計算機科学実験3 中間レポート2

杉本風斗

平成 28 年 5 月 12 日

1 概要

課題 9 の意味解析と課題 10 の中間表現への変換を実装した.
まずは具体的な実行例を説明する.

1.1 使い方

実行方法はレポート 1 で説明したのと同じである. 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, 中間表現を表す文字列が出力される.
簡単な Small C プログラムを与えて実行した例を示す.

```
$ cat example/0.sc
int main() {
  return 1 + 2;
}

$ ./small-c example/0.sc


main()
int #tmp_0
#tmp_0 = (+ 1 2)
return #tmp_0
```

出力の形式については中間表現への変換の項であとで説明する.

意味解析に失敗するソースコードを入力に与える例を示す.

```
$ cat type_error.sc
int main() {
  int a;
  int *p;
  p = a;
}

$ ./small-c type_error.sc
4:3: type error: int* = int


$ cat decl_error.sc
int main() {
  int a, b;
  int a;
}

$ ./small-c decl_error.sc
3:7: ‘a‘ is already defined
```

2 課題 9: 意味解析

2.1 オブジェクト情報の収集

サンプルプログラムで動作させた例を示す. 環境変数 DEBUG を設定すると, 解析した構文木の内容を pretty print して表示されるようになっている.

IdentifierExpression などの Symbol フィールドに型などのオブジェクト情報が埋め込まれていることが確認できる.

```
$ cat example/sum.sc
int sum(int a, int b) {
  return a + b;
}

int main() {
  print(sum(100, 20));
}

$ DEBUG=1 ./small-c example/sum.sc
...
[]main.Statement{
```

```
&main.FunctionDefinition{
  pos:        1,
  TypeName:   "void",
  Identifier: &main.IdentifierExpression{
    pos:    6,
    Name:   "print",
    Symbol: &main.Symbol{
      Name:  "print",
      Level: 0,
      Kind:  "proto",
      Type:  main.FunctionType{
        Return: main.BasicType{
          Name: "void",
        },
        Args: []main.SymbolType{
          main.BasicType{
            Name: "int",
          },
        },
      },
      Offset: 0,
    },
  },
  Parameters: []main.Expression{
    &main.ParameterDeclaration{
      pos:        12,
      TypeName:   "int",
      Identifier: &main.IdentifierExpression{
        pos:    16,
        Name:   "i",
        Symbol: (*main.Symbol)(nil),
      },
    },
  },
  Statement: nil,
},
&main.FunctionDefinition{
  pos:        1,
  TypeName:   "int",
  Identifier: &main.IdentifierExpression{
    pos:    5,
    Name:   "sum",
    Symbol: &main.Symbol{
```

```
    Name:  "sum",
    Level: 0,
    Kind:  "fun",
    Type:  main.FunctionType{
      Return: main.BasicType{
        Name: "int",
      },
      Args: []main.SymbolType{
        main.BasicType{
          Name: "int",
        },
        main.BasicType{
          Name: "int",
        },
      },
    },
    Offset: 0,
  },
},
Parameters: []main.Expression{
  &main.ParameterDeclaration{
    pos:        9,
    TypeName:   "int",
    Identifier: &main.IdentifierExpression{
      pos:    13,
      Name:   "a",
      Symbol: &main.Symbol{
        Name:  "a",
        Level: 1,
        Kind:  "parm",
        Type:  main.BasicType{
          Name: "int",
        },
        Offset: 0,
      },
    },
  },
  &main.ParameterDeclaration{
    pos:        16,
    TypeName:   "int",
    Identifier: &main.IdentifierExpression{
      pos:    20,
      Name:   "b",
```

```
      Symbol: &main.Symbol{
        Name:  "b",
        Level: 1,
        Kind:  "parm",
        Type:  main.BasicType{
          Name: "int",
        },
        Offset: 0,
      },
    },
  },
},
Statement: &main.CompoundStatement{
  pos:          23,
  Declarations: []main.Statement{},
  Statements:   []main.Statement{
    &main.ReturnStatement{
      pos:   23,
      Value: &main.BinaryExpression{
        Left: &main.IdentifierExpression{
          pos:    34,
          Name:   "a",
          Symbol: &main.Symbol{...},
        },
        Operator: "+",
        Right:    &main.IdentifierExpression{
          pos:    38,
          Name:   "b",
          Symbol: &main.Symbol{...},
        },
      },
      FunctionSymbol: &main.Symbol{
        Name:  "#func",
        Level: 1,
        Kind:  "",
        Type:  main.FunctionType{
          Return: main.BasicType{
            Name: "int",
          },
          Args: []main.SymbolType{...},
        },
        Offset: 0,
      },
```

```
      },
    },
  },
},
&main.FunctionDefinition{
  pos:        44,
  TypeName:   "int",
  Identifier: &main.IdentifierExpression{
    pos:    48,
    Name:   "main",
    Symbol: &main.Symbol{
      Name:  "main",
      Level: 0,
      Kind:  "fun",
      Type:  main.FunctionType{
        Return: main.BasicType{
          Name: "int",
        },
        Args: []main.SymbolType{},
      },
      Offset: 0,
    },
  },
  Parameters: []main.Expression{},
  Statement:  &main.CompoundStatement{
    pos:          55,
    Declarations: []main.Statement{},
    Statements:   []main.Statement{
      &main.ExpressionStatement{
        Value: &main.FunctionCallExpression{
          Identifier: &main.IdentifierExpression{
            pos:    59,
            Name:   "print",
            Symbol: &main.Symbol{...},
          },
          Argument: &main.FunctionCallExpression{
            Identifier: &main.IdentifierExpression{
              pos:    65,
              Name:   "sum",
              Symbol: &main.Symbol{...},
            },
            Argument: &main.ExpressionList{
              Values: []main.Expression{
```

```
                &main.NumberExpression{
                  pos:   69,
                  Value: "100",
                },
                &main.NumberExpression{
                  pos:   74,
                  Value: "20",
                },
              },
            },
          },
        },
      },
    },
  },
}

...
```

2.1.1 実装

オブジェクト情報の収集は, analyze.go の Analyze() 関数に実装されている.

Analyze() 関数は入力された抽象構文木を再帰的に辿り, オブジェクト情報の収集や式の形の検査を行う. 抽象構文木の関数定義ノードを解析する関数の例を説明する.

関数定義に含まれる関数名やパラメータを見て, オブジェクト情報を環境に登録したり, 型情報を取得する処理を行っている.

エラーを発見した場合, その場で処理を中断し例外を吐くことはしないで, エラーを配列に格納して解析処理を続けている. 複数のエラーがソースコードにあった場合に, 一度に見えるようにするためである. 出力側では特に工夫していないので, 大量の同じようなエラーが出力されることがあるが, とりあえずあまり気にしないことにしている.

また, 関数定義がプロトタイプ宣言でない場合には内容の statement が含まれるので, statement を解析する関数を呼び出している.

```
func analyzeFunctionDefinition(s *FunctionDefinition, env *Env) []error {
  errs := []error{}

  identifier := findIdentifierExpression(s.Identifier)
```

```
argTypes := []SymbolType{}

for _, p := range s.Parameters {
  parameter, ok := p.(*ParameterDeclaration)
  if ok {
    argType := BasicType{Name: parameter.TypeName}
    argTypes = append(argTypes, composeType(parameter.Identifier, argType))
  }
}

returnType := BasicType{Name: s.TypeName}
symbolType := FunctionType{Return: returnType, Args: argTypes}

kind := ""
if s.Statement != nil {
  kind = "fun"
} else {
  kind = "proto"
}

err := env.Register(identifier, &Symbol{
  Kind: kind,
  Type: symbolType,
})

if err != nil {
  errs = append(errs, SemanticError{
    Pos: s.Pos(),
    Err: err,
  })
}

if s.Statement != nil {
  paramEnv := env.CreateChild()
  // Set special symbol to analyze function type
  paramEnv.Add(&Symbol{
    Name: "#func",
    Type: symbolType,
  })

  for _, p := range s.Parameters {
    parameter, ok := p.(*ParameterDeclaration)
```

```
      if ok {
        identifier := findIdentifierExpression(parameter.Identifier)
        argType := composeType(parameter.Identifier, BasicType{Name: parameter.TypeName})

        err := paramEnv.Register(identifier, &Symbol{
          Kind: "parm",
          Type: argType,
        })

        if err != nil {
          errs = append(errs, SemanticError{
            Pos: parameter.Pos(),
            Err: fmt.Errorf("parameter ‘%s‘ is already defined", identifier.Name),
          })
        }
      }
    }

    errs = append(errs, analyzeStatement(s.Statement, paramEnv)...)
  }

  return errs
}
```

2.2 重複定義の検査，式の形の検査

重複定義の検査，式の形の検査は, オブジェクト情報の収集と同じ Analyze() 関数で行っている. オブジェクト情報の収集と同じであるので, 実装の説明は省略して, 実行例をしめす.

```
$ cat error.sc
int f() {}

int main() {
  int a, b, a;
  int *p;
  f = 100;
  p = &(b + 10);
}

$ ./small-c error.sc
4:13: ‘a‘ is already defined
5:3: ‘f‘ is not variable
```

```
5:3: ‘f‘ is not variable
6:9: the operand of ‘&‘ must be on memory
```

2.3 型検査

型検査の実行例は概要に示したとおりである.

型検査は type.go に実装されている. main からは CheckType() 関数を呼び出している. CheckType() は解析済みの抽象構文木を受け取って, 型エラーがあった場合エラーオブジェクトを返す.

Analyze() 関数と同様に, 再帰的に構文木を辿って, 埋め込まれたオブジェクト情報から型を順番に調べている.

```
func CheckType(statements []Statement) error {
  for _, s := range statements {
    err := CheckTypeOfStatement(s)
    if err != nil {
      return err
    }
  }

  return nil
}
```

3 課題 10: 中間表現への変換

実行例は概要で示したとおりである.

中間表現への変換処理は ir.go に実装されている. main からは CompileIR() を呼び出している. CompileIR() は解析済みの抽象構文木を受け取って, 中間表現プログラムの構造体を返す.

3.1 中間表現の構造体の定義

中間表現の構造体定義の例を示す. 講義資料の説明に沿う形で定義している.

```
type IRProgram struct {
  Declarations []*IRVariableDeclaration
  Functions []*IRFunctionDefinition
}
```

```
type IRStatement interface {
  String() string
}
type IRExpression interface {
  String() string
}

type IRVariableDeclaration struct {
  Var *Symbol
}

type IRFunctionDefinition struct {
  Var *Symbol
  Parameters []*IRVariableDeclaration
  Body IRStatement
  VarSize int
}

type IRCompoundStatement struct {
  Declarations []*IRVariableDeclaration
  Statements []IRStatement
}

type IRAssignmentStatement struct {
  Var *Symbol
  Expression IRExpression
}
```

3.2 中間表現の文字列出力

中間表現の構造体には, 出力用に簡単な文字列に変換する String() 関数が実装されている.

たとえば, プログラムを実行した際にはこのように表示される. コード中にコメントをつけて説明する.

文字列出力では, 複合文の表示は省略されている. 複合文を表示してしまうととても読みにくくなってしまうためである.

```
sum(int a, int b) // 関数の定義
int #tmp_0        // 変数定義. #がついた名前はコンパイラが実装の都合で生成した一時変数を表している
#tmp_0 = (+ a b)  // 式は S 式で表現している
```

```
return #tmp_0

main()
int #tmp_2
int #tmp_3
int #tmp_4
int #tmp_1
#tmp_2 = 100
#tmp_3 = 20
#tmp_4 = sum(#tmp_2, #tmp_3) // 関数呼び出し
#tmp_1 = #tmp_4
print(#tmp_1)
```

3.3 変換処理の実装

抽象構文木を再帰的に辿り, 文や式を変換する実装になっている.

文を変換するとき, 一時変数が新たに必要になった場合には, その一時変数の定義を含んだ複合文に変換している.

式を変換するときは, 式を複合文をそのまま変換することはできないので, 式を中間表現に変換した結果とともに必要な一時変数の宣言のリストを返している. 式で必要な一時変数は, 呼び出し元の文の変換処理で複合文にまとめて格納される.

If 文の中間表現への変換を例に示す.

tmpvar() は unique な名前の一時変数を返し, label() はジャンプラベル用の unique な文字列を返す.

条件式を条件判定用の一時変数に格納するようにし, 中間表現の制御構文を使って中間表現の列に変換している. 条件式用に一時変数が必要になるので, その一時変数の宣言を含めた複合文に変換して返すようにしている.

```
func compileIRStatement(statement Statement) IRStatement {
...
  case *IfStatement:
    conditionVar := tmpvar()

    trueLabel := label("true")
    falseLabel := label("false")
    endLabel := label("end")
```

```
    condition, decls, beforeCondition := compileIRExpression(s.Condition)

    statements := []IRStatement{
      &IRAssignmentStatement{
        Var: conditionVar,
        Expression: condition,
      },
      &IRIfStatement{
        Var: conditionVar,
        TrueLabel: trueLabel,
        FalseLabel: falseLabel,
      },
      &IRLabelStatement{ Name: trueLabel },
      compileIRStatement(s.TrueStatement),
      &IRGotoStatement{ Label: endLabel },
      &IRLabelStatement{ Name: falseLabel },
    }

    if s.FalseStatement != nil {
      statements = append(statements, compileIRStatement(s.FalseStatement))
    }

    statements = append(statements, &IRLabelStatement{ Name: endLabel })

    return &IRCompoundStatement{
      Declarations: append(IRVariableDeclarations([]*Symbol{conditionVar}), decls...),
      Statements: append(beforeCondition, statements...),
    }
...
}
```

4 感想と工夫した点

デバッグの際に正しい中間表現が生成されていることを確認するのに苦労した.

中間表現の変換では一時変数を大量に使うので, 変換結果の中間表現は複合文が大量にネストしたオブジェクトになってしまい, そのまま表示するとかなり読みにくい.

そこで, 中間表現を簡単な文字列に変換する処理をつくったところ, デバッグがかなりやりやすくなった. 実装がちょっとめんどくさい気がしたけど, 結果的には時間の節約になって嬉しかった.