計算機科学実験3 中間レポート1

杉本風斗

平成 28 年 4 月 28 日

1 概要

Small C の字句・構文解析および抽象構文木の変換処理をするプログラムを作成した.
実装言語は Go (https://golang.org/) を使用した.

2 プログラムの概要

プログラムの実行方法とソースファイルの構造を説明する.

2.1 使い方

実装に使用した Go のバージョンは 1.6 である. ソースファイルの置いたディレクトリ内で以下のようにしてビルドして実行できる.

```
$ make
$ ./small-c program.sc
```

引数にソースファイルを与えて実行すると, 抽象構文木が pretty print されて出力される.
簡単な Small C プログラムを与えて実行した例を示す.

```
$ cat test.sc
int main() {
  return 1 + 2;
}

$ ./small-c test.sc
[]main.Statement{
  &main.FunctionDefinition{
    pos:        1,
    TypeName:   "int",
    Identifier: &main.IdentifierExpression{
      pos:    5,
```

```
      Name:   "main",
      Symbol: (*main.Symbol)(nil),
    },
    Parameters: []main.Expression{},
    Statement:  &main.CompoundStatement{
      pos:          12,
      Declarations: []main.Statement{},
      Statements:   []main.Statement{
        &main.ReturnStatement{
          pos:   12,
          Value: &main.BinaryExpression{
            Left: &main.NumberExpression{
              pos:   23,
              Value: "1",
            },
            Operator: "+",
            Right:    &main.NumberExpression{
              pos:   27,
              Value: "2",
            },
          },
          FunctionSymbol: (*main.Symbol)(nil),
        },
      },
    },
  },
}
```

出力の形式はあとで説明する抽象構文木の構造体定義に基づいている.

不正な文法のファイルを与える例を示す. ソースファイル中のエラーの位置とともにエラーメッセージが表示される.

```
$ cat error.sc
int a
int b;

$ ./small-c error.sc
2:1: syntax error: unexpected TYPE, expecting ';' or ','
```

2.2 ソースファイルの構造

プログラムを構成する主要なソースファイルを説明する.

- parser.go.y: yacc を用いた字句解析および構文解析
- ast.go: 抽象構文木の構造体の定義
- parse.go: 構文解析のラッパ関数および抽象構文木の変換処理
- main.go: コマンドラインから呼び出される main 関数

parse_test.go などのファイルは開発用のユニットテストである. 以下のコマンドでまとめて実行できる.

```
$ make test
```

3 構文解析

3.1 抽象構文木の構造体定義

抽象構文木のデータ構造を説明する. 構造体の定義は ast.go に書かれている. 説明のため, 以下には ast.go の内容から構造体定義の部分だけ抜き出したものを示す.

変換処理や意味解析の処理が複雑にならないように, 構造体の数が多くならないように工夫をした.

大きく分けて Expression, Statement, 定義の三種類がある. 構文木要素のソースコード上の位置は pos という field に格納されている. ただし, BinaryExpression など子要素を含む複合的な構造体は, 子要素からソースコード上の位置を求めることができるので, ソースコード上の位置を直接 field に格納していない.

```
type Node interface {
  Pos() token.Pos
}

type Expression interface {
  Node
}

type ExpressionList struct {
  Values []Expression
}

type NumberExpression struct {
  pos   token.Pos
  Value string
}
```

```
type IdentifierExpression struct {
  pos    token.Pos
  Name   string
  Symbol *Symbol
}

type UnaryExpression struct {
  pos      token.Pos
  Operator string
  Value    Expression
}

type BinaryExpression struct {
  Left     Expression
  Operator string
  Right    Expression
}

type FunctionCallExpression struct {
  Identifier Expression
  Argument   Expression
}

type ArrayReferenceExpression struct {
  Target Expression
  Index  Expression
}

type PointerExpression struct {
  pos   token.Pos
  Value Expression
}

type Declarator struct {
  Identifier Expression
  Size       int
}

type Declaration struct {
  pos         token.Pos
  VarType     string
  Declarators []*Declarator
}
```

```
type FunctionDefinition struct {
  pos        token.Pos
  TypeName   string
  Identifier Expression
  Parameters []Expression
  Statement  Statement
}

type Statement interface {
  Node
}

type CompoundStatement struct {
  pos          token.Pos
  Declarations []Statement
  Statements   []Statement
}

type ExpressionStatement struct {
  Value Expression
}

type IfStatement struct {
  pos            token.Pos
  Condition      Expression
  TrueStatement  Statement
  FalseStatement Statement
}

type WhileStatement struct {
  pos       token.Pos
  Condition Expression
  Statement Statement
}

type ForStatement struct {
  pos       token.Pos
  Init      Expression
  Condition Expression
  Loop      Expression
  Statement Statement
}
```

```
type ReturnStatement struct {
  pos   token.Pos
  Value Expression
  FunctionSymbol *Symbol
}

type ParameterDeclaration struct {
  pos        token.Pos
  TypeName   string
  Identifier Expression
}
```

3.2 字句解析

字句解析には, go の標準ライブラリの go/scanner を使った. 字句解析処理は, Lexer 構造体の Lex() 関数に書いており, 構文解析部から逐次 Lex() を呼び出すという仕組みになっている.

```
type Lexer struct {
    scanner.Scanner
    result []Statement
    token Token
    pos token.Pos
    errMessage string
}

func (l *Lexer) Lex(lval *yySymType) int {
  pos, tok, lit := l.Scan()
  token_number := int(tok)

  // 省略

  lval.token = Token{ tok: tok, lit: lit, pos: pos }
  l.token = lval.token

  return token_number
}
```

3.3 構文解析

構文解析器には yacc の Go 実装である goyacc (https://golang.org/cmd/yacc/) を使用した. 構文定義の文法は本家 yacc と同じである. 本家 yacc と違うのは, プログラムを記述する場所で C で

はなく Go で記述できるという点のみだと考えてよい.

parser.go.y の構文定義の一部を例として示す. 構文解析器から得られたトークン情報などから構造体を順番に組み立てる処理をしている.

```
statement
  : ';'
  {
    $$ = nil
  }
  | expression ';'
  {
    $$ = &ExpressionStatement{ Value: $1 }
  }
  | compound_statement
  | IF '(' expression ')' statement
  {
    $$ = &IfStatement{ pos: $1.pos, Condition: $3, TrueStatement: $5 }
  }
  | IF '(' expression ')' statement ELSE statement
  {
    $$ = &IfStatement{ pos: $1.pos, Condition: $3, TrueStatement: $5, FalseStatement: $7 }
  }
  | WHILE '(' expression ')' statement
  {
    $$ = &WhileStatement{ pos: $1.pos, Condition: $3, Statement: $5 }
  }
  | FOR '(' optional_expression ';' optional_expression ';' optional_expression ')' statement
  {
    $$ = &ForStatement{ pos: $1.pos, Init: $3, Condition: $5, Loop: $7, Statement: $9 }
  }
  | RETURN optional_expression ';'
  {
    $$ = &ReturnStatement{ pos: $1.pos, Value: $2 }
  }
```

parse.go で, ほかの部分から呼び出すための構文解析関数を Parse() を定義している. yacc から生成された関数を呼び出したり, エラー情報を適切にくっつけたりしている.

```
func Parse(src string) ([]Statement, error) {
  fset := token.NewFileSet()
  file := fset.AddFile("", fset.Base(), len(src))

  l := new(Lexer)
```

```
  l.Init(file, []byte(src), nil, scanner.ScanComments)
  yyErrorVerbose = true

  fail := yyParse(l)
  if fail == 1 {
    lineNumber, columnNumber := posToLineInfo(src, int(l.pos))
    err := fmt.Errorf("%d:%d: %s", lineNumber, columnNumber, l.errMessage)

    return nil, err
  }

  return l.result, nil
}
```

4 抽象構文木の変換処理

抽象構文木の変換処理は, parse.go の Walk() 関数に書いている.

構文解析部が返した抽象構文木を再帰的にたどっていき, 置き換えるべき表現を見つけたら変換した構造体を返すという処理をしている.

for 文を while 文に置き換える処理の例を示す.

```
func Walk(statement Statement) Statement {
  switch s := statement.(type) {
  case *ForStatement:
    // for (init; cond; loop) s
    // => init; while (cond) { s; loop; }
    body := Walk(s.Statement)
    return &CompoundStatement{
      Statements: []Statement{
        &ExpressionStatement{Value: s.Init},
        &WhileStatement{
          pos:       s.Pos(),
          Condition: s.Condition,
          Statement: &CompoundStatement{
            Statements: []Statement{
              body,
              &ExpressionStatement{Value: s.Loop},
            },
          },
        },
      },
```

```
}
```

5 main 関数

main 関数では, 構文解析関数 Parse() と抽象構文木を変換する関数 Walk() を呼び出して, 結果を出力する処理をしている. ソースファイルを見れば明らかだと思うので, ここでは説明を省略する.

6 感想

Go や型付きの言語での開発に不慣れなせいか, 抽象構文木の構造体をうまく定義するのに苦労した. 構文解析器を書いている途中で構造体の定義がよくないことがわかって何度も書き換えたりした.

はじめにオブジェクト指向っぽい空気感で書いていたところ, 何度も戸惑ったり痛い目にあったりした. Go は文法がオブジェクト指向言語っぽいが, 継承などの機能はないので実際はオブジェクト指向ではない.

もう少し気合を入れて開発することで, はやく Go に慣れていきたい.