science of Computer Programming*, 7:61–86, 1986.

[vWea75] A. van Wijngaarden et al. Revised Report on the Algorithmic Language ALGOL 68. *Acta Informatica*, 5:1–236, 1975.

[WAL+90] P. Weis, M.V. Aponte, A. Laville, M. Mauny, and A. Suarez. *The CAML Reference Manual*. INRIA, France, 1990. Technical Report 121.

[Wan84] Mitchell Wand. A Semantic Prototyping System. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 213–221, June 1984.

[Wha76] R.M. Wharton. Resolution of Ambiguity in Parsing. *Acta Informatica*, 6:387–395, 1976.

[Wir71] Niklaus Wirth. The Programming Language Pascal. *Acta Informatica*, 1(1):35–63, 1971.