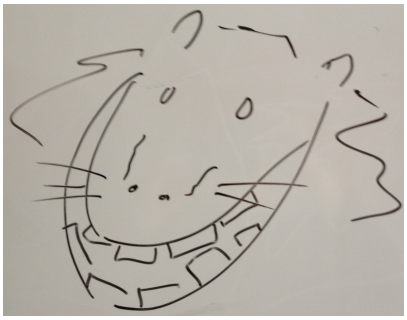
Learn You a gocc for Great Good

or

How to save the world by using compiler theory

2013-09-03



Contents

1 Introduction

gocc is a compiler kit which generates lexers, parsers and stand-alone DFAs from an EBNF file.

gocc lexers are deterministic finite state automata (DFA), which recognise regular languages.

gocc parsers are pushdown automata (PDA), which recognise LR-1 languages. LR-1 is the set of languages, which can be parsed deterministically. Some context free grammars (CFG) are outside LR-1, because they produce ambiguous derivations. gocc recognises ambiguous grammars and can automatically resolve LR-1 conflicts (see section 9).

gocc can also be used to generate stand-alone finite state automata for parsing simple regular languages. See for example: see the mail address example (section 8).

gocc supports parser error recovery (see section 13).

gocc supports action expressions, embedded in the input grammar, for the specification of semantic actions. For simple applications action expressions can be used to implement a syntax directed translation directly within the grammar. See the example in section 6. For more complex applications the action routines can be used to build an abstract syntax tree (AST), which is further processed by later stages of the application. See the example in section 12.

gocc has been successfully used to develop a query language compiler; a configuration / control language for a distributed system; parsers for protocol messages specified in ABNF [4] and gographviz (`http://code.google.com/p/gographviz/`). In addition gocc1 was used to generate the parser for gocc2, and gocc2 to generate the lexer and parser for gocc3.

gocc was designed to be easy to use and experience has shown that its users require very little background knowledge of language and compiler theory to apply it to simple language applications, such as syntax directed translation. An appreciation of mathematical formalism is usually enough and this guide is intended to provide sufficient information for such users, provided they understand:

- The go language;
- How to use context free grammars;
- How to separate lexical, syntactic and semantic analysis.

More complex applications, such as compiled languages and advanced protocol message parsing require more background, especially:

- The relationship between languages, grammars and automata;
- The relationship between regular and context free grammars;

- The equivalence of finite state automata with regular grammars; and of pushdown automata with context free grammars;
- The meaning and limits of top down/predictive parsing, bottom up parsing and deterministic parsing;
- The implications of language ambiguity as well as shift/reduce and reduce/reduce conflicts;
- The implications of grammars that generate languages outside the class of context free languages;
- Compiler design.

The author considers the *Dragon Book* [3] still the best reference for these topics. The reader is also directed to [2] for a modern treatment of compiler design, as well as [1] for a comprehensive treatment of the parsing techniques used in gocc.

gocc was conceived out of need in the year after Google released the *Go* language. At the time there was no other parser generator available, which could generate parsers in the *Go* language. The author set out to create a parser generator for the set of all deterministically parseable languages, which implied the LR(1) technique. Although there are now alternatives to gocc available to *Go* programmers we offer gocc to the community in the hope that someone may find it useful and as a token of thanks to Google for the gift of *Go* .

2 Copyright

3 Definition of terms

AST Abstract syntax tree
CFG Context Free Grammar
DFA Deterministic Finite State Automaton. gocc lexers are DFAs.
PDA Pushdown Automaton. gocc parsers are PDA's, and can recognise all deterministically parseable (LR-1) languages.

4 Getting started

1. Download and install *Go* from `http://golang.org`.
2. Set your **GOPATH** environment variable. See `http://golang.org/doc/code.html`.
3. Install *gocc* :
   (a) In your command line run: **go get code.google.com/p/gocc/** (go get will git clone gocc into GOPATH/src/code.google.com/p/gocc and run go install)

   or

   (b) Alternatively clone the repository: `https://code.google.com/p/gocc/source/checkout`. Followed by: `go install code.google.com/p/gocc`.

Test your installation by running `make test` from `$GOPATH/src/code.google.com/p/gocc`.

5 How to create and use a parser with gocc

Figure 1 shows the high-level design of a user application, which uses a parser generated with gocc.

- The user creates a target grammar conforming to the gocc BNF standard (see section 14).
- gocc reads the target grammar and generates the components shown in heavy outline in fig 1, i.e.: the lexer, parser, token and error packages.
- The user also creates a package called by the compiler to execute semantic actions for each recognised production of the target grammar. The methods of the semantic package provided by the user correspond to the method calls specified in the action expressions of the target grammar.
- The user application initialises a lexer object with the input text. Then it calls the **Parse(...)** method of the parser.
- Once created, the lexer and parser objects may be used repeatedly for successive inputs. For each input the lexer must be initialised with the next input text and the parser's **Parse(...)** method called with a reference to the lexer.
- The parser reads a stream of tokens (lexical elements) from the lexer (lexer) by repeatedly calling the lexer interface method, **lexer.Scan()** .

  ```
  type Scanner interface {
      Scan() (tok *token.Token)
  }
  ```

  Each call to **lexer.Scan** returns a pointer to token.Token.
- The lexer reads a stream of input characters and recognizes the tokens specified in the target grammar. After reaching the end of input it returns the end of input token to every call to **lexer.Scan()** .

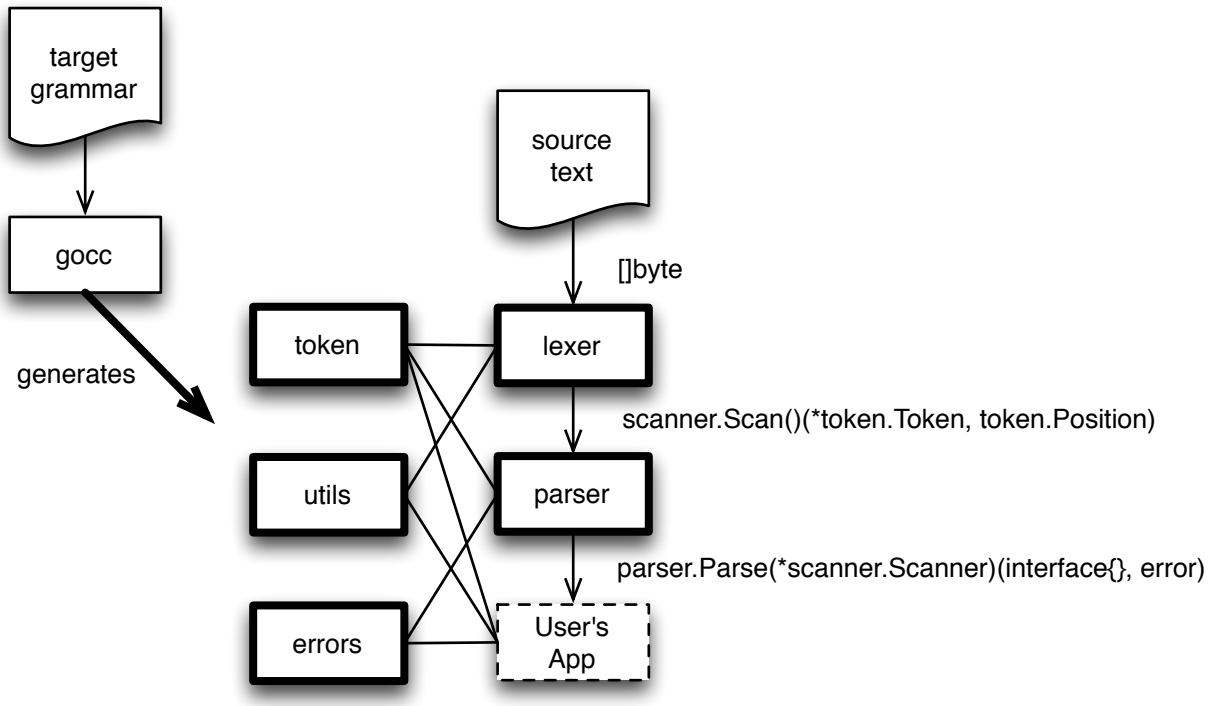


Figure 1: High-level design

- Whenever the parser recognises the complete body of a production of the target grammar, it calls the function specified in the action expression associated with that production alternative. The parsed symbols of the recognised production are passed as parameters to the action expression (see section 14). The result of the action expression is placed on the parser's stack as an attribute of the recognised language symbol.
- When the parser recognises the complete start production of the grammar it calls its associated action expression. The result of the action expression is returned to the user application as type **interface{}** together with a **nil** error value.
- If the parser encounters an error in the input it may perform automatic error recovery (see section 13). If the error is recoverable the parser places all the parsed language symbols associated with the error (completed productions as well as tokens) in a symbol of type ***errors.Error** and places this symbol on the parser stack. The parser then discards input tokens until it encounters an input token which may validly follow the recovered production and parsing continues normally. When error recovery is specified the user application must handle the error symbols which it may receive as attributes in calls to action expressions, or which may be returned as a top-level result of the parse to the calling application.
- If the parser encounters an irrecoverable error it returns a non-**nil** error value together with an *indeterminate* parse result.

6 First example

This example shows how action expressions in the BNF can be used to implement a syntax directed translation scheme without the need for further user provided packages, such as AST.

The source code of the following example can be found at

`$GOPATH/src/code.google.com/p/gocc/example/calc`

The grammar implements the simple desktop calculator described in [3]. The generated code is both a parser and an interpreter for the calculator.

The following files are provided by the user:

```
> ls -R
calc.bnf calc_test.go
```

calc.bnf contains the grammar for this example.

calc_test.go will be used to execute the generated code. It represents the user application.

6.1 Step 1: generate code

To generate code we run gocc from the directory containing `calc.bnf` with the following command:

```
> gocc calc.bnf
```

After running gocc we see that the directory structure now contains the following files:

```
> ls -R
LR1_sets.txt calc_test.go first.txt lexer_sets.txt token
calc.bnf errors lexer parser util

./errors:
errors.go

./lexer:
acttab.go lexer.go transitiontable.go

./parser:
action.go actiontable.go gototable.go parser.go productionstable.go

./token:
token.go

./util:
litconv.go rune.go
```

The generated files are:

LR1_sets.txt and first.txt Files containing information about the parser table generation process. They are useful for debugging the parser.

lexer_sets.txt File containing information about the lexer generation process. It us useful for debugging the lexer.

errors/errors.go Declares `type Error`, which is used during automatic recovery from errors in the input. See section 13 for more details.

lexer/ Contains the files generated for the lexer.

parser/ Contains the files generated for the parser. The interpreter code is embedded within `parser/productionstable.go`.

token/token.go Contains the declaration of the tokens of the grammar.

util/litconv.go Contains functions to convert a token literal to a value, e.g.: int64 or rune.

util/rune.go Contains func() RuneToString, which is used by debug code in the lexer.

6.2 The example grammar

`calc.bnf` displays the main features of a gocc BNF file:

```
/* Lexical part */

_digit : '0'-'9' ;

int64 : '1'-'9' {_digit} ;

!whitespace : ' ' | '\t' | '\n' | '\r' ;

/* Syntax part */

<<
import(
    "code.google.com/p/gocc/example/calc/token"
    "code.google.com/p/gocc/example/calc/util"
)
>>

Calc : Expr;

Expr
    : Expr "+" Term          << $0.(int64) + $2.(int64), nil >>
    | Term
    ;

Term
    : Term "*" Factor        << $0.(int64) * $2.(int64), nil >>
```

```
    | Factor
    ;

Factor
    : "(" Expr ")"            << $1, nil >>
    | int64                   << util.IntValue($0.(*token.Token).Lit) >>
    ;
```

The BNF has two parts: a lexical and a syntax part. Both parts are optional – gocc can be used to generate only a lexer or only a parser – but every BNF must have at least a lexical part or a syntax part. This example has both and gocc will generate a lexer, a parser and all the support they require to function.

It declares the imported package `code.google.com/p/gocc/example/calc/token` (generated by gocc), which will be used in the action expressions of some productions of the grammar.

The lexical part consists of:

1. A token definition, `int64`, which is used in the syntax part. Token identifiers always start with a lower case letter in the range: `'a'-'z'`. The token `int64` must start with a digit in the range, `'1'-'9'`, followed by zero or more digits in the range, `'0'-'9'`. It uses the regular definition, `_digit`, to declare the range, `'0'-'9'`. The curly braces around `_digit` indicate that it may be repeated zero or more times.

   A lexical pattern may also be enclosed in square brackets, e.g.: `[ '0'-'9' ]`, to indicate zero or one instances of it (in this case a decimal digit) may be present. See section 14 for details.

2. `!whitespace` declares an ignored token. The `'!'` before `whitespace` indicates that the token is to be ignored. The generated lexer suppresses ignored tokens so that the parser never sees them. They can be uses for white space and comments.

3. A regular definition, `_digit`. Regular definition identifiers always start with `'_'`. Regular definitions act like macros definitions in the BNF and allow tokens to be constructed from pre-defined components. Regular definitions and token definitions may not be mutually or self-recursive. That would lead to a context free grammar, which cannot be recognised by a finite state automaton.

gocc supports c-style line and block comments. The calc BNF contains two block comments.

The syntax part consists of:

1. An optional file header section, which is textually included in the generated parser file, `parser/productionstable.go`. The file header allows the user to specify imports, declarations and functions, which are required by the action routines in the syntax part. In the calc example the file header contains two import declarations.

2. The syntax start symbol, `Calc`, is the highest level syntax production of the BNF. The parser will try to match the input source with a valid derivation of the start symbol. All syntax production identifiers start with an upper case letter in the range, `'A'-'Z'`.

3. Each alternative of a production may optionally have an action expression. E.g.:

```
Factor : int64                  << util.IntValue($0.(*token.Token).Lit) >>
```

has an action expression, `<< util.IntValue($0.(*token.Token).Lit) >>`, which contains a call to the function, IntValue, in the generated file util/litconv.go. Attribute `$0`, associated with the symbol `int64` in the body of the production, is type asserted to `*token.Token`, of which the field `Lit` is passed as parameter to the function. Action expressions must always return a value of type, `(interface{}, error)`. If the returned error parameter is `nil`, the `interface{}` parameter is placed on the parser stack as the attribute of the recognised production.

4. Each recognised term in a production has an associated attribute. For example: the production alternative:

   ```
   Term : Term "*" Factor          << $0.(int64) * $2.(int64), nil >>
   ```

   has three symbols in its body. Each has an associated attribute on the stack: `Term`, type `int64`; `"*"`, type `*token.Token`; and `Factor`, type `int64`. They may be referred to in the action expression as $0, $1 and $2, respectively.

   The action expression in this example means the following: *return the product of $0 and $2, both cast to type* `int64`*, together with* `nil`*.*

When the parser has recognised the whole body of a production alternative, it calls the associated action expression with the attributes of the recognised language symbols of that body. If the action expression returns a non-nil error the parser stops and returns the error to the calling user application. If the action expression returns a nil error the parser replaces the recognised language symbols of the production on its stack with the attribute returned by the action expression.

If a recognised production alternative does not have a specific action expression, e.g.:

```
Calc : Expr;
```

the parser always pushes $0 on the stack as the attribute of the production.

The first alternative of `Expr` returns the sum of the attributes of `Expr` and `Term` after casting them to `int64`. The second alternative returns the attribute of `Term`.

The first alternative of `Term` returns the product of `Term` and `Factor` after casting them to `int64`. The second term returns the attribute of `Factor`.

The first alternative of `Factor` simply returns the attribute of the parenthesised `Expr`. The second alternative returns the value of a numeric token.

In the second alternative of `Factor` we use a method on the input token, which returns `(int64, error)`. Therefore the types of all numbers are `int64`.

6.3 The test program

The root folder of the **Calc** example contains **calc_test.go** , which has the following test program. In addition to testing the code it shows how to initialise and use the generated lexer and parser/interpreter.

```
package calc

import (
    "code.google.com/p/gocc/example/calc/lexer"
    "code.google.com/p/gocc/example/calc/parser"
    "fmt"
    "testing"
)

type TI struct {
    src    string
    expect int64
}

var testData = []*TI{
    {"1 + 1", 2},
    {"1 * 1", 1},
    {"1 + 2 * 3", 7},
}

func Test1(t *testing.T) {
    p := parser.NewParser()
    pass := true
    for _, ts := range testData {
        s := lexer.NewLexer([]byte(ts.src))
        sum, err := p.Parse(s)
        if err != nil {
            pass = false
            t.Log(err.Error())
        }
        if sum != ts.expect {
            pass = false
            t.Log(fmt.Sprintf("Error: %s = %d. Got %d\n", ts.src, sum, ts.expect))
        }
    }
    if !pass {
        t.Fail()
    }
}
```

6.4 Step 2: running go test

From the root folder of the **Calc** example, execute the following command:

```
> go test -v .
```

which generates the following output:

```
> go test -v
warning: building out-of-date packages:
    code.google.com/p/gocc/example/calc/token
    code.google.com/p/gocc/example/calc/lexer
    code.google.com/p/gocc/example/calc/errors
    code.google.com/p/gocc/example/calc/parser
installing these packages with 'go test -i' will speed future tests.

=== RUN Test1
--- PASS: Test1 (0.00 seconds)
PASS
ok      code.google.com/p/gocc/example/calc 0.017s
```

Congratulations! You have executed your first gocc-generated code.

7 Commandline syntax

```
> gocc -h
usage: gocc flags bnf_file

  bnf_file: contains the BNF grammar

Flags:
  -a=false: automatically resolve LR(1) conflicts
  -debug_lexer=false: enable debug logging in lexer
  -debug_parser=false: enable debug logging in parser
  -h=false: help
  -o="$HOME/goprj/src/code.google.com/p/gocc": output dir.
  -p="code.google.com/p/gocc": package
  -prof=false: write profile to file
  -u=false: allow unreachable productions
  -v=false: verbose
```

8 Example: parsing simple mail addresses

This example shows how gocc can be used to generate a stand-alone FSA to parse a regular language. The goal is to parse simple mail address specifications like: `mailbox@gmail.com` or `"mail box"@gmail.com`. The source code of the sample can be found at

`$GOPATH/src/code.google.com/p/gocc/example/mail`

`mail.bnf` contains:

```
!whitespace : '\t' | '\n' | '\r' | ' ' ;

_atext  : 'A'-'Z' | 'a'-'z' | '0'-'9'
```

```
        | '!' | '#' | '$' | '%' | '&' | '\'' | '*' | '+' | '-' | '/'
        | '=' | '?' | '^' | '_' | '`' | '{' | '|' | '}' | '~'
        | '\u0100'-'\U0010FFFF'
;

_atom : _atext {_atext} ;

_dotatom : _atom {'.' _atom} ;

_quotedpair : '\\' . ;

_quotedstring : '"' (_quotedpair | .) {_quotedpair | .} '"' ;

addrspec : (_dotatom | _quotedstring)  '@' _dotatom ;
```

The production

```
_quotedpair : \\ . ;
```

uses `'.'` to specify that any UTF-8 rune will be accepted after a `'\'` character in the input.

We generate code for the example by

```
gocc mail.bnf
```

which produces the following generated code:

```
> ls -R
lexer lexer_sets.txt mail.bnf parser_test.go token util

./lexer:
acttab.go lexer.go transitiontable.go

./token:
token.go

./util:
rune.go
```

Note that no parser had been generated, because the BNF does not include a syntax part. gocc generates only a DFA/lexer for the lexical part, which is present in the BNF. This application uses only a DFA to parse its input.

`parser_test.go` shows how the generated DFA could be repeatedly invoked to parse a stream of email addresses:

```
package mail

import (
    "code.google.com/p/gocc/example/mail/lexer"
    "code.google.com/p/gocc/example/mail/token"
    "testing"
```

```
)

var testData1 = map[string]bool{
    "mymail@google.com":          true,
    "@google.com":                false,
    `"quoted string"@mymail.com`: true,
    `"unclosed quote@mymail.com`: false,
}
```

Function `Test1` proves that the DFA correctly recognises addresses according to the specification.

```
func Test1(t *testing.T) {
    for input, ok := range testData1 {
        l := lexer.NewLexer([]byte(input))
        tok := l.Scan()
        switch {
        case tok.Type == token.INVALID:
            if ok {
                t.Errorf("%s", input)
            }
        case tok.Type == token.TokMap.Type("addrspec"):
            if !ok {
                t.Errorf("%s", input)
            }
        default:
            t.Fatalf("This must not happen")
        }
    }
}
```

Function `Test2` shows how to invoke the lexer repeatedly to parse a stream of addresses.

```
var checkData2 = []string{
    "addr1@gmail.com",
    "addr2@gmail.com",
    "addr3@gmail.com",
}

var testData2 = `
    addr1@gmail.com
    addr2@gmail.com
    addr3@gmail.com
`

func Test2(t *testing.T) {
    l := lexer.NewLexer([]byte(testData2))
    num := 0
```

```
    for tok := l.Scan(); tok.Type == token.TokMap.Type("addrspec"); tok = l.Scan() {
        if string(tok.Lit) != checkData2[num] {
            t.Errorf("%s != %s", string(tok.Lit), checkData2[num])
        }
        num++
    }
    if num != len(checkData2) {
        t.Fatalf("%d addresses parsed", num)
    }
}
```

9 Handling LR(1) conflicts

If a target grammar is outside the class of LR(1) grammars it cannot be parsed deterministically with one symbol lookahead. This condition manifests as LR(1) conflicts, of which there are two types:

Shift/Reduce conflict: The parser has recognised a valid production body on the stack, and can reduce it to the corresponding production.

However, the same symbols are also a valid prefix of the body of another, longer production. The parser could continue to shift the input symbols and attempt to recognise the longer production.

gocc uses the *maximal-munch rule* (see [2]) to resolve this conflict by always choosing shift over reduce. The longest valid production will therefore always be recognised.

Reduce/Reduce conflict: The parser has recognised a valid sequence of symbols, which can be reduced to more than one production.

gocc will always reduce the production that was declared first in the grammar.

10 Example: reduce/reduce conflict handling

The source code of the following example can be found at

`$GOPATH/src/code.google.com/p/gocc/example/reducereduce`

```
RR : A | B ;

B : a ;

A : a | A a ;
```

When we run *gocc* on `$GOPATH/src/code.google.com/p/gocc/example/reducereduce/rr.bnf` we discover a reduce/reduce conflict:

```
> gocc -v rr.bnf
LR(1) conflict: S4 Reduce:3(B) / Reduce:4(A)
ABORTING: 1 LR(1) conflicts
```

gocc does not generate code because the default for automatic LR(1) conflict resolution is `off`. From the output we see that *gocc* could reduce either of production `B` or `A` in state `4`.

gocc generates a number of informational files, and at this point we turn to

`$GOPATH/src/code.google.com/p/gocc/example/reducereduce/sm_sets.txt`

to analyse the conflict.

`sm_set.txt` contains the LR(1) sets, which will be translated into the states of the parser. Each state contains a set of *LR(1) items*, which specifies what the parser expects in that state.

An LR(1) item is a production alternative with the position of the parser marked by a •, and the next symbol expected after this production body, in double angle brackets. Alternatives of a production are in separate items. For example:

$A : a\bullet << \$ >>$

indicates that the compiler has recognised the production alternative, `A : a` and next expects to see the end of input character, `$`.

Getting back to our R/R conflict, `S4` in `sm_states.txt` represents state 4 and contains the following items:

$$
\begin{aligned}
&S4\{\\
&\qquad A : a\bullet << \$ >>\\
&\qquad B : a\bullet << \$ >>\\
&\qquad A : a\bullet << a >>\\
&\}
\end{aligned}
$$

We see that the bodies of all items in S4 are the same and that the parser has completely recognised them. Two items reduce to production `A` and one to production `B`. This is the reduce/reduce conflict: `A` vs `B`.

When *gocc* is run with the `-a` option it will automatically resolve this conflict by reducing production `B`, because it is declared in `rr.bnf` before `A`:

```
> gocc -a rr.bnf
Resolved 0 shift/reduce, 1 reduce/reduce conflicts
```

11 Example: Shift/reduce conflict handling

The source code of the following example can be found at

`$GOPATH/src/code.google.com/p/gocc/example/shiftreduce`

It is the classic example of the dangling else in the *C* language:

```
Stmt :
        if expr then Stmt
    |   if expr then Stmt else Stmt
;
```

When we run *gocc* on `$GOPATH/src/code.google.com/p/gocc/example/shiftreduce/sr.bnf` we discover a shift/reduce conflict:

```
> gocc -v sr.bnf
LR(1) conflict: S11 Shift:12 / Reduce:1(Stmt)
ABORTING: 1 LR(1) conflicts
```

The problem is in the last two items of state 11, where the next symbol is `else` and the parser can both shift and reduce:

$$
\begin{array}{l}
S11\{ \\
\quad Stmt : if\ expr\ then\ Stmt\bullet\ <<\ \$\ >> \\
\quad Stmt : if\ expr\ then\ Stmt\ \bullet else\ Stmt\ <<\ \$\ >> \\
\quad Stmt : if\ expr\ then\ Stmt\bullet\ <<\ else\ >> \\
\quad Stmt : if\ expr\ then\ Stmt\ \bullet else\ Stmt\ <<\ else\ >> \\
\}
\end{array}
$$

When automatic LR(1) conflict resolution is selected by the `-a` option, *gocc* resolves this conflict in the same way as specified in the *C* language specification: by shifting and parsing the longest valid production (*maximal-munch*). This means recognising the `else`-statement as part of the second `if`.

12 Example: Using an AST

The following example illustrates the use of user-provided action expressions to produce a simple abstract syntax tree (AST) for a list of simple statements.

The code for the example can be found at

`$GOPATH/src/code.google.com/p/gocc/example/astx`

The grammar is in `ast.bnf`:

```
<< import "code.google.com/p/gocc/example/astx/ast" >>

StmtList :
      Stmt             << ast.NewStmtList($0) >>
    | StmtList Stmt    << ast.AppendStmt($0, $1) >>
;

Stmt :
      id               << ast.NewStmt($0) >>
;
```

At the top of the grammar is an file header section containing an import statement for the user-provided package, `code.google.com/p/gocc/example/astx/ast`.

The production action expressions will use functions from the package, `ast`.

The start production, `StmtList` returns a tuple: `(ast.StmtList, error)`, as we can see from the code of functions `NewStmtList` and `AppendStmt` in

`$GOPATH/src/code.google.com/p/gocc/example/astx/ast.go`:

```
package ast

import(
    "code.google.com/p/gocc/example/astx/token"
)

type (
    StmtList []Stmt
    Stmt  string
)

func NewStmtList(stmt interface{}) (StmtList, error) {
    return StmtList{stmt.(Stmt)}, nil
}

func AppendStmt(stmtList, stmt interface{}) (StmtList, error) {
    return append(stmtList.(StmtList), stmt.(Stmt)), nil
}

func NewStmt(stmtList interface{}) (Stmt, error) {
    return Stmt(stmtList.(*token.Token).Lit), nil
}
```

Note the following:

- The attributes of the language symbols in the production are passed to the action expressions as parameters, referred to as $\$0, \$1, ...$
- The type of the parameters passed to the action expressions is `interface{}` and must be type asserted by the called function to the expected type.
- The parser will return the result of a successful parse, a `StmtList`, to the calling application as type `interface{}`. The calling application must type assert the returned value to the expected type.

If we run

```
go test -v .
```

from the directory

`$GOPATH/src/code.google.com/p/gocc/example/astx/`

we get the following output:

```
> go test -v .
warning: building out-of-date packages:
        code.google.com/p/gocc/example/astx/token
        code.google.com/p/gocc/example/astx/ast
        code.google.com/p/gocc/example/astx/errors
        code.google.com/p/gocc/example/astx/parser
        code.google.com/p/gocc/example/astx/lexer
installing these packages with 'go test -i .' will speed future tests.

=== RUN TestPass
input: a b c d e f
output: [a b c d e f]
--- PASS: TestPass (0.00 seconds)
=== RUN TestFail
input: a b ; d e f
--- FAIL: TestFail (0.00 seconds)
ast_test.go:23:  Error: illegal -1(-1) ; @ 1:5, expected one of: id $
        FAIL
exit status 1
FAIL code.google.com/p/gocc/example/astx 0.015s
```

The first test, `TestPass`, has a valid input string, `"a b c d e f"`; and parses successfully; and returns the expected StmtList, `[a b c d e f]`.

The input to the second test, `TestFail`, contains an invalid identifier, `;`. The parser returns an error, indicating that it encountered an invalid token when it expect a token of type `id` or the end of input.

13 Example: Parser error recovery

Without error recovery a *gocc* parser terminates when it reaches the first error in the input. Sometimes it is convenient to attempt to continue the parse and this can be achieved in *gocc* by specifying in the grammar which productions can recover from errors in the input.

When the *gocc* reserved word, `error`, is the first symbol in a production alternative, it indicates that that production can recover from input errors.

We modify the AST example to illustrate error recovery. See:

`$GOPATH/src/code.google.com/p/gocc/examples/errorrecovery/er.bnf`:

```
<< import "code.google.com/p/gocc/example/errorrecovery/ast" >>

StmtList :
      Stmt             << ast.NewStmtList($0) >>
    | StmtList Stmt    << ast.AppendStmt($0, $1) >>
;
```

```
Stmt :
      id                << ast.NewStmt($0) >>
    | error
;
```

The production, `Stmt`, now has an alternative: `| error`

This indicates to *gocc* that input errors can be handles in production `Stmt`.

From the directory,

`$GOPATH/src/code.google.com/p/gocc/examples/errorrecovery/`,

run `go test` as follows:

```
> go test -v .
warning: building out-of-date packages:
        code.google.com/p/gocc/example/errorrecovery/token
        code.google.com/p/gocc/example/errorrecovery/ast
        code.google.com/p/gocc/example/errorrecovery/errors
        code.google.com/p/gocc/example/errorrecovery/parser
        code.google.com/p/gocc/example/errorrecovery/lexer
installing these packages with 'go test -i .' will speed future tests.

=== RUN TestFail
input: a b ; d e f
parser.firstRecoveryState: State 3
parser.firstRecoveryState: State 1, canRecover, true
output: [
    a
    error:
        Err: nil
        ErrorToken: ";"(-1)
        ErrorPos: 1:5
        ErrorSymbols: ["b"(1)]
        ExpectedTokens: [error $ id]
    d
    e
    f
]
--- PASS: TestFail (0.00 seconds)
PASS ok      code.google.com/p/gocc/example/errorrecovery 0.015s
```

The test case can be found in
`$GOPATH/src/code.google.com/p/gocc/examples/errorrecovery/er_test.go`.
It calls the parser with input string, `"a b ; d e f"`, which contains an invalid token, `;`.

From the `go test` output we see that the parser successfully recovered from the input error and returned a `StmtList` containing an error symbol between `[a` and `d, e, f]`. The `id`, `b`, was lost in

the error recovery. TBD: Explain exactly why the 'b' was lost. The errored token was ; (invalid token) when the parser expected one of `error, $` (end of input) or `id`.

The parser returned an error value of `nil`, because it successfully recovered from the error.

Note:

1. When error recovery is allowed the user's code must expect errors and handle the appropriately in the code called by the production action expressions, as well as by the code handling the results returned by the parser.
2. The parser will still return a non-`nil` error value if it encounters an irrecoverable error.

See `$GOPATH/src/code.google.com/p/gocc/examples/errorrecovery/errors/error.go` for the definition of errors.Error.

14 gocc syntax

A gocc source file contains UTF-8 encoded Unicode text.

c-style block and line comments are allowed in the source code. All comments are suppressed by the lexer. In the following BNF **!comment** means that **comment** is a token, which is recognised and suppressed by the lexer – it is not seen by the parser.

```
!comment : _lineComment | _blockComment ;

_lineComment : '/' '/' {.} '\n' ;

_blockComment : '/' '*' {. | '*'} '*' '/' ;
```

gocc source code consists of a sequence of tokens separated by white space. The white space characters are:

Character	Unicode value	go char literal
space	0x20	' '
horizontal tab	0x09	'\t'
newline	0x0a	'\n'
carriage return	0x0d	'\r'

White space is suppressed by the lexer.

```
!whitespace : ' ' | '\t' | '\n' | '\r' ;
```

A gocc source file must contain at least one of a lexical part or a syntax part.

```
Grammar
    : LexicalPart SyntaxPart
    | LexicalPart
```

```
    | SyntaxPart
    ;
```

The lexical part is a sequence of lexical productions.

```
LexicalPart
    : LexProductions
    ;
```

```
LexProductions
    : LexProduction
    | LexProductions LexProduction
    ;
```

Each lexical production is token definition or a regular definition or an ignored token definition. Token identifiers start with a lower case character in the range 'a'-'z', regular definition identifiers start with '_' and ignored token identifiers start with '!'.

Regular definitions are used as building blocks for other regular definitions, token definitions and ignored token definitions. Regular definitions may not be mutually or self recursive.

Token definitions define the tokens that are recognised by the lexer and passed to the parser.

Ignored token definitions define tokens that are recognised by the lexer but suppressed so that the parser never sees them. Comments and the white space separating tokens are examples of ignored tokens.

```
LexProduction
    : tokId ":" LexPattern ";"          // token definition
    | regDefId ":" LexPattern ";"       // regular definition
    | ignoredTokId ":" LexPattern ";"   // ignored token defition
    ;
```

Token identifiers start with a lowercase letter in the range 'a'-'z'.

```
tokId : _tokId ;

_tokId : _lowcase {_id_char} ;

_lowcase : 'a'-'z' ;

_id_char : _upcase | _lowcase | '_' | _digit ;

_upcase : 'A'-'Z' ;

_digit : '0'-'9' ;
```

Regular definition identifiers start with '_'.

```
regDefId : '_' {_id_char} ;
```

Ignored token identifiers start with `'!'`

```
ignoredTokId : '!' _tokId ;
```

A lexical pattern is one or more alternatives, each consisting of a sequence of terms.

```
LexPattern
    : LexAlt
    | LexPattern "|" LexAlt
    ;

LexAlt
    : LexTerm
    | LexAlt LexTerm
    ;
```

A lexical term can be one of:

`.`	Match any UTF-8 rune. This match is only applied after all more specific terms have been matched.
`char_lit`	A specific character literal, e.g.: 'a'.
`char_lit "-" char_lit`	An inclusive range, e.g.: 'a'-'z'
`regDefId`	the identifier of a regular definition production in the BNF
`"[" LexPattern "]"`	An optional lexical pattern
`"{" LexPattern "}"`	Zero or more instances of a lexical pattern
`"(" LexPattern ")"`	A grouped lexical pattern, e.g.: `('a'` 'b')—

```
LexTerm
    : "."
    | char_lit
    | char_lit "-" char_lit
    | regDefId
    | "[" LexPattern "]"
    | "{" LexPattern "}"
    | "(" LexPattern ")"
    ;
```

Character literals are specified as a valid Unicode or byte value, enclosed in single quotes:

```
char_lit
    : '\'' (_unicode_value | _byte_value) '\''
    ;
```

A Unicode character may be specified as a character literal (e.g.: `a`), a Unicode value or an escaped character.

```
_unicode_value
    : .                    // Any UTF-8 character literal
    | _little_u_value
    | _big_u_value
    | _escaped_char
    ;

_byte_value
    : _octal_byte_value
    | _hex_byte_value
    ;

_little_u_value
    : '\\' 'u' _hex_digit _hex_digit _hex_digit _hex_digit
    ;

_big_u_value
    : '\\' 'U' _hex_digit _hex_digit _hex_digit _hex_digit
               _hex_digit _hex_digit _hex_digit _hex_digit
    ;

_escaped_char
    : '\\' ( 'a' | 'b' | 'f' | 'n' | 'r' | 't' | 'v' | '\\' | '\'' | '"' )
    ;

_octal_byte_value
    : '\\' _octal_digit _octal_digit _octal_digit
    ;

_hex_byte_value
    : '\\' 'x' _hex_digit _hex_digit
    ;

_octal_digit
    : '0' - '7'
    ;

_hex_digit
    : '0' - '9'
    | 'A' - 'F'
    | 'a' - 'f'
    ;
```

The syntax part may start with an optional file header, followed by a sequence of syntax productions. The file header is textually included in the generated parser file, `productiontable.go`, and is used to declare imports, constants, types and functions required by the action expressions of the

productions.

```
SyntaxPart
    : FileHeader SyntaxProdList
    | SyntaxProdList
    ;

FileHeader
    : action_lit
    ;

SyntaxProdList
    : SyntaxProduction
    | SyntaxProdList SyntaxProduction
    ;
```

Syntax productions define a context free language and may be mutually and self recursive. Left recursion is preferable to right recursion in LR grammar productions, as this leads to less stack activity during parsing. A syntax production starts with an identifier, followed by ":" and one or more alternative bodies, which are separated by "—".

Syntax production identifiers start with an upper case character in the range 'A'-'Z'.

```
SyntaxProduction
    : prodId ":" Alternatives ";"
    ;

prodId
    : _upcase {_id_char}
    ;

Alternatives
    : SyntaxBody
    | Alternatives "|" SyntaxBody
    ;
```

A syntax body is a sequence of syntax symbols, optionally followed by an action literal. If a syntax body starts with the keyword, `error`, it indicates that the parser may accept an error in the input of the production and continue parsing if it is able to recover from the error.

A syntax body may also be empty, indicated by the reserved word, `empty`. This allows you to write productions with optional elements, such as:

```
A : B "c";
B : "b" ;
```

This grammar will match strings, `b c` or `c`.

```
SyntaxBody
    : Symbols
```

```
    | Symbols action_lit
    | "error"
    | "error" Symbols
    | "error" Symbols action_lit
    | "empty"
    ;

Symbols
    : Symbol
    | Symbols Symbol
    ;
```

An action literal is any string of characters enclosed in "¡¡" "¿¿". The contents of the action literal is invoke as an expression when the associated production body has been recognised by the parser. The action expression must return a tuple of type `(interface{}, error)`. If the second parameter is not `nil` the parser terminates and returns the error to its caller. If the second parameter is `nil` the parser associates first parameter with the recognise syntax production as its attribute.

```
action_lit
    : '<' '<' . {.} '>' '>'
    ;
```

gocc recognises the following terminal syntax symbols:

`prodId`	The identifier of a syntax production defined in the grammar.
`tokId`	The identifier of lexical production defined in the grammar.
`string_lit`	A string literal value, e.g.: `"a string"`

```
Symbol
    : prodId
    | tokId
    | string_lit
    ;
```

A string literal can be any valid go raw string (e.g.: `` `a raw string` ``) or interpreted string (e.g.: `"an interpreted string"`).

```
string_lit
    : _raw_string
    | _interpreted_string
    ;

_raw_string
    : '`' {.} '`'
    ;

_interpreted_string
```

```
    : '"' { _unicode_value | _byte_value } '"'
    ;
```

References

[1] Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques. A Practical Guide. Second Edition.* Monographs in Computer Science, Springer, 2008

[2] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J.H. Jacobs and Koen Langendoen. *Modern Modern Compiler Design. Second Edition.* Springer 2012

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman. *Compilers. Principles, Techniques, & Tools. Second Edition.* Addison Wesley, 2007

[4] D. Crocker, Ed. *Augmented BNF for Syntax Specifications: ABNF* RFC 5234, January 2008

[5] *The Go Language Specification* `http://golang.org/ref/spec`