Cmajor Parser Generator Tool
version 1.0

Seppo Laakko

September 25, 2014

Contents

Chapter 1

Introduction

Cmajor Parser Generator is a recursive descent parser generator tool for Cmajor. It is a source-to-source tool: it reads parser definition files (.parser) and generates corresponding Cmajor classes (.cm). When the generated classes are compiled and linked with **System.Text.Parsing** library, the result is a top down recursive descent kind of parser.

Chapter 2

Tutorial

In this tutorial we will implement a simple four function calculator using Cmajor Parser Generator.

2.1 Building a Desc Calculator

- Create a new Cmajor Console Application in Cmajor Development Environment and name it 'dc'.
- Add a new text file to your project and name it 'dc.parser'. The contents of the dc.parser is in listing 2.1. ([1].)

Listing 2.1: dc.parser

```
/******************
 * Desk Calculator *
 ******************/

grammar DcGrammar
{
    using stdlib.number;
    using stdlib.spaces;
    skip spaces;

    expr: double
        ::= term:t{ value = t; } ('+' term:pt{ value = value + pt; }
            | '-' term:mt{ value = value - mt; })*
        ;

    term: double
        ::= factor:f{ value = f; } ('*' factor:tf{ value = value * tf
            ; } | '/' factor:df{ value = value / df; })*
        ;

    factor: double
        ::= ('-' primary:mp){ value = -mp; }
        |   primary:p{ value = p; }
```

[1] The source code can be found in .../examples/cmparsergen/Examples/dc directory

```
          ;

    primary: double
          ::= '(' expr:e{ value = e; } ')'
          |    number:n{ value = n; }
          ;
}
```

The parser consists of a single *grammar* named DcGrammar. The DcGrammar uses two *external rules* from **stdlib** grammar that is available to all parser projects. The **stdlib.number** is a rule for parsing a numeric value and **stdlib.spaces** is a rule for skipping white space around more interesting tokens.

The grammar defines the **spaces** rule to be a *skip rule*.

The grammar contains four *rules*: **expr**, **term**, **factor** and **primary**. **expr** rule is the first so it is implicitly set as the *start rule* of the grammar. ([2])

After each rule there is a colon and type name (double). This is the type that each rule returns — it is the *synthesized attribute* of the rule.

The *body* of the rule is defined after a `::=` symbol that is pronounced “produces”. The body consists of characters enclosed in quotes, names of other rules (nonterminals) and metasymbols like parentheses, braces, vertical bars, and asterisks:

- A character enclosed in quotes matches the enclosed character.
- A nonterminal (name of another rule) acts like a procedure call in a program: it matches the contents of the referred rule. When there are more than one instances of the same nonterminal in a body of a rule, the nonterminals need to be renamed using a colon and an identifier after the name of the rule.
- Parentheses mean grouping.
- Braces enclose semantic actions. A semantic action is a Cmajor compound statement within the .parser file. It can do same things as Cmajor statements can do, but a few things are special: Keyword **value** means the resulting value (synthesized attribute) of the rule. By assigning in to it you set the resulting value of the rule. Results of the called rules (their synthesized attributes) can be referred by names of the rules or by their renaming identifiers.
- Sequential elements in the body of the rule mean matching elements from left to right. For example Foo ::= A 'X' B;, means when we are matching rule Foo, we first match rule A then character 'X' and then a rule B.
- A vertical bar separates alternatives. The parser tests each alternative from left to right in turn and stops as soon as one matches. If none of this rule's alternatives match, the next of the calling rule's altertives are tried, and so on.
- Asterisk means matching the preceding construct 0 or more times.

• Now add a new text file to your project and name it 'dc.pp' (.pp for "Parser Project File"). The contents of the dc.pp is in listing 2.2.

[2]the start rule can also be set using the **start** statement

Listing 2.2: dc.pp

```
project dc;

reference <StdLib.pl>;
source <dc.parser>;
```

- The *project* statement in a project file sets the name of the project.
- The *reference* statement adds a reference to a library file (.pl) to the libraries this project uses. By compiling the project with the **cmajorparsergen** utility automatically generates a library file for that project.

  StdLib.pl is a library file that contains the interface of the **stdlib** grammar.
- The *source* statement includes a (.parser) file in a project.

- Generate Cmajor source code using **cmparsergen** tool:
  - Open command prompt and change to 'dc' project directory.
  - Execute command `cmparsergen dc.pp` from there:

```
C:\Temp\dc>cmparsergen dc.pp
Cmajor parser generator version 1.0
Parsing project file 'C:/Temp/dc/dc.pp'...
Compiling project 'dc'...
Parsing library files...
< C:/Programming/cmajor++/system/ext/System.Text.Parsing/StdLib.pl
> C:/Programming/cmajor++/system/ext/System.Text.Parsing/StdLib.pl
Parsing source files...
> C:/Temp/dc/dc.parser
Linking...
Expanding code...
Generating source code...
=> C:/Temp/dc/dc.cm
Generating library file...
=> C:/Temp/dc/dc.pl
```

- Add the generated dc.cm to your project.
- Add a new Cmajor source file to your project and name it 'main.cm'. The contents of the main.cm is in listing 2.3:

Listing 2.3: main.cm

```
using System;

void main()
{
    try
    {
        DcGrammarPtr grammar = DcGrammar.Create();
        while (true)
        {
```

```
10                  Console.Out() << "> ";
11                  string line = Console.ReadLine();
12                  if (line == "exit")
13                  {
14                      break;
15                  }
16                  try
17                  {
18                      double value = grammar->Parse(line, 0, "");
19                      Console.Out() << "= " << value << endl();
20                  }
21                  catch (const Exception& ex)
22                  {
23                      Console.Out() << ex.Message() << endl();
24                  }
25              }
26          }
27          catch (const Exception& ex)
28          {
29              Console.Error() << ex.ToString() << endl();
30          }
31      }
```

- In line 7 we create an instance of the grammar. The grammar is automatically destroyed in the end of the program.
- In line 11 we read a line from terminal.
- In line 18 we parse the read expression using our grammar. The *Parse* function returns the value (the synthesized attribute) of the start rule.
- If the input is not syntactically valid expression, the parser throws an exception that we catch in line 21 and then print and error message.

• Add a reference to **System.Text.Parsing** library to the project by right clicking the **dc** project and enabling
Project References... / Add System Extension Library Reference... / System.Text.Parsing check box. Then build the project:

```
Cmajor release mode compiler version 0.9.0.0 (x64)
Parsing project file 'C:/Temp/dc/dc.cmp'...
Symbols  read from 'C:/Temp/dc/debug/llvm/dc.cmp.sym'
Building project 'dc' using debug configuration...
Parsing library files...
> C:/Programming/cmajor++/system/debug/llvm/system.cml
> C:/Programming/cmajor++/system/debug/llvm/os.cml
> C:/Programming/cmajor++/system/debug/llvm/support.cml
> C:/Programming/cmajor++/system/ext/System.Text.Parsing/debug/llvm/System.Text.Parsi
> C:/Programming/cmajor++/system/ext/System.Text.Parsing.CmObjectModel/debug/llvm/Sys
Loading libraries...
<- os
<- support
<- system
<- System.Text.Parsing.CmObjectModel
<- System.Text.Parsing
Parsing sources...
> C:/Temp/dc/dc.cm
> C:/Temp/dc/main.cm
Type checking signatures...
PT> C:/Programming/cmajor++/system/ptr.cm
PT> C:/Programming/cmajor++/system/stack.cm
PT> C:/Programming/cmajor++/system/list.cm
PT> C:/Programming/cmajor++/system/iterator.cm
PT> C:/Programming/cmajor++/system/set.cm
PT> C:/Programming/cmajor++/system/function.cm
PT> C:/Programming/cmajor++/system/rbtree.cm
PT> C:/Programming/cmajor++/system/pair.cm
PT> C:/Programming/cmajor++/system/map.cm
Instantiating 411 enqueued functions and generating intermediate code...
PT> C:/Programming/cmajor++/system/concept.cm
Compiling...
> C:/Temp/dc/__interface__.cm
> C:/Temp/dc/dc.cm
PT> C:/Programming/cmajor++/system/rvalue.cm
PT> C:/Programming/cmajor++/system/ext/System.Text.Parsing/Object.cm
PT> C:/Programming/cmajor++/system/algorithm.cm
> C:/Temp/dc/main.cm
PT> C:/Programming/cmajor++/system/stream.cm
PT> C:/Programming/cmajor++/system/eh.cm
Generating exception table...
=> C:/Temp/dc/__exception_table__.cm
Generating library file...
=> C:/Temp/dc/debug/llvm/dc.cml
Generating exception type file...
=> C:/Temp/dc/debug/llvm/dc.exc
Writing dependency information...
=> C:/Temp/dc/debug/llvm/dc.dep
Archiving...
=> C:/Temp/dc/debug/llvm/dc.cma
Linking...
> C:/Programming/cmajor++/system/debug/llvm/cmsys.o
> C:/Programming/cmajor++/system/debug/llvm/dbgheap.o
> C:/Programming/cmajor++/system/debug/llvm/mutextbl.o
> C:/Programming/cmajor++/system/debug/llvm/threadtbl.o
> C:/Temp/dc/debug/llvm/dc.cma
> C:/Programming/cmajor++/system/ext/System.Text.Parsing/debug/llvm/System.Text.Parsi
> C:/Programming/cmajor++/system/ext/System.Text.Parsing.CmObjectModel/debug/llvm/Sys
> C:/Programming/cmajor++/system/debug/llvm/system.cma
> C:/Programming/cmajor++/system/debug/llvm/support.cma
> C:/Programming/cmajor++/system/debug/llvm/os.cma
=> C:/Temp/dc/debug/llvm/dc.exe
1 project built, 0 projects up-to-date.
21 seconds
```

- Test the dc program for example by pressing CTRL+5:

```
Running 'C:/Temp/dc/debug/llvm/dc.exe':

> 1 + (2 * 3)
= 7
> foo
Parsing file '' failed at line 1:
<expr> expected:
foo
^
> exit

Process returned exit code 0.
```

- Hurray! We got it working :)

Chapter 3

Syntax

3.1 Notation

The structure of the Cmajor Parser Generator input files is presented in this text using grammars that are in extended Backus-Naur form. A grammar consists of

1. *Terminal* symbols that are the elementary symbols of the language generated by the grammar. Terminals are presented like `this` in a type-writer font.

2. *Nonterminal* symbols also called syntactic variables that represent sets of strings of terminal symbols. Nonterminals are presented like $\langle this \rangle$ in italic font within angle brackets.

3. *Operators* ( ), |, −, *, +, ?, [ ], and \ that operate on strings of terminals and nonterminals. If an operator character is ment to be a terminal symbol, it is quoted like this: '`*`'. The meaning of each operator appears in table 3.1.

4. *Productions* each of which consists of a nonterminal symbol called the *head* of the production, symbol ::= (pronounced "produces"), and a sequence of terminals, nonterminals and operators collectively called the *body* of the production.

Table 3.1: Grammar Operators

Expression	**Meaning**
$\alpha \mid \beta$	α or β
$\alpha - \beta$	α but not β
$\alpha*$	zero or more α's
$\alpha+$	one or more α's
$\alpha?$	zero or one α's
$(\alpha\ \beta)$	grouping of α and β together
`[a-z]`	a terminal character in range a . . . z
`[^\r\na-c]`	any terminal character except a carriage return, a newline or a character in range a . . . c

3.2 Parser File Format

⟨*parser-file*⟩ ::= ⟨*namespace-content*⟩

⟨*namespace-content*⟩ ::= (⟨*using-declarations*⟩? (⟨*grammar*⟩ | ⟨*namespace*⟩)*

A Parser File (.parser) contains using-declarations, namespaces and grammars.

3.2.1 Using Declarations

⟨*using-declarations*⟩ ::= ⟨*using-declaration*⟩+

⟨*using-declaration*⟩ ::= ⟨*namespace-import*⟩ | ⟨*alias*⟩

⟨*namespace-import*⟩ ::= `using` ⟨*qualified-id*⟩ ';'

⟨*alias*⟩ ::= `using` ⟨*identifier*⟩ '=' ⟨*qualified-id*⟩ ';'

⟨*identifier*⟩ ::= [a-zA-Z_] [a-zA-Z_0-9]*

⟨*qualified-id*⟩ ::= ⟨*identifier*⟩ ('.' ⟨*identifier*⟩)*

Using declarations are placed in the beginning of generated Cmajor source file.

3.2.2 Namespaces

⟨*namespace*⟩ ::= `namespace` ⟨*qualified-id*⟩ '{' ⟨*namespace-content*⟩ '}'

A namespace can contain using-declarations, grammars and namespaces.

3.2.3 Grammars

⟨*grammar*⟩ ::= `grammar` ⟨*parser-identifier*⟩ '{' ⟨*grammar-content*⟩ '}'

⟨*parser-identifier*⟩ ::= ⟨*identifier*⟩ - ⟨*parser-keyword*⟩

⟨*parser-keyword*⟩ ::= `using` | `grammar` | `start` | `skip` | `token` | `keyword` | `keyword_list` |
`empty` | `space` | `anychar` | `letter` | `digit` | `hexdigit` | `punctuation` | `var`

⟨*qualified-soul-id*⟩ ::= ⟨*soul-identifier*⟩ ('.' ⟨*soul-identifier*⟩)*

⟨*grammar-content*⟩ ::= (⟨*start-clause*⟩ | ⟨*skip-clause*⟩ | ⟨*rule-link*⟩ | ⟨*rule*⟩)*

⟨*start-clause*⟩ ::= `start` ⟨*soul-identifier*⟩ ';'

⟨*skip-clause*⟩ ::= `skip` ⟨*soul-identifier*⟩ ';'

⟨*rule-link*⟩ ::= `using` ⟨*soul-identifier*⟩ '=' ⟨*qualified-soul-id*⟩ ';'
| `using` ⟨*qualified-soul-id*⟩ ';'

Grammar is the basic building block of Cmajor Parser Generator. Grammar consists of rules, links to rules in other grammars, and start and skip clauses.

The start clause defines the starting rule of the grammar. If the grammar has no start clause, the first rule of the grammar is the starting rule.

The skip clause defines the skip rule of the grammar. The skip rule is used to skip unimportant tokens such as white space and comments during parsing.

A rule link brings a rule from another grammar into the scope of a grammar. The first form gives the rule a new name. It is used when the name of the rule would conflict a rule name in the target grammar. The second form retains the original name of the rule.

Listing 3.1 gives an example of elements in a Parser File.

Listing 3.1: Example1.parser

```
1  namespace N.S
2  {
3      using Foo.Bar;
4      using Baz = Foo.Bar.Baz;
5
6      grammar A
7      {
8          start S;
9          skip Spaces;
10
11         using B.Alpha;
12         using BBeta = B.Beta;
13
14         Spaces      ::= [ \t\r\n]+
15                     ;
16
17         S           ::= Alpha | BBeta | Gamma
18                     ;
19
20         Beta        ::= "beta"
21                     ;
22
23         Gamma       ::= "gamma"
24                     ;
25
26     }
27     grammar B
28     {
29         Alpha       ::= "alpha"
30                     ;
31
32         Beta        ::= "beta"
33                     ;
34     }
35 }
```

Grammars and using-declarations are enclosed inside a namespace N.S (line 1).

A using declarations in line 3 and 4 are placed to the generated .cm file as is.

Rule S is set as the starting rule of grammar A (line 8). Rule Spaces is set as the skipping rule of grammar A (line 9).

In line 11 the rule Alpha in grammar B is brought to grammar A's scope. In line 12 the rule Beta in grammar B is brought to grammar A's scope under an alias name BBeta, because it would otherwise conflict the rule 'Beta' in grammar A.

3.2.4 Rules

⟨*rule*⟩ ::= ⟨*rule-header*⟩ `::=` ⟨*rule-body*⟩

⟨*rule-header*⟩ ::= ⟨*soul-identifier*⟩ ⟨*signature*⟩

⟨*rule-body*⟩ ::= ⟨*alternative*⟩ ';'

⟨*signature*⟩ ::= ⟨*parameter-list*⟩? ⟨*return-type*⟩?

⟨*parameter-list*⟩ ::= '(' (⟨*variable*⟩ | ⟨*parameter*⟩) (',' (⟨*variable*⟩ | ⟨*parameter*⟩))* ')'

⟨*variable*⟩ ::= `var` ⟨*cmajor-type-expression*⟩ ⟨*cmajor-identifier*⟩

⟨*parameter*⟩ ::= ⟨*cmajor-type-expression*⟩ ⟨*cmajor-identifier*⟩

⟨*return-type*⟩ ::= ':' ⟨*cmajor-type-expression*⟩

A rule consists of rule header and rule body separated by the "::=" symbol. A rule header defines the name of the rule and possibly a list of parameters and variables for the rule. The parameters are also called *inherited attributes* of the rule. A rule can also return a value. The return value is also called the *synthesized attribute* of the rule.

A Cmajor type expression names a Cmajor type. A Cmajor identifier names a Cmajor variable or parameter (see [2]).

A rule body consists of a combination of composite, primary and primitive parsers.

Listing 3.2 gives an example of a rule that has an inherited attribute, a local variable and it returns a value.

Listing 3.2: Example2.parser

```
grammar A
{
    Id(SymbolTablePtr s, var bool upper): IdPtr
            ::= token(([a-z_] | [A-Z]{ upper = true; })[a-zA-Z0-9_]*){
                value = IdPtr(new Id(match, upper)); s->Add(value); }
}
```

Rule 'Id' has an inherited attribute 's' and a local variable 'upper'. It returns a value of type 'IdPtr'.

Note: The Cmajor types used for inherited and synthesized attributes and local variables must be default constructible. Thus reference types cannot be used, but pointer types are okay, because a pointer is default constructible (null).

3.2.5 Composites

3.2.5.1 Alternatives

⟨*alternative*⟩ ::= ⟨*sequence*⟩ ('|' ⟨*sequence*⟩)*

When parsing input, each alternative is tried in order. If the input matches an alternative, other alternatives coming after it will not be tried.

In listing 3.3 a rule Fruit accepts alternative strings “apple”, “orange” and “banana”. The matching string is returned as the synthesized attribute of the rule.

Listing 3.3: Example3.parser

```
grammar A
{
    Fruit: string ::=       (   "apple"
                            |   "orange"
                            |   "banana"
                            )
                            {
                                value = match;
                            }
                            ;
}
```

3.2.5.2 Sequence

⟨*sequence*⟩ ::= ⟨*difference*⟩+

Elements in a sequence are matched from left to right.

Listing 3.4 gives an example of a sequence: An if-statement is a sequence of keyword “if”, a left parenthesis, an expression, a right parenthesis and a statement. Then can come a sequence of keyword “else” and another statement. The if-statement ends with a semicolon.

Listing 3.4: Example4.parser

```
grammar StatementGrammar
{
    IfStatement: StatementPtr
            ::= (   keyword("if") '(' Expr ')' Statement:ifs
                    (keyword("else") Statement:elses)? ';'
                )
                {
                    value = StatementPtr(new IfStat(Expr, ifs, elses));
                }
            ;
}
```

3.2.5.3 Difference

⟨*difference*⟩ ::= ⟨*exclusive-or*⟩ ('-' ⟨*exclusive-or*⟩)*

An input string matches a difference A - B if it matches A but not B.

Listing 3.5 gives an example of a difference: An ‘identifier’ is a string that matches rule ‘id_char_sequence’ but not rule ‘kwd’.

Listing 3.5: Example5.parser

```
grammar Id
{
    id_char_sequence    ::= token([a-zA-Z_][a-zA-Z_0-9]*)
                        ;

    kwd                 ::= keyword_list(id_char_sequence,
                            ["bool", "double", "int", "signed"])
                        ;

    identifier          ::= id_char_sequence - kwd
                        ;
}
```

3.2.5.4 Exclusive-Or

⟨*exclusive-or*⟩ ::= ⟨*intersection*⟩ ('^' ⟨*intersection*⟩)*

A string matches `A ^ B` if it matches either A or B but not both.

3.2.5.5 Intersection

⟨*intersection*⟩ ::= ⟨*list*⟩ ('&' ⟨*list*⟩)*

A string matches A & B if it matches both A and B.

3.2.5.6 List

⟨*list*⟩ ::= ⟨*postfix*⟩ ('%' ⟨*postfix*⟩)?

A list A % B is a short-hand notation for construct A (B A)*, that is: a sequence of A's separated by B's.

Listing 3.6 gives an example of a list construct: a parameter list is a sequence of parameters separated by commas.

Listing 3.6: Example6.parser

```
grammar FunctionDecl
{
    FunctionDeclaration ::= Identifier '(' ParameterList ')' ';'
                        ;

    ParameterList       ::= Parameter % ','
                        ;

    Parameter           ::= TypeExpr Identifier
                        ;
}
```

3.2.5.7 Postfix

⟨*postfix*⟩ ::= ⟨*primary*⟩ ('*' | '+' | '?')?

An expression A* means zero or more A's, expression A+ means one or more A's, and A? means zero or one A's.

Listing 3.7 gives an example of postfix operators: A digit sequence is sequence of one or more digits (line 3), an integer is a digit sequence with an optional sign prefix (zero or one sign characters) (line 9), an identifier is a letter followed by zero or more letters and digits (line 12).

Listing 3.7: Example7.parser

```
1  grammar Lexical
2  {
3      digit_sequence  ::= token(digit+)
4                      ;
5
6      sign            ::= '+' | '-'
7                      ;
8
9      integer         ::= token(sign? digit_sequence)
10                     ;
11
12     id              ::= token(letter (letter | digit)*)
13                     ;
14 }
```

3.2.6 Primaries

⟨*primary*⟩ ::= (⟨*rule-call*⟩ | ⟨*primitive*⟩ | ⟨*grouping*⟩ | ⟨*token*⟩) ⟨*expectation*⟩? ⟨*action*⟩?

A primary is rule-call, primitive, grouping or a token followed by an optional expectation and an optional action.

3.2.6.1 Rule Call

⟨*rule-call*⟩ ::= (⟨*nonterminal*⟩ '(' ⟨*expression-list*⟩ ')' | ⟨*nonterminal*⟩) ⟨*alias*⟩?

⟨*nonterminal*⟩ ::= ⟨*soul-identifier*⟩

⟨*alias*⟩ ::= ':' ⟨*soul-identifier*⟩

A call of a rule can take two forms. If the rule has inherited attributes, the rule is called by passing it a list of arguments enclosed in parentheses. If the rule has no inherited attributes, the rule is called without an argument list. In the first case there should be no space between the name of the rule and the left parenthesis.

In both cases the name of the rule (nonterminal) can be given an alias name. The alias name is be mandatory if the called rule has inherited or synthesized attributes and it is called many times from the body of some rule. In that case the alias names should be distinct.

For example, if the rule A is called many times from rule R in grammar G and not given alias names, it results an error "object 'G.R.A' already exists (detected in scope 'G.R')".

Listing 3.8 gives an example of a rule call of the second form. Rule ‘Expression’ has been given alias names because it returns a value and the returned value is needed in the semantic action.

Listing 3.8: Example8.parser

```
grammar Statements
{
    ForStatement: StatementPtr
                    ::= (keyword("for")
                        '('
                            ForInitStmt
                            Expression:condition? ';'
                            Expression:increment?
                        ')'
                        Statement
                    )
                    {
                        value = StatementPtr(new ForStat(ForInitStmt,
                            condition, increment, Statement));
                    }
                    ;

    ForInitStmt: StatementPtr
                    ::= ...

    Expression: ObjectPtr
                    ::= ...
}
```

3.2.6.2 Grouping

⟨*grouping*⟩ ::= ’(’ ⟨*alternative*⟩ ’)’

The precedence of each grammar operator from the weakests to the strongest is defined by this composite grammar. The ’|’-operator has least precedence, then comes sequence, difference ’-’, and so on. Thus grammar expression $AB - C|D$ means $(A(B - C))|D$. By using parentheses, the grammar operator evaluation order can be changed. For example, the previous expression could be grouped as $(AB) - (C|D)$.

3.2.6.3 Token

⟨*token*⟩ ::= `token` ’(’ ⟨*alternative*⟩ ’)’

The token construct inhibits the effect of the current skipping rule. Typically the skipping of white space must be inhibited when parsing a lexical token such as identifier or number.

Listing 3.9 gives an example of using the token construct when parsing an identifier. If the skipping is not inhibited, a string like ”a bc d” would be accepted as an identifier.

Listing 3.9: Example9.parser

```
grammar Id
{
    identifier ::= token(letter (letter | digit)*)
               ;
}
```

3.2.6.4 Expectation

⟨*expectation*⟩ ::= '!'

A recursive descent parser works by matching rules from left to right and from top to bottom. When a construct does not match, the input must be backtracked and a next alternative tried. Sometimes the backtracking behaviour is not wanted. For example when parsing programming language if-statement, after seeing the **if** keyword, there is no other possible alternatives: it must definitely be an if-statement.

The expectation construct causes an ExpectationFailure exception to be thrown if the input does not match the construct preceding the expectation symbol.

Listing 3.10 gives an example of applying the expectation semantics to parsing an if-statement.

Listing 3.10: Example10.parser

```
grammar StatementGrammar
{
    IfStatement: StatementPtr
            ::= (   keyword("if") '('! Expr! ')'! Statement:ifs!
                    (keyword("else") Statement:elses!)? ';'!
                )
                {
                    value = StatementPtr(new IfStat(Expr, ifs, elses));
                }
            ;
}
```

3.2.6.5 Actions

⟨*action*⟩ ::= ⟨*compound-statement*⟩ ('/' ⟨*compound-statement*⟩)?

A grammar construct can be associated a semantic action that is fired when the input matches the construct preceding the action. By using the '/' operator, another semantic action (here called *failure action* can also be fired when the input did not match the construct.

A semantic action is a Cmajor compound statement (see [2]). The Cmajor parsing library (System.Text.Parsing) calls the semantic action with the following parameters:

```
void Action(const string& match, const string& content,
    const Position& position, const string& fileName, bool& pass);
```

- `match` contains the matching input.
- `content` contains the whole input.
- `Position` represents the current input position. It is defined as the following class:

```
class Position
{
    int index;
    int file;
    int line;
    int column;
};
```

where

  - `index` is the index of the first matching character in the whole input string.
  - `file` is the number of the file passed to the `Parse` member function when calling the parser.
  - `line` is the line number of the first matching character (starting from 1).
  - `column` is the column number of the first matching character (starting from 1).

- `fileName` is the name of the file passed to the `Parse` member function when calling the parser.
- `pass` is a Boolean reference parameter. If the semantic action sets `pass` to false, the parser behaves like the input did not match the construct preceding the action.

In addition to these parameters, the inherited attributes, local variables, synthesized attributes of the so far called rules and synthesized attribute of the rule that contains the semantic action are visible to the action by their name (or alias name in case of nonterminals). The synthesized attribute of the rule that contains the semantic action is visible under the name `value`.

Listing 3.11 gives an example how these can be used in an semantic action that constructs a syntax tree node 'ForStat' for a for-statement.

Listing 3.11: Example11.parser

```
grammar Statements
{
    ForStatement: StatementPtr
                        ::= (keyword("for")
                            '('
                                ForInitStmt
                                Expression:condition? ';'
                                Expression:increment?
                            ')'
                            Statement
                        )
                        {
                            value = StatementPtr(new ForStat(ForInitStmt,
                                condition, increment, Statement));
                        }
                        ;

    ForInitStmt: StatementPtr
                        ::= ...

    Expression: ObjectPtr
                        ::= ...
}
```

A failure action has no parameters:

```
void FailureAction();
```

3.2.7 Primitives

⟨*primitive*⟩ ::= ⟨*char*⟩ | ⟨*string*⟩ | ⟨*charset*⟩ | ⟨*keyword*⟩ | ⟨*keyword-list*⟩ | ⟨*empty*⟩ | ⟨*space*⟩ | ⟨*anychar*⟩ | ⟨*letter*⟩ | ⟨*digit*⟩ | ⟨*hexdigit*⟩ | ⟨*punctuation*⟩

3.2.7.1 Character

⟨*char*⟩ ::= `'` (`[^\\\r\n]` | ⟨*escape*⟩) `'`

⟨*escape*⟩ ::= \ [xX] hex | [dD] dec | `[^dDxX]`

⟨*hex*⟩ ::= [0-9a-fA-F]+

⟨*dec*⟩ ::= [0-9]+

A character primitive consists of a character or escape enclosed in apostrophes. It matches a single input character.

An escape consists of a backslash followed by

- `x` or `X` and an ASCII code in hexadecimal.
- `d` or `D` and an ASCII code in decimal.
- simple escape character a, b, f, n, r, t, v, 0 with the same meaning as is "C" family of languages.
- other character meaning the character itself.

3.2.7.2 String

⟨*string*⟩ ::= `"` (`[^"\\\r\n]`+ | ⟨*escape*⟩)* `"`

A string primitive consists of string of characters and escapes enclosed in quotes. It matches an input string.

3.2.7.3 Character Set

⟨*charset*⟩ ::= '[' ^? ⟨*charset-range*⟩* ']'

⟨*charset-range*⟩ ::= ⟨*charset-char*⟩ ('-' ⟨*charset-char*⟩)?

⟨*charset-char*⟩ ::= `[^\\\]]` | ⟨*escape*⟩

A character set primitive consists of a set of characters, character ranges and escapes. It matches a single character that is in the set. If the set is prefixed by ^ character, it matches a single character that is not in the set.

3.2.7.4 Keyword

⟨*keyword*⟩ ::= `keyword` ’(’ ⟨*keyword-body*⟩ ’)’

⟨*keyword-body*⟩ ::= ⟨*string*⟩ (’,’ ⟨*qualified-id*⟩)?

A keyword primitive consists of a keyword **keyword** followed by keyword body enclosed in parentheses. A keyword body is a single string or a string and a name of a continuation rule separated by a comma. In the first case the continuation rule is set as (*letter*|*digit*|_|.)+.

A keyword primitive differs from a string primitive in that it cannot continue with a string that matches the continuation rule. Thus string ”classified” matches string primitive ”class”, but does not match keyword primitive **keyword(”class”)**.

3.2.7.5 Keyword List

⟨*keyword-list*⟩ ::= `keyword_list` ’(’ ⟨*keyword-list-body*⟩ ’)’

⟨*keyword-list-body*⟩ ::= ⟨*qualified-id*⟩ ’,’ ⟨*string-array*⟩

⟨*string-array*⟩ ::= ’[’ ⟨*string*⟩ (’,’ ⟨*string*⟩)* ’]’

A keyword list primitive consists of the keyword **keyword_list** followed by the keyword list body enclosed in parentheses. A keyword list body consists of a selector rule, a comma and an array of keywords enclosed in brackets.

An input string that matches both the selector rule and one of the enlisted keywords matches the whole keyword list primitive.

3.2.7.6 Empty

⟨*empty*⟩ ::= `empty`

An empty primitive matches anything without consuming any input.

3.2.7.7 Character Classification Primitives

⟨*space*⟩ ::= `space`

⟨*anychar*⟩ ::= `anychar`

⟨*letter*⟩ ::= `letter`

⟨*digit*⟩ ::= `digit`

⟨*hexdigit*⟩ ::= `hexdigit`

⟨*punctuation*⟩ ::= `punctuation`

A **space** primitive matches single character that is a white space character according to the Cmajor `IsSpace` function.

A **anychar** primitive matches any single input character.

A **letter** primitive matches single character that is an alphabetic character according to the Cmajor `IsAlpha` function.

A **digit** primitive matches single character that is an decimal digit character according to the Cmajor `IsDigit` function.

A **hexdigit** primitive matches single character that is a hexadecimal digit character according to the Cmajor `IsHexDigit` function.

A **punctuation** primitive matches single character that is a punctuation character according to the Cmajor `IsPunctuation` function.

3.3 Project File Format

⟨*project-file*⟩ ::= `project` ⟨*qualified-id*⟩ ';' ⟨*project-file-content*⟩

⟨*project-file-content*⟩ ::= (⟨*source*⟩ | ⟨*reference*⟩)*

⟨*source*⟩ ::= `source` ⟨*file-path*⟩ ';'

⟨*reference*⟩ ::= `reference` ⟨*file-path*⟩ ';'

⟨*file-path*⟩ ::= `< [^>\r\n]+ >`

A Parser Project File (.pp) contains the name of the project and paths to Parser source (.parser) and library (.pl) files.

In listings 3.12 and 3.13 is an example of parser project that uses the Cmajor Parser Generator Standard Library (see appendix A):

Listing 3.12: Example.pp

```
project Example;
reference <StdLib.pl>;
source <Example.parser>;
```

Listing 3.13: Example.parser

```
grammar Example
{
    using stdlib.spaces_and_comments;
    skip spaces_and_comments;

    using stdlib.int;

    foo: int      ::=   int:i{ value = i; }
                  ;
}
```

Typically each grammar is in its own source file, although this is not enforced.

When the **cmparsergen** tool processes the project file it first reads the referenced library files (*.pl) and source files (*.parser) and places each object into its proper scope. It then links the grammars, and performs code expansion. After that it generates source code, and finally generates a library file (.pl) than can be referenced in other projects.

In listings 3.14 and 3.15 is an example of parser project that uses the previously generated Example parser project.

Listing 3.14: RefExample.pp

```
project RefExample;
reference <../Example/Example.pl>;
reference <StdLib.pl>;
source <RefExample.parser>;
```

Listing 3.15: RefExample.parser

```
grammar RefExample
{
    using Example.foo;
    using stdlib.spaces_and_comments;
    skip spaces_and_comments;

    ref: int      ::=   foo{ value = foo; }
                  ;
}
```

Appendix A

Cmajor Parser Generator Standard Library

The source code can also be found in `.../system/ext/System.Text.Parsing/StdLib.parser`.

```
/* ========================================================================
    Copyright (c) 2012-2014 Seppo Laakko
    http://sourceforge.net/projects/cmajor/

    Distributed under the GNU General Public License, version 3 (GPLv3).
    (See accompanying LICENSE.txt or http://www.gnu.org/licenses/gpl.html
        )

 ========================================================================
    */

// Copyright (c) 1994
// Hewlett-Packard Company
// Copyright (c) 1996
// Silicon Graphics Computer Systems, Inc.
// Copyright (c) 2009 Alexander Stepanov and Paul McJones

namespace System.Text.Parsing
{
    grammar stdlib
    {
        spaces                                  ::= space+
                                                ;

        newline                                 ::= "\r\n" | "\n" | "\r"
                                                ;

        comment                                 ::= line_comment |
            block_comment
                                                ;

        line_comment                            ::= "//" [^\r\n]* newline
```

```
                                          ;

        block_comment                     ::= "/*" (string | char | (
            anychar - "*/"))* "*/"
                                          ;

        spaces_and_comments               ::= (space | comment)+
                                          ;

        digit_sequence                    ::= token(digit+)
                                          ;

        sign                              ::= '+' | '-'
                                          ;

        int: int                          ::= token(sign?
            digit_sequence)
                                          {
                                              value = ParseInt(match);
                                          }
                                          ;

        uint: uint                        ::= digit_sequence
                                          {
                                              value = ParseUInt(match);
                                          }
                                          ;

        hex: ulong                        ::= token(hexdigit+)
                                          {
                                              value = ParseHex(match);
                                          }
                                          ;

        hex_literal: ulong                ::= token(("0x" | "0X") hex!)
            { value = hex; }
                                          ;

        octaldigit: int                   ::= [0-7]{ value = cast<int>(
            match[0]) - cast<int>('0'); }
                                          ;

        real: double                      ::= token(sign?
            fractional_real)
                                          {
                                              value = ParseDouble(match
                                                  );
                                          }
                                          ;

        ureal: double                     ::= fractional_real
                                          {
```

```
                                                    value = ParseDouble(match);
                                                }
                                                ;

        fractional_real                         ::= token(digit_sequence? '.' digit_sequence)
                                                |   token(digit_sequence '.')
                                                ;

        number: double                          ::= real:r{ value = r; } | int:i{ value = i; }
                                                ;

        bool: bool                              ::= keyword("true"){ value = true; } | keyword("false"){ value = false; }
                                                ;

        identifier: string                      ::= token((letter | '_') (letter | digit | '_')*)
                                                {
                                                    value = match;
                                                }
                                                ;

        qualified_id: string                    ::= token(identifier:first ('.' identifier:rest)*)
                                                {
                                                    value = match;
                                                }
                                                ;

        escape: char                            ::= token('\\'
                                                (
                                                    [xX] hex:x{ value = cast<char>(x); }
                                                |   (octaldigit:o1 octaldigit:o2 octaldigit:o3){ value = cast<char>(64 * o1 + 8 * o2 + o3); }
                                                |   [dD] uint:decimalEscape{ value = cast<char>(decimalEscape); }
                                                |   [^dDxX]
                                                {
                                                    char c = match[0];
                                                    switch (c)
                                                    {
                                                        case 'a': value = '\a'; break;
                                                        case 'b': value = '\b'; break;
                                                        case 'f': value = '\f'; break;
                                                        case 'n': value = '\n'; break;
                                                        case 'r': value = '\r'; break;
                                                        case 't': value = '\t'; break;
                                                        case 'v': value = '\v'; break;
                                                        case '0': value = '\0'; break;
                                                        default:  value = c; break;
                                                    }
                                                }
                                                )
                                                )
                                                ;

        char: char                          ::= token('\''
                                                (   [^\\\r\n]{ value = match[0]; }
                                                |   escape{ value = escape; }
                                                )   '\''!)
                                                ;

        string: string                      ::= token('"'
                                                    (
                                                        ([^"\\\r\n]+){ value.Append(match); }
                                                    |   escape{ value.Append(escape); }
                                                    )*
                                                '"'!)
                                                ;
        }
    }
```

Appendix B

Grammar Debugging

Grammars can be debugged by calling the grammar's SetLog() member function before parsing starts. SetLog accepts a pointer to System.IO.OutputStream. See listing B.1.

Listing B.1: Calculator.cm

```
using Calculator;
using System;

int main()
{
    try
    {
        CalculatorGrammarPtr calculator = CalculatorGrammar.Create();
        calculator->SetLog(&Console.Out());
        Console.Out() << "> ";
        string line = Console.ReadLine();
        while (line != "exit")
        {
            try
            {
                double result = calculator->Parse(line, 0, "");
                Console.Out() << "= " << result << endl();
            }
            catch (const Exception& ex)
            {
                Console.Error() << ex.Message() << endl();
            }
            Console.Out() << "> ";
            line = Console.ReadLine();
        }
    }
    catch (const Exception& ex)
    {
        Console.Error() << ex.Message() << endl();
        return 1;
    }
    return 0;
}
```

When logging is enabled the grammar generates XML output that shows parsing progress:

```
> 1+2
<parse>
<spaces>
 <try>1+2</try>
 </fail>
</spaces>
<Expr>
 <try>1+2</try>
  <Term>
   <try>1+2</try>
    <Factor>
     <try>1+2</try>
      <number>
       <try>1+2</try>
        <real>
         <try>1+2</try>
          <sign>
           <try>1+2</try>
           </fail>
          </sign>
          <fractional_real>
           <try>1+2</try>
            <digit_sequence>
             <try>1+2</try>
             <success>1</success>
            </digit_sequence>
            <digit_sequence>
             <try>1+2</try>
             <success>1</success>
            </digit_sequence>
           </fail>
          </fractional_real>
          <exponent_real>
           <try>1+2</try>
            <digit_sequence>
             <try>1+2</try>
             <success>1</success>
            </digit_sequence>
            <exponent_part>
             <try>+2</try>
             </fail>
            </exponent_part>
           </fail>
          </exponent_real>
         </fail>
      </real>
      <int>
       <try>1+2</try>
        <sign>
         <try>1+2</try>
         </fail>
        </sign>
        <digit_sequence>
         <try>1+2</try>
         <success>1</success>
        </digit_sequence>
       <success>1</success>
      </int>
     <success>1</success>
    </number>
   <success>1</success>
  </Factor>
 <success>1</success>
</Term>
<Term>
 <try>2</try>
  <Factor>
   <try>2</try>
    <number>
     <try>2</try>
      <real>
       <try>2</try>
        <sign>
         <try>2</try>
         </fail>
        </sign>
        <fractional_real>
         <try>2</try>
          <digit_sequence>
           <try>2</try>
           <success>2</success>
          </digit_sequence>
          <digit_sequence>
           <try>2</try>
           <success>2</success>
          </digit_sequence>
         </fail>
        </fractional_real>
        <exponent_real>
         <try>2</try>
          <digit_sequence>
           <try>2</try>
```

```
             <success>2</success>
            </digit_sequence>
            <exponent_part>
             <try></try>
             </fail>
            </exponent_part>
           </fail>
          </exponent_real>
         </fail>
        </real>
        <int>
         <try>2</try>
          <sign>
           <try>2</try>
           </fail>
          </sign>
          <digit_sequence>
           <try>2</try>
           <success>2</success>
          </digit_sequence>
         <success>2</success>
        </int>
       <success>2</success>
      </number>
     <success>2</success>
    </Factor>
   <success>2</success>
  </Term>
 <success>1+2</success>
</Expr>
<spaces>
 <try></try>
 </fail>
</spaces>
</parse>
= 3
>
```

Bibliography

[1] Aho, A. V., M. S. Lam, R. Sethi, and J. D. Ullman: Compilers: Principles, Techniques, & Tools. Second Edition. Addison-Wesley, 2007.

[2] Seppo Laakko: Cmajor Programming Language Specification. Version 2.0. *http://sourceforge.net/projects/cmajor/files/spec/LanguageSpec.pdf/download*