Arbeitsbereich DBIS
Fachbereich Informatik
Universität Hamburg
Vogt-Kölln Straße 30
D-2000 Hamburg 54
Germany

Tycoon

Title:

Using the Tycoon Compiler Toolkit

Author: Gerald Schröder, Florian Matthes

Identification: DBIS Tycoon Report 061-92

Status: Initial Version

Date: June 1992

Description: This document explains the structure and the use of the Tycoon compiler toolkit, a library of generic modules in the Tycoon system library implemented in P-Quest.

Related Documents: *P-Quest* Installation and User Manual [Mat91]
P-Quest User Manual [NMM92] (in German)
The Quest Language and System (Tracking Draft) [Car90]

Contents

1 Introduction

The motivation for the *Tycoon Compiler Toolkit* comes from our experience in building and maintaining compilers and language-sensitive tools for Pascal/R, Modula-2, Modula/R, DBPL on various software platforms (e.g. see [MSS91, Nie91, SM91, Sch91]). Although the possible approaches to compiler construction and compiler modularization are well-explored and widely accepted, many algorithms have to be programmed from scratch for every new compiler. In particular, a substantial effort in the development of tools for database environments (schema design tools, transaction specification tools, proof assistants, or ad-hoc query interfaces) is devoted to the clerical tasks of scanning, parsing, unparsing and the recovery from erroneous user-input with precise error messages.

The *Tycoon Compiler Toolkit* supplies tools to construct and change these "simple" parts of compilers in a structured and standardized way within a strongly-typed polymorphic programming language. It does not rely on ad-hoc interfaces to external tools (like yacc, lex, bison, rex, ell) that do not allow to exploit the full power of persistent higher-order languages. It also provides a modular framework to reuse and exchange parts of existing compilers developed with the *Tycoon Compiler Toolkit* exploiting the module mechanisms of *P-Quest*. A standardized approach to the definition of new compilers is given in this text.

The toolkit is to be understood as a first step in a research programme where we are going to explore new ways in the usage of compiler toolkits as extensible language processor components in integrated database application systems [MS91]. This will be accomplished by using *one* language for the definition of *all* compiler components and by exploiting the language and system support of languages like *P-Quest* (like dynamic binding, persistence management, higher-order functions) to give up the strict separation between language definition, compiler generation and language utilization time

The toolkit has been designed by Florian Matthes who also programmed the parser generator. Gerald Schröder programmed the scanner generator and the demo.

This guide describes the steps required to build a compiler using the initial version of the toolkit. After a short introduction into the structure of compilers, a small demo compiler is presented. The source text of this demo is available in the Tycoon libraries and should be consulted for a deeper understanding of the examples.

2 Logical compiler structure

Before it can be judged what a compiler toolkit can afford, the general structure of a compiler has to be understood first. This description follows [ASU86], further details can be found in books on compiler construction, e.g. [WG85].

Several *logical* compiler *passes* can be distinguished. They successively transform a given source text into a machine-dependent object code. These passes do not have to correspond to *physical passes* on the source text and its internal representation. It is possible and usual practice to fold several logical passes into one physical pass. But for the construction of a compiler the main focus lies on logical passes.

In Figure 1 both main passes of a compiler are shown: The *frontend* and the *backend*. The frontend transforms the source text into a compiler dependent, but mostly machine independent, intermediate language. While this transformation takes place the compiler checks the source text for errors and reports them. The backend starts only when the frontend has not found any errors. It transforms the implicitly error-free intermediate representation into a machine dependent object

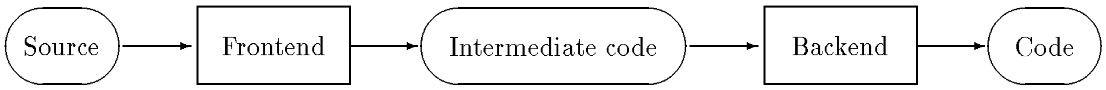


Figure 1: Overall compiler structure

code format, possibly enriched by some information for tools such as a linker, debugger or performance analyser.

The *Tycoon Compiler Toolkit* concentrates mainly on building the frontend of a compiler. So the backend is not mentioned in further detail.

In figure 2a) a model of the logical passes of the frontend is shown, while figure 2b) contains an example of the transformation steps. In the following paragraphs, the model of figure 2a) is explained with references to the example of figure 2b).

First, the source text, in an abstract view a *character stream*, is transformed by a *scanner* into a *symbol stream*. The scanner analyses the character stream and searches for a string, called *lexem*, matching some regular expression. If a lexem was found the corresponding symbol is put to the symbol stream.

In the example the scanner first finds the lexem "x" and puts the symbol *id* to the symbol stream. Then the lexemes "+" resp. "1" are found and the symbols *plus* resp. *num* are returned.[1]

The symbol stream is input to the *parser* which is divided into two parts. The first part checks if the symbols form a correct sentence of the language, usually called *program* or *module*. The language is described by a grammar, normally a context free grammar. The parser tries to match the symbol stream with the rules of the grammar. This process yields a *parse tree*, sometimes called *concrete syntax tree*, which shows how the grammar rules have been applied to form the sentence. The leaves of the parse tree are the input symbols to the parser.

In the example of figure 2b) there is a rule saying that an expression can be an identifier followed by a plus sign followed by a numerical literal. So, the first part of the parser builds the parse tree shown in figure 2b).[2]

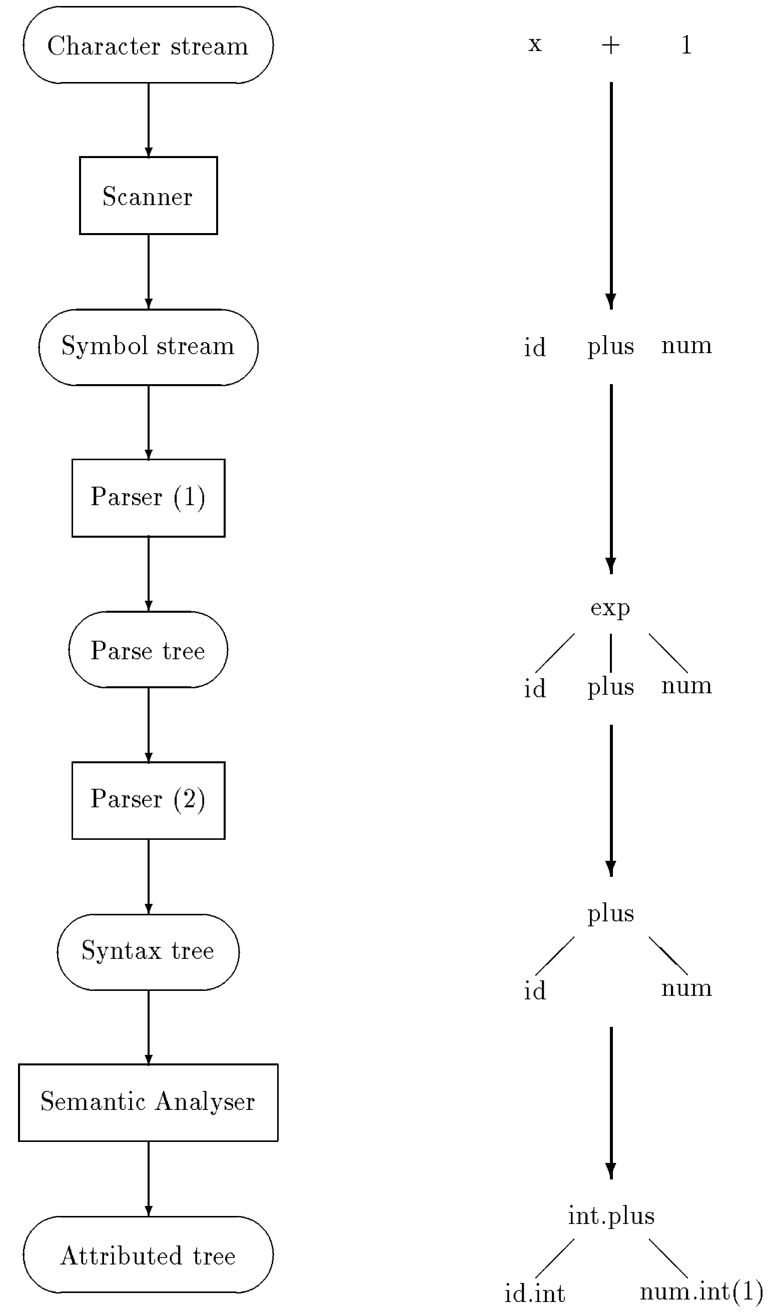
The parse tree is input to the second part of the parser which builds an *abstract syntax tree*. This tree contains only the bare syntax information, but abstracts from superfluous symbols such as parentheses or keywords as **begin**. The construction of the syntax tree is performed by *attributes* and *actions* that are attached to the grammar rules. Therefore, the grammar is called *attributed grammar*. The syntax tree is constructed while parsing a sentence.

In the example above, it is not necessary to know that the source text matches the rule describing an expression. But the information is preserved that an operation "plus" on two objects, namely an "id" and a "num", is mentioned in the source text; see figure 2b).

The syntax tree is further analysed by other passes which depend on the semantic rules of the language. These passes could check, e.g.,

- whether all used identifiers have been declared;

[1]In reality the scanner would not transform the *whole* source text into symbols *before* the next pass is started, but it would be called to return only the next symbol when this is needed.
[2]In reality the parse tree is never built explicitly, but exists implicitly in the parser structure.



a) Passes and data types b) Example values

Figure 2: Components of a compiler frontend

- whether the operands of expressions have compatible types, where "compatibility" has to be defined by the language, e.g. if real and integer operands are compatible;
- whether functions are called with the correct number and type of arguments.

In the example above, it is checked if the identifier "x" has been declared and if it is compatible to the numeric literal "1" which could be defined as an integer or real value.

Summing up, the frontend gives answers to the following questions:

1. Is the source text a correct sentence of the language?
2. What is an (error-free) intermediate representation of the source text?

3 Compiler toolkits

In this chapter two aspects of compiler toolkits are discussed:

1. Which achievements can be expected from compiler toolkits?
2. What has to be payed for these achievements?

Normally, compilers are implemented in some programming language and the programmer is responsible for the correctness of the implementation. With a compiler toolkit the focus lies on the *specification* of the language the compiler should work on and on the semantics of this language. If the specification is correct the toolkit generates a correct compiler. Thus the language designer has only the responsibility for the correctness of the specification. The toolkit helps to prove the correctness, e.g. by defining a language to specify the grammar rules and by some automatic checks on the rules.

If the language is changed or extended, normally the whole implementation of the compiler has to be reviewed. Experience teaches that errors are most likely to happen and are hard to find. With a toolkit only the specification of the language is changed and the toolkit guarantees that the newly generated compiler is error-free.

Compilers that have been coded by hand often are not modularized in the same manner. It is hard to understand which parts of two distinct compilers do the same things, e.g. which modules form the scanner and where the interface to the parser can be found. It is almost impossible to interchange parts of compilers. In a compiler toolkit the interfaces between the parts are well-defined and constant between all compilers. Therefore, it is easy to understand which part supplies which service and it is possible to exchange or reuse a part.

It is not a trivial problem to build a compiler. Compilers are very complicated tools where algorithms of many domains have to be used. Many problems have to be solved by any compiler, e.g. efficient reading and analysing of the source text, which takes most of the compiler runtime, and error recovery. Solving this problems again and again is time consuming, tedious and error prone. A compiler toolkit saves time and enables the language designer to concentrate on the really difficult task of defining the language, not implementing the compiler. By the way, hopefully the implementor of the toolkit has solved all common problems in the most efficient way, leading to fast and small compilers without bothering about the best algorithms for common problems.

But there are some drawbacks when using compiler toolkits:

1. Experience has shown that many hand-coded compilers are leaner and faster than automatically generated ones, although there are also counter examples.
2. Working with a toolkit leads to dependence on the tools, that is, loss of flexibility.

4 The structure of the *Tycoon Compiler Toolkit*

The toolkit can be separated into two parts:

1. Standardized interfaces between certain parts of a (generated) compiler.
2. Modules that build implementations for the specified interfaces.

4.1 Interfaces

The predefined interfaces of a compiler are:

Source:	The abstraction of a character stream, giving successive access to the characters of a source text.
Scanner:	The abstraction of a symbol stream, delivering symbols scanned from the source character stream.
Parser:	The abstraction of a syntactic analyser that checks the symbol stream for correctness and performs some actions, e.g. building a syntax tree.
ErrorLog:	The abstraction of a log for error messages generated by the compiler.

Matching implementations for these interfaces can be implemented in two ways:

1. Using of the toolkit which automatically generates the implementations.
2. Coding the implementions by hand.

4.2 Scanner generator

To build a concrete scanner for a set of symbols, the following modules can be used:

RegExpr:	Building regular expressions.
Symbol:	Representation of symbol values which can be used optionally.
LexicalAnalyzer:	The abstraction of a lexical analyser which is syntactically equivalent to a scanner but has the semantic difference that no “white spaces”, i. e. blank, tab, carriage return, comment, are skipped, and that no error recovery is done.
RegToLex:	This module takes a set of symbols and regular expressions defining the corresponding lexems, and produces a lexical analyser from these definitions.

Note that there is currently no tool to build a scanner. One or more lexical analysers are built automatically, but they have to be plugged together by hand to form a scanner. This will be done automatically later on but the problem of glueing them together has not been addressed yet. In the mean time, the module *demoReg.impl* can be used as a template; see section 5.3.

4.3 Parser generator

To build a concrete parser for a specific grammar the following modules can be used:

Grammar:	Specification of the grammar rules and attached actions, and transformation of the definitions to a parser. Currently, only LL(1)-grammars are supported.
Lexicon:	Mapping of keywords to symbols and grammar rules.

4.4 Summary

In contrast to other compiler toolkits, e.g. lex / yacc,

- no meta language is used to define scanners and parsers, but all definitions are performed in the host language *P-Quest*. This guarantees full type correctness already at language-definition time.
- no source code is produced which has to be compiled, but instead functions are generated and manipulated as first class language objects (flexible naming, dynamic binding, persistent storage, ...).

Therefore, the opportunity to construct or load parts of the compiler at *runtime* is gained. There is no possibility of type errors at runtime because *P-Quest* can perform all checks at compile time, especially on the actions attached to the grammar rules.

5 Example: Construction of an expression interpreter

In this chapter an expression interpreter is constructed using the *Tycoon Compiler Toolkit*. This example shows which steps you have to take on building your own compilers or interpreters.

5.1 Overview

The result of our effort is an expression interpreter, e.g. an interpreter that reads and parses expressions, and evaluates them instead of building a syntax tree.

The following modules are parts of the example:

demoMain:	Main program that uses the generated scanner and parser to interpret simple expressions.
demoGen:	Main program that generates a scanner, a parser, and a lexicon of keywords to save them on disk.
DemoGram, demoGram:	Top level definition of the parser.
DemoExp, demoExp:	Definition of expressions for the parser.
DemoReg, demoReg:	Definition of the scanner and two lexical analysers.
DemoSym, demoSym:	Definition of symbols and keyword lexicon.
Bindings, bindings:	Representation of values and simple mapping of identifiers to values.

To run the demo, follow these steps:

1. Compile the modules bottom up, that is, start with *Binding*, or use *make*.
2. Start *P-Quest*, import *demoGen* and call *demoGen.go()*. Now the scanner, parser and lexicon are generated.

3. Import *demoMain* and call *demoMain.go()*. You can choose the input from a file, e.g. *demo.txt*, or use interactive mode. In interactive mode, use `Ctrl-D` to terminate input; you may have to type `Ctrl-D` more than once.

5.2 The grammar

The expression interpreter accepts the following syntax (see section 5.5 for examples how these rules are implemented):

top `::=` { `'let' id '=' exp 'in'` | `exp` } `';'`
At top level we can perform binding of expressions to identifiers or evaluate expressions.

exp `::= term` { `('+'|'-') term` }
An expression is a term optionally plus or minus other terms. Evaluation order is left to right.

term `::= factor` { `('*'|'/') factor` }
A term is a factor optionally multiplied with or divided by other terms. Evaluation order is left to right.

factor `::= '(' exp ')' | int | real | id`
A factor is an expression put in parentheses or an integer literal or a real literal or an identifier.

For example, this is a correct top level input:

let *a* = *2* **in**
let *b* = *a* * *2* **in**
let *c* = *b* - *2* **in**
a * (*b* - *c* + *2*) / *8* ;

The result is “1”.

5.3 Scanner definiton

The scanner is defined by *demoReg* and it is generated by *demoGen*. It is composed of two lexical analysers: The first one skips whitespaces like blanks, tabs and carriage returns, the second one recognizes the symbols of the language. Each symbol is accompanied with a regular expression defining the lexem. Here is a list of symbols and regular expressions:

symbol	regular expression
point	'.'
equal	'='
plus	'+'
minus	'–'
times	'*'
div	'/'
leftparen	'('
rightparen	')'
int	{'0'..'9'}$^+$
real	{'0'..'9'}$^+$ '.' {'0'..'9'}$^+$
id	{'A'..'Z', 'a'..'z'}{'A'..'Z', 'a'..'z', '0'..'9', '_'}$^+$
wrong	*predefined*
endOfInput	*predefined*

The symbols *wrong* and *endOfInput* are predefined by the module *Symbol*. The keywords **let** and **in** are recognized as symbol *id* and have to be separated from identifiers by using the keyword lexicon defined in *DemoSym*.

To define a scanner, take the following steps:

1. Find the symbols and their description by regular expressions. The symbol that matches the longest possible input is returned, i. e. "1.0" is *real* and not *int*, *point*, *int*. Ambiguous definitions are not permitted, i. e. it is not possible to define keywords if they are matching the regular expression defining identifiers.[3] It is not possible to declare a right context, i. e. that "1.." should be treated as *int* and *ellipsis* instead of *real* and *point*.

   For example, the following lines (taken from *demoReg*) describe the lexem of the symbols *int* and *real*, using the previously generated object *lexGen*:

   ```
   let lexGen =
   regToLex.new(:DemoSym_T
     symbol.wrong
     symbol.endOfInput
   )
     (* get generator for lexical analyser (object)*)

   let digitR = regExpr.range('0' '9')
     (* digit in the range from 0 to 9 *)
   let digitsR = regExpr.rep(digitR)
     (* repetition of digits, at least one *)

   lexGen.setSymbol(digitsR demoSym.int)
     (* one or more digits form an integer number *)

   let realR =
     regExpr.seq of
       digitsR
       regExpr.string(".")
       digitsR
     end

   lexGen.setSymbol(realR demoSym.real)
     (* a sequence of digits, dot, digits forms a real number *)
   ```

2. Find strategies for scanning the input. I. e. whether whitespaces should be skipped; whether nested comments, which cannot be described by regular expressions, are allowed; how the scanner recovers from errors; ...

3. Define one or more lexical analysers with the modules *RegToLex* and *RegExpr*. Note that a lexical analyser does no error-reporting or error-recovery, but delivers the symbol *wrong* instead. A lexical analyser is generated by calling the method *generate* which raises an exception if the definition is erroneous.

   See module *demoReg* for the definition of two lexical analysers. One lexical analyser skips whitespaces, the other recognizes the symbols of the language. Note that the generation of an analyser could take place at compile-time if it would be declared in the body of the module, i. e.

[3] Some lookup in a hash table might be more space and time efficient than a lexical analyser distinguishing dozens of keywords.

let *analyser* = *lexGenObject.generate(...)*

In this example the analysers are generated at run-time when the function *demoReg.generator* is called.

4. Plug the lexical analysers together according to your strategies. In the example this is done in the module *demoReg* locally in the function *generator*. The following strategies are implemented: Whitespaces are skipped, no comments are allowed, errors are reported and error recovery is simply repetition of skipping and scanning until a valid symbol or end-of-input is found. Note that there is a problem recognizing end-of-input because the module *Source* only reports *once* that end-of-input is reached. So only one lexical analyser recognizes end of input but maybe delivers a valid symbol if the previous read characters formed a correct lexem.

   The full source text of the scanner generator function is listed in figure 3.

5. *demoReg.generator* returns a scanner function that needs a concrete source text and a concrete error report function to become a concrete scanner fulfilling the definition of delivering a symbol stream. In *demoReg* this function is declared locally to the function *generator*; it is an unnamed function that is returned as result of the function *generator*.

In this process there are many possibilities to save something to reuse it later on:

- Save the *definition of a lexical analyser* and extend it later on to recognize more symbols:

  ```
  let lexGen =
    regToLex.new(:Symbol_T
      symbol.wrong
      symbol.endOfInput
    )

  dynamic.extern(
    writer.file("lexGen")
    dynamic.new(lexGen)
    dynamic.fast)
  ```

- Save the *generated lexical analyser* to use it in another scanner that needs the same functionality:

  ```
  let analyser = lexGen.generate(symbol.wrong symbol.endOfInput)

  dynamic.extern(
    writer.file("analyser")
    dynamic.new(analyser)
    dynamic.fast)
  ```

- Save the *scanner function* to use it in a compiler for arbitrary source input.

  ```
  let scanner = demoReg.generator(symbol.wrong symbol.endOfInput)

  dynamic.extern(
    writer.file("scanner")
    dynamic.new(scanner)
    dynamic.fast)
  ```

```
let generator :Scanner_Generator(DemoSym_T) =
  fun(wrong, endOfInput :DemoSym_T) :Scanner_Binder(DemoSym_T)
begin
  let skipBinder :LexicalAnalyzer_Binder(DemoSym_T) =
    skipGen.generate(wrong endOfInput)
  let lexBinder :LexicalAnalyzer_Binder(DemoSym_T) =
    lexGen.generate(wrong endOfInput)
  fun(src :Source_T err :Scanner_ErrorReport) :Scanner_T(DemoSym_T)
  begin
    let lex = lexBinder(src)
    let skip = skipBinder(src)
    let local =
    tuple
      let var sym = wrong
      let var eof = false
    end
    let nextSym() :Ok =
    begin
      if not(local.eof) then
        skip.next()
        local.eof := skip.sym() is endOfInput
        if not(local.eof) then
          lex.next()
          local.eof := lex.sym() is endOfInput
        end
      end
    end
    tuple
      let sym() :DemoSym_T = local.sym
      let string = lex.string
      let position = lex.position
      let next() :Ok =
      begin
        nextSym()
        while not(local.eof) andif {lex.sym() is symbol.wrong} do
          err(src.current() src.position())
          try
            src.next()
          else (* source.endOfInput *)
            local.eof := true
          end
          nextSym()
        end (* while *)
        local.sym := if local.eof then endOfInput else lex.sym() end
        ok
      end (* next *)
    end (* tuple *)
  end (* fun *)
end (* generator *)
```

Figure 3: Scanner generator function

5.4 Keywords

Keywords require a special handling because they are terminal symbols but normally also match the regular expression for identifiers. So the lexical analyser recognizes an identifier and the scanner has to look up a keyword lexicon whether the identifier found is a keyword.

These steps are taken to handle keywords:

1. Generate a keyword lexicon with the module *Lexicon*, e.g. when defining the symbols of the language. In the example this is done in the module *demoSym* by this line:

   ```
   let keywordLexicon = lexicon.new(nextsym())
   (* 'nextsym()' returns the first symbol that can be used for keywords *)
   ```

2. Insert the appropriate keywords in the lexicon, e.g. while defining the grammar of the language. This is done in the module *demoGram*:

   ```
   let letKW = lexicon.insert(demoSym.keywordLexicon "let")
   let inKW = lexicon.insert(demoSym.keywordLexicon "in")
   ```

   Save the keyword lexicon; see *demoGen*:

   ```
   dynamic.extern(
     writer.file("lexicon")
     dynamic.new(demoSym.keywordLexicon)
     dynamic.fast)
   ```

3. Define a new layer between scanner and parser which looks up the keyword lexicon if an identifier is found. Therefore the lexicon has to be loaded at runtime, i. e. in *demoMain*:

   ```
   let keywordLexicon =
     dynamic.be(
       :lexicon.T
       dynamic.intern(reader.file("lexicon") dynamic.fast))

   let scanner =
   tuple
     let sym() :DemoSym_T =
       if scanner.sym() is demoSym.id then
         try
           lexicon.lookupSymbol(keywordLexicon)(scanner.string())
         else
           demoSym.id
         end
       else
         scanner.sym()
       end
       let string = scanner.string
       let position = scanner.position
       let next = scanner.next
   end
   ```

5.5 Parser definition

Parsers are defined and generated by *Grammar*. Attributed grammars are used to define actions that are executed while a sentence of the language is parsed for correctness. Normally, these actions build an abstract syntax tree.

In the example two modules define the grammar and attached actions: *DemoGram* and *DemoExp*. Instead of building a syntax tree the parser binds identifiers to values and evaluates expressions. It also checks if all used identifiers have been declared and if the types are compatible.

Follow these instructions to build a parser:

1. Define the grammar with attached actions using *Grammar*. Currently only LL(1)-grammars can be used.

   Figure 4 shows an example for the definition of the grammar rule *factor* (see section 5.2 for the complete grammar). It is taken from *demoExp*. *grammar.alt* describes alternative clauses; note that all sub-clauses of the alternative clause have to take inherited attributes und deliver derived attributes of the same type, what is checked by the *P-Quest* compiler. *grammar.seq* describes sequence clauses, which have an attached action (see section 5.6). In this case, the action takes an inherited attribute and the derived attributes of the three parts, and simply hands through one derived attribute, while the others are dismissed.

```
(* - factor ::= '(' exp ')' | int | real | id *)

let factor =
  grammar.alt of

    grammar.seq3(leftparenS exp rightparenS
      fun(:bindings.T
           :Source_Position
         v :Value
           :Source_Position)
           :Value
       v)

    intS

    realS

    idS

  end
```

Figure 4: Grammar rule *factor*

2. Generate the parser by calling the method *newParser*. While the generation takes place it is checked whether the grammar is of the correct (LL(1)-)form.

   This is done in *demoGen*:

   let *parser* = *grammar.newParser(demoGram.top)*

3. The result is a parser function that takes a concrete scanner as parameter and checks the symbol stream of this scanner for correctness.

In this process there are two possibilities to save something:

1. Save the definition of the grammar to extend it later on:

   ```
   dynamic.extern(
     writer.file("grammar")
     dynamic.new(demoGram.top)
     dynamic.fast)
   ```

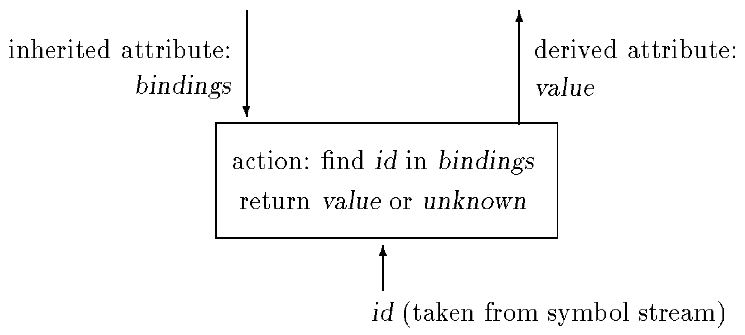
2. Save the generated parser function to use it in a compiler.

   ```
   dynamic.extern(
     writer.file("parser")
     dynamic.new(parser)
     dynamic.fast)
   ```

5.6 Action definition

Actions that for example build a syntax tree are defined along with the grammar that describes the language. Each rule (terminal or non-terminal) takes an inherited attribute, performs actions and delivers a derived attribute. Further information about attribute grammars can be found, for example, in [WG85, RT88, RT89].

Figure 6 shows an example taken from *demoExp* that describes the actions that have to be performed when an identifier is expected. See Figure 5 for a graphical representation of the example. The type of the inherited attribute is *bindings.T*, the type of the derived attribute is *bindings.Value*. The action has to transform the inherited to the derived attribute. If an identifier is found the value it has been bound to is looked up in the (inherited) bindings. If the identifier has been declared, the associated value is returned. Otherwise the value *unknown* is returned. If no identifier has been found also the value *unknown* is returned.



inherited attribute: *bindings*
derived attribute: *value*
action: find *id* in *bindings*
return *value* or *unknown*
id (taken from symbol stream)

Figure 5: Node of parse tree for *idS*

5.7 Generating the scanner and the parser

Scanner, parser, and lexicon are generated by *demoGen*. While the generation process takes place status information is reported. This can be suppressed by setting flags in *grammar* and *regToLex*.

The generated scanner, parser and lexicon are saved into the files *scanner*, *parser* and *lexicon*. Strictly speaking, two *functions* are saved, a scanner function that, given a source and an error report function, returns a scanner, and a parser function that, given a scanner and a error report/recovery function, parses a symbol stream.

```
let idS =
  grammar.terminal(
    :bindings.T                          (* – type of inherited attribute *)
    :bindings.Value                         (* – type of derived attribute *)
    demoSym.id                                 (* – symbol: 'identifier' *)
    fun(b :bindings.T                  (* – normal action: takes bindings *)
        s :Parser_ScannerState)              (* – and the scanner state *)
      :Bindings_Value                              (* – returning a value *)
      try                                    (* – install exception handler *)
        bindings.lookup(b s.string())    (* – lookup identifier in binding *)
      else                                                 (* – not found *)
        (* error report *)
        unknown                                        (* – value unknown *)
      end
    fun(b :bindings.T                              (* – action in error case *)
        s :Parser_ScannerState)          (* – (id expected, but not found) *)
      :Bindings_Value
    begin
      (* error report *)
      unknown
    end
  )
```

Figure 6: Action definition for *idS*

5.8 Running the demo

The main "compiler-interpreter" is *demoMain*. It defines the functions for error reporting and recovery, asks the user for input, opens the input source, loads the scanner and parser functions and the lexicon, generates the concrete scanner, and starts the parser.

References

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[Car90] L. Cardelli. The Quest Language and System (Tracking Draft). Digital systems research center, DEC SRC Palo Alto, 1990. (shipped as part of the Quest V.12 system distribution).

[Mat91] F. Matthes. P-Quest: Installation and User Manual. DBIS Tycoon Report 101-91, Fachbereich Informatik, Universität Hamburg, West Germany, October 1991.

[MS91] F. Matthes and J.W. Schmidt. Towards Database Application Systems: Types, Kinds and Other Open Invitations. In *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, volume 504 of *Lecture Notes in Computer Science*, April 1991.

[MSS91] F. Matthes, G. Schröder, and J.W. Schmidt. VAX Modula-2 User's Guide; VAX DBPL User's Guide. DBPL Memo 121-91, Fachbereich Informatik, Universität Hamburg, West Germany, December 1991.

[Nie91] P. Niebergall. Language-Sensitive Technology for Database Program Development. DBPL-Memo 108-91, Fachbereich Informatik, Universität Hamburg, West Germany, 1991.

[NMM92] C. Niederée, S. Müßig, and F. Matthes. P-Quest User Manual. DBIS Tycoon Report 102-92, Fachbereich Informatik, Universität Hamburg, West Germany, February 1992. (in German).

[RT88] T.W. Reps and T. Teitelbaum. *The Synthesizer Generator: A System For Constructing Language-Based Editors.* Texts and Monographs in Computer Science. Springer-Verlag, 1988.

[RT89] T.W. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual.* Texts and Monographs in Computer Science. Springer-Verlag, third edition, 1989.

[Sch91] Gerald Schröder. Studienarbeit: Die Standardisierung von Modula-2. Master's thesis, Fachbereich Informatik, Universität Hamburg, West Germany, November 1991.

[SM91] J.W. Schmidt and F. Matthes. Naming Schemes and Name Space Management in the DBPL Persistent Storage System. In *Proceedings of the Fourth International Workshop on Persistent Object Systems, Martha's Vineyard, Massachusetts.* Morgan Kaufmann Publishers, January 1991.

[WG85] W.M. Waite and G. Goos. *Compiler Construction.* Texts and monographs in computer science. Springer-Verlag, 1985.