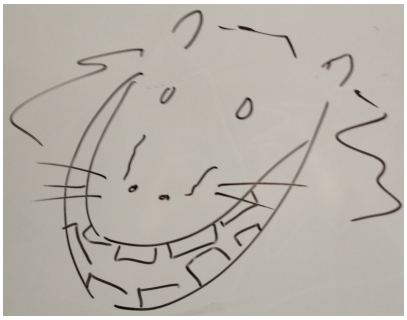
Learn You a gocc for Great Good

or

How to save the world by using compiler theory

2013-04-24



Contents

1 Introduction

gocc is a compiler kit which generates lexers and parsers from a common EBNF file. It can also be used to generate finite state automata, for example: see the mail address example (section 8).

gocc is an LR(1) parser generator with automatic LR(1) conflict resolution (see section 9) and automatic parser error recovery (see section 13). It has a simple syntax directed translation scheme (SDT) embedded in the input grammar, which is used to specify semantic actions; or, for simple applications, to specify a direct implementation of syntax directed translation within the grammar.

gocc has been successfully used to develop a query language compiler; a configuration/control language for a distributed system; as well as a parser for protocol messages specified in ABNF [4]. It is currently used in the development of an ASN.1 compiler.

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:

- 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 and shift/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
SDT Syntax Directed Translation Scheme

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 appendix A).
- gocc reads the target grammar and generates the components shown in heavy outline in fig 1, i.e.: the scanner, parser, token and error packages.

  Note: the scanner is an optionally generated component (see section 7).
- The user creates a user application, which creates the scanner and parser objects.
- 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 SDT statements in the target grammar.
- The user application initialises a scanner object with the input text. Then it calls the **Parse(...)** method of the parser.
- Once created, the scanner and parser objects may be used repeatedly for successive inputs. For each input the scanner must be initialised with the next input text and the parser's **Parse(...)** method called with a reference to the scanner.
- The parser reads a stream of tokens (lexical elements) from the scanner (lexer) by repeatedly calling the scanner interface method, **scanner.Scan()** .

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

  Each call to **scanner.Scan** returns two values: a pointer to token.Token and token.Position. The former contains information of the last token scanned and the latter its position in the input text.
- The scanner 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 **scanner.Scan()** .
- Whenever the parser recognises the complete body of a production of the target grammar, it calls the function specified in the SDT element associated with that production. The parsed symbols of the recognised production are passed as parameters to the SDT function (see section **??**). The result of the SDT call is placed on the parser's stack as an attribute of the recognised language symbol.

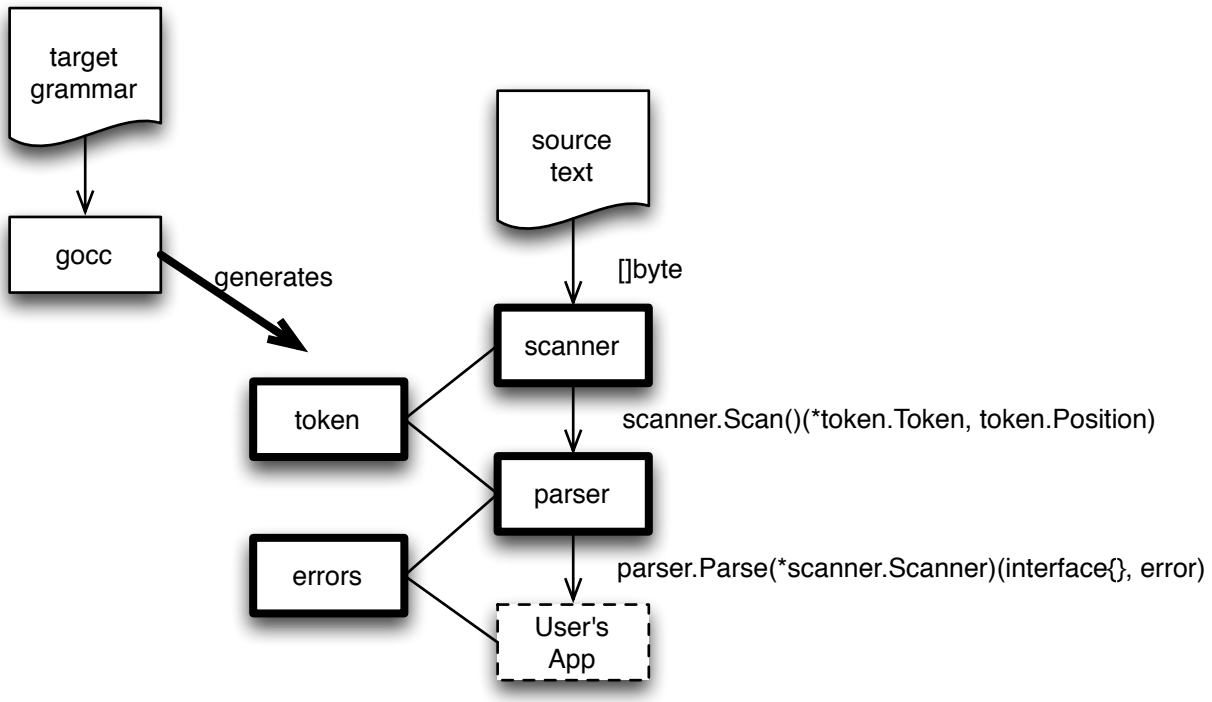


Figure 1: High-level design

- When the parser recognises the complete start production of the grammer it calls its associated SDT element. The result of the SDT call 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 ***error.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 SDT elements, 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 **nil** error value together with an *indeterminate* parse result.

6 First example

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/
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 -s calc.bnf
```

gocc is invoked with the option, `-s`, to generate a default scanner for the project. See section **??** for more about the scanner.

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

```
> ls -R calc/
calc.bnf errors scanner sm_first_bodies.txt sm_transitions.dot
calc_test.go parser sm_first.txt sm_sets.txt token

calc/errors:
errors.go

calc/parser:
parser.go tables.go

calc/scanner:
scanner.go

calc/token:
token.go tokens.go
```

The generated files are:

sm_*.txt Files containing information about the table generation process. They are useful for debugging.

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

parser/parser.go, parser/tables.go contain the parser for the target language with the interpreter code embedded.

token/token.go, token/tokens.go contain the declaration of the tokens of the grammar.

6.2 The example grammar

```
<< import "calc/token" >>

Calc : Expr
;

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

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

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

The BNF of the example starts with an optional initial SDT. It declares the imported package `calc/token`, which will be used in SDT statements of some productions of the grammar.

The text of the initial SDT is expanded at the start of the `parser/tables.go`.

Every production alternative of the grammar has either an implicit or explicit SDT, which translates to a function with the signature:

`func ([]parser.Attrib)(parser.Attrib, error)`

where `parser.Attrib` is of type `interface{}`.

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

Any expression in a production's SDT must return `(parser.Attrib, nil)`. The expression may refer to the attributes of the language symbols of the recognised production body, $P : x_0..x_n$ as $\$0..\$n$.

An implicit SDT function is of the form:

```
func(X []parser.Attrib) (parser.Attrib, error) {
    return X[0], nil
}
```

Therefore the implicit (omitted) SDT is equivalent to the explicit SDT, `<< $0, nil >>`.

The first production of the grammar, `Calc`, is the start production. The body of `Calc` contains only one non-terminal: `Expr`, which is used recursively in the grammar. It has an implicit SDT which returns the attribute of `Expr`.

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 scanner and parser/interpreter.

```
/*** Lexical items ***/

/* Token definitions */

tokId :
    _tokId
    ;

regDefId
    : '_' {_id_char}
    ;

prodId
    : _upcase {_id_char}
    ;

string_lit
    : _raw_string
    | _interpreted_string
    ;

ignoredTokId
    : '!' _tokId
    ;

/* Ignored tokens */
```

```
!comment
    : _lineComment | _blockComment
    ;

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

/* Regular definitions */

_upcase
    : 'A'-'Z'
    ;

_lowcase
    : 'a'-'z'
    ;

_digit
    : '0'-'9'
    ;

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

_octal_digit
    : '0' - '7'
    ;

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

_unicode_value
    : .
    | _little_u_value
    | _big_u_value
    | _escaped_char
    ;

_byte_value
    : _octal_byte_value
    | _hex_byte_value
```

```
    ;

_octal_byte_value
    : '\\' _octal_digit _octal_digit _octal_digit
    ;

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

_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' | '\\' | '\'' | '"' )
    ;

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

_tokId
    : _lowcase {_id_char}
    ;

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

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

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

_lineComment
```

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

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


/*** Syntactic items ***/

<< import "gocc3/ast" >>

Grammar
    : LexicalPart SyntaxPart
    | LexicalPart
    | SyntaxPart
    ;

LexicalPart
    : LexProductions
    ;

LexProductions
    : LexProduction
    | LexProductions LexProduction
    ;

LexProduction
    : tokId ":" LexPattern ";"
    | regDefId ":" LexPattern ";"
    | ignoredTokId ":" LexPattern ";"
    ;

LexPattern
    : LexAlt
    | LexPattern "|" LexAlt
    ;

LexAlt
    : LexTerm
    | LexAlt LexTerm
    ;

LexTerm
    : "."
    | char_lit
    | char_lit "-" char_lit
```

```
    | regDefId
    | "[" LexPattern "]"
    | "{" LexPattern "}"
    | "(" LexPattern ")"
    ;

SyntaxPart
    : FileHeader SyntaxProdList
    | SyntaxProdList
    ;

SyntaxProdList
    : SyntaxProduction
    | SyntaxProdList SyntaxProduction
    ;

SyntaxProduction
    : prodId ":" Alternatives ";"
    ;

Alternatives
    :    SyntaxBody
    |    Alternatives "|" SyntaxBody
    ;

SyntaxBody
    : Symbols
    | Symbols action_lit
    | "error"
    | "error" Symbols
    | "error" Symbols action_lit
    | "empty"
    ;

Symbols
    : Symbol
    | Symbols Symbol
    ;

Symbol
    : prodId
    | tokId
    | string_lit
    ;

// Shared productions
```

```
FileHeader
    : action_lit
    ;
```

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:

```
warning: building out-of-date packages:
    code.google.com/p/gocc/example/calc/token
    code.google.com/p/gocc/example/calc/errors
    code.google.com/p/gocc/example/calc/parser
    code.google.com/p/gocc/example/calc/scanner
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 0.106s
```

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

7 Commandline syntax

```
gocc is an LR(1) parser generator.

Usage:
    gocc [options] bnf_file

gocc reads the BNF target grammar from bnf_file and generates a parser
(and optionally a scanner) for the grammar.

Options:
    -a          Automatically resolve LR(1) conflicts.
                default: off

    -o          Output directory.
                default: working directory (current directory)
                Run gocc without arguments to see default.

    -p          package of the parser application.
```

```
                default: working directory without prefix: $GOPATH/src/
                Run gocc without arguments to see default.

    -s          Generate a scanner
                default: false

    -u          allow unreachable productions
                (only recommended during debugging of grammar)

    -v          verbose
                Shows list of LR(1) conflicts.
```

8 Example: parsing simple mail addresses

TBD

9 Handling LR(1) conflicts

If a target grammer 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:

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

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 SDT rules 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 SDT containing an import statement for the user-provided package, `code.google.com/p/gocc/example/astx/ast`.

The production SDTs 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 SDT function calls as parameters, referred to as $\$0, \$1, ...$
- The type of the parameters passed to the functions in the SDTs 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 ouput:

```
> 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/scanner
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/scanner
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 SDTS, 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.

A gocc target grammar

A gocc target grammar is written in UTF-8. See section B for a definition of the lexical elements of a gocc target grammar.

```
/*** Lexical items ***/

/* Token definitions */

tokId :
    _tokId
    ;

regDefId
    : '_' {_id_char}
    ;

prodId
    : _upcase {_id_char}
    ;

string_lit
    : _raw_string
    | _interpreted_string
    ;
```

```
ignoredTokId
    : '!' _tokId
    ;

/* Ignored tokens */

!comment
    : _lineComment | _blockComment
    ;

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

/* Regular definitions */

_upcase
    : 'A'-'Z'
    ;

_lowcase
    : 'a'-'z'
    ;

_digit
    : '0'-'9'
    ;

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

_octal_digit
    : '0' - '7'
    ;

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

_unicode_value
    : .
    | _little_u_value
    | _big_u_value
    | _escaped_char
```

```
    ;

_byte_value
    : _octal_byte_value
    | _hex_byte_value
    ;

_octal_byte_value
    : '\\' _octal_digit _octal_digit _octal_digit
    ;

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

_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' | '\\' | '\'' | '"' )
    ;

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

_tokId
    : _lowcase {_id_char}
    ;

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

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

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

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

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


/*** Syntactic items ***/

Grammar
    : LexicalPart SyntaxPart
    | LexicalPart
    | SyntaxPart
    ;

LexicalPart
    : LexProductions
    ;

LexProductions
    : LexProduction
    | LexProductions LexProduction
    ;

LexProduction
    : tokId ":" LexPattern ";"
    | regDefId ":" LexPattern ";"
    | ignoredTokId ":" LexPattern ";"
    ;

LexPattern
    : LexAlt
    | LexPattern "|" LexAlt
    ;

LexAlt
    : LexTerm
    | LexAlt LexTerm
    ;

LexTerm
```

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

SyntaxPart
    : FileHeader SyntaxProdList
    | SyntaxProdList
    ;

SyntaxProdList
    : SyntaxProduction
    | SyntaxProdList SyntaxProduction
    ;

SyntaxProduction
    : prodId ":" Alternatives ";"
    ;

Alternatives
    : SyntaxBody
    | Alternatives "|" SyntaxBody
    ;

SyntaxBody
    : Symbols
    | Symbols action_lit
    | "error"
    | "error" Symbols
    | "error" Symbols action_lit
    | "empty"
    ;

Symbols
    : Symbol
    | Symbols Symbol
    ;

Symbol
    : prodId
    | tokId
    | string_lit
    ;
```

```
// Shared productions

FileHeader
    : action_lit
    ;
```

B Lexical elements

TBD: Expand this appendix.

The basic unit of lexical elements is the UTF-8 character.

gocc has the following tokens:

id An id starts with a Unicode letter and is followed by any sequence of unicode letter or '_'

string Strings can be both types of *Go* string literal: interpreted strings (e.g.: `"Hello World"`) or raw strings (`` `Hello World` ``).

char Can be any of:

- A simple character declaration, e.g.: 'a';
- An octal character literal, e.g.: '\141';
- A hexadecimal character literal, e.g.: '\x61';
- A unicode literal, e.g.: '\u61' or '\U0061';
- Or an escaped character, such as '\n'.

See the *Go* specification [5] for details.

sdt_lit An SDT literal is enclosed in double angle brackets, e.g.:

```
<< ast.AddFoo($0, $1) >>
```

gocc supports both types of *Go* comments:

1. Line comments start with the sequence `//` and stop at the end of the line.
2. General comments start with the sequence `/*` and continue through the sequence `*/`.

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`