به نام خدا

ایجاد تحلیلگر لغوي و نحوي

با استفاده از

GoldParser

شرح کلاسها به زبان Delphi

استاد راهنما : دکتر سعید پارسا

تهیه کنندگان:

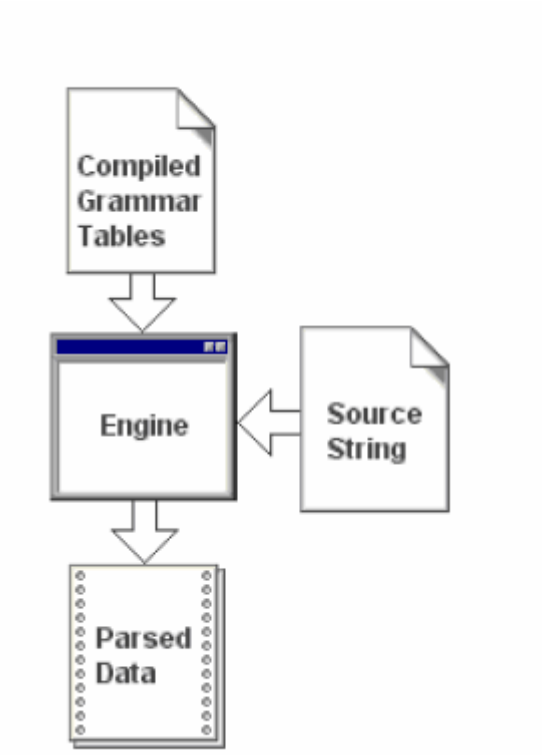
محمد حسن فلك‌مسیر

علي فاکري

حامد احمدي

مقدمه

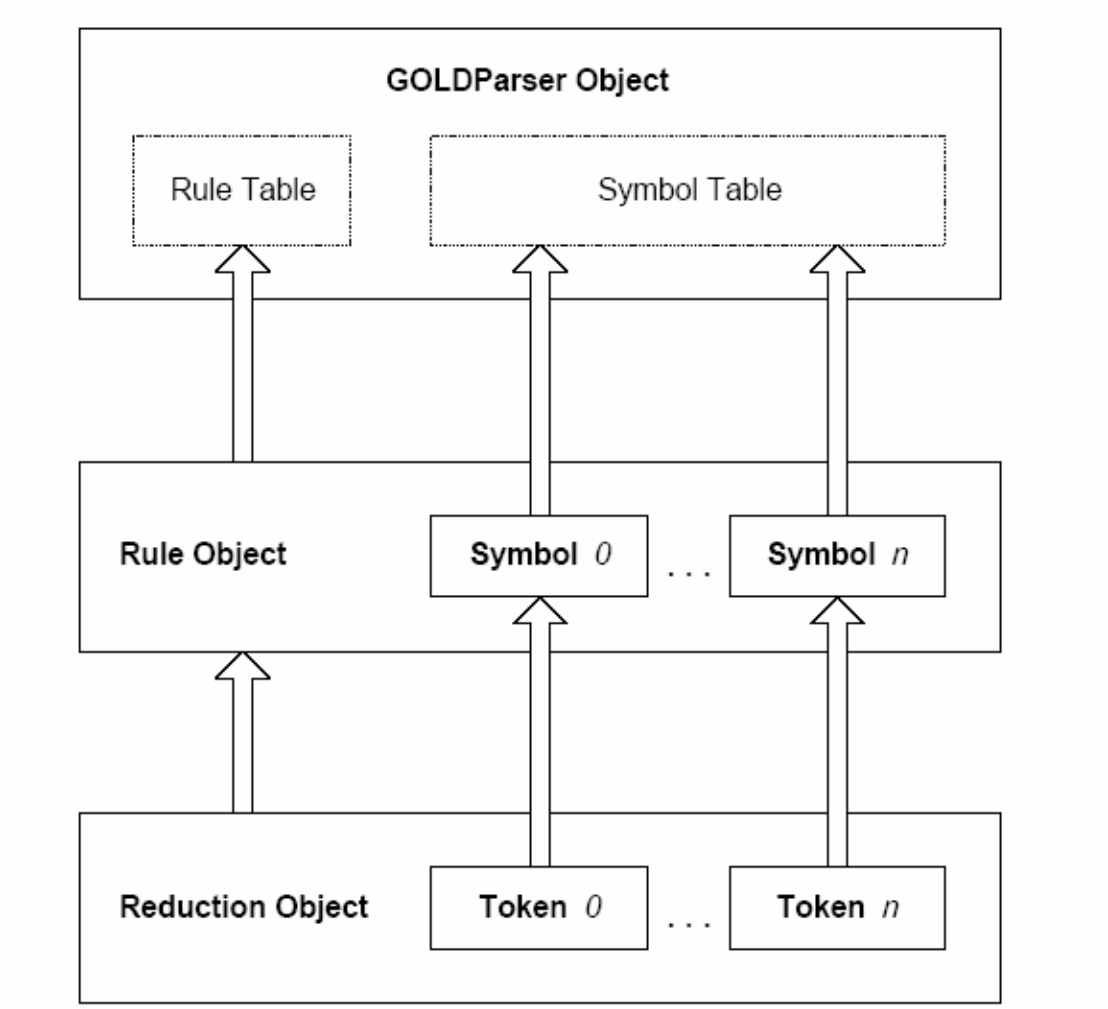
در زير مراحل توليد يك پارسر براي تحليل لغوي و نحوي بوسيله Delphi با استفاده از GoldParser Package را مشاهده مي‌كنيد. براي اين كار مي‌توانيد از كلاس GoldParser يك نمونه در برنامه خود تعريف كنيد و از پيغام هاي ارسالي پس از هر بار فراخواني متد Parse استفاده كنيد ، يا اينكه كلاس مورد نظر خود براي Parse را از كلاس TGoldParser مشتق نماييد. چون GoldParser يك مولد پارسر OpenSource مي‌باشد مي‌توانيد كلاسهاي موجود در آنرا بهبود بخشيده و امكانات جديد از جمله روش هايي بهينه تر براي بازگشت از خطا يا كلاسهاي جديد براي ايجاد كد از جداول توليد شده، در حين مراحل پارس توسعه دهيد.



به طور خلاصه GoldParser با استفاده از يك گرامر با Syntax مخصوص GoldParser و انجام مراحل تحليلي معين روي آن كليه جداول از جمله جدول LALR(1) و ماشين DFA را ايجاد و در حافظه Load مي‌كند. اين جدول قابل ذخيره سازي براي استفاده در آينده نيز مي باشد. براي شروع عمل تحليل مي‌توان گرامر با Syntax مخصوص GoldParser را با استفاده از كلاس GrammarReader كامپايل كرد يا اينكه بوسيله محيط GoldParser عمل كامپايل را انجام داد و سپس با فراخواني متد LoadCompiledGrammar و دادن مسير فايل گرامر كامپايل شده جداول را در حافظه Load كرد. پس از اين كار با استفاده از متد OpenTextString ورودي مورد نظر براي تحليل را به كلاس GoldParser مي‌دهيم سپس با فراخواني متد Parse و طي مراحل پارس تا زمان پذيرش كامل ورودي يا رخ دادن خطا عمل تحليل نحوي و لغوي را انجام مي‌دهيم.

در زير به شرح كلاس هاي مورد نياز براي مراحل تحليل نحوي و لغوي توسط GoldParser كه به زبان Delphi نوشته شده اند مي‌پردازيم. مشابه اين كلاسها براي ساير زبانها و

محيطهاي برنامه سازي نيز موجود مي‌باشد وليكن ابتدا به تشريح مختصر عملكرد GoldParser و سلسله مراتب كلاسها به همراه نحوه همكاري آنها مي‌پردازيم.



پياده سازي GoldParse به زبان Delphi از ۵ كلاس مختلف تشكيل شده كه براي ارائه اطلاعات ذخيره شده در جداول موجود در گرامر كامپايل شده و اطلاعاتي كه در زمان Parse ايجاد مي‌شود به كار مي‌رود. همانطور كه اشاره شد عمليات اصلي بوسيله كلاس GoldParserانجام مي‌شود. اين كلاس حاوي جدول قوانين و جدول نمادها (Rule Table و Symbol Table) مي‌باشد.

در زمان پارس كلاس GoldParser نمونه هاي موجود از كلاس Reduction و Token در خود را ايجاد و مقدار دهي مي‌كند. كلاس Rduction حاوي يك سري از Token هايي كه مربوط به قانون reduce داده شده مي‌باشد.

TGoldParser

كلاس اصلي به كار گرفته شده توسط GoldParser است كه قابليت انجام تمامي عمليات لازم براي پارس كردن يك فايل متني را دارا مي‌باشد.

اين كلاس شامل :

۱. كد ايجاد جدول LALR(1)
۲. كد ماشين DFA
۳. جدول كاراكترها
۴. و ساير ساختمان داده هاي لازم براي انجام عمل پارس مي‌باشد

در زير به تعريف كلاس مي‌پردازيم

```
Type
 TGOLDParser = class
 Private
  FGrammarReader: TObject;
  FVariableList: TVariableList;
  FSymbolTable: TSymbolTable;
  FInitialDFAState: Integer;
  FInitialLALRState: Integer;
  FCharacterSetTable: TStringList;
  FRuleTable: TRuleTable;
  FDFA: TFStateTable;
  FActionTables: TLRActionTables;

  FTablesLoaded: Boolean;
  FTrimReductions: Boolean;

  FErrorSymbol: TSymbol;
  FEndSymbol: TSymbol;

  FCompareMode: TCompareMode;

  FCurrentLALR: Integer;
  FLineNumber: Integer;
  FCommentLevel: Integer;
```

```
  FHaveReduction: Boolean;

  FStack: TTokenStack;
  FTokens: TTokenStack;
  FInputTokens: TTokenStack;

  FSource: TSourceFeeder;
  procedure PrepareToParse;
  function RetrieveToken(Source: TSourceFeeder): TToken;
  procedure DiscardRestOfLine;
  function ParseToken(NextToken: TToken): Integer;
  function GetCurrentReduction: TReduction;
  procedure SetCurrentReduction(NewReduction: TReduction);
  function GetCurrentToken: TToken;
 public
  constructor Create;
  destructor Destroy; override;
  procedure Reset;
  // Resets the GOLDParser. The parser's internal tables are not affected.
  procedure Clear;
  // The GOLDParser is reset and the internal tables are cleared.
  procedure PushInputToken(TheToken: TToken);
  // Pushes the token onto the front of the GOLDParser's internal input queue.
  // It will be the next token analyzed by the parsing engine.
  function LoadCompiledGrammar(const FileName : string) : boolean; overload;
  function LoadCompiledGrammar(const Stream : TStream) : boolean; overload;
  // If the Compiled Grammar Table file is successfully loaded
  // the method returns True; otherwise False. This method must
  // be called before any Parse calls are made.
  function OpenTextString(Text: String): Boolean;
```

// Opens the SourceString for parsing. If successful the method returns True; otherwise False.

function ReadTextString: string;

function Parse: Integer;

// Executes a parse. When this method is called, the parsing engine

// reads information from the source text (either a string or a file)

// and then reports what action was taken. This ranges from a token

// being read and recognized from the source, a parse reduction, or a type of error.

function Parameter(ParamName: string): string;

// Returns a string containing the value of the specified parameter.

// The ParameterName is the same as the parameters entered in the

// grammar's description. These include: Name, Version, Author, About,

// Case Sensitive and Start Symbol. If the name specified is invalid,

// this method will return an empty string.

function PopInputToken: TToken;

// Removes the next token from the front of the parser's internal input queue.

property TrimReductions: Boolean read FTrimReductions write FTrimReductions;

// Returns/sets the TrimReductions flag. When this property is set to True,

// the parser engine will automatically trim (i.e. remove) unneeded reductions

// from the parse tree. For more information please click here.

property CurrentReduction: TReduction read GetCurrentReduction write SetCurrentReduction;

// Returns/sets the reduction made by the parsing engine.

// When a reduction takes place, this property will be set to

// a Reduction object which will store the reduced rule and its related tokens.

// This property may be reassigned a customized object if the developer so desire.

```
  // The value of this property is only valid when the Parse() method returns
  //the gpMsgReduction message.
  property CurrentLineNumber: Integer read FLineNumber;
  // Returns the current line in the source text.
  property CurrentToken: TToken read GetCurrentToken;
  // Returns the token that is ready to be parsed by the engine.
  // This property is only valid when when the gpMsgTokenRead message is
  // returned from the Parse method.
  property VariableList : TVariableList read FVariableList;
  property SymbolTable : TSymbolTable read FSymbolTable;
  // Returns the parser's internal Symbol Table.
  property CharacterSetTable : TStringList read FCharacterSetTable;
  property RuleTable : TRuleTable read FRuleTable;
  // Returns the parser's Rule Table
  property DFA : TFStateTable read FDFA;
  property ActionTables : TLRActionTables read FActionTables;
  property InitialDFAState : integer read FInitialDFAState write FInitialDFAState;
  property InitialLALRState : integer read FInitialLALRState write FInitialLALRState;
  property TokenTable : TTokenStack read FTokens;
  // Returns the token Table.
 end;
```

براي شروع كار با اين كلاس ابتدا گرامر كامپايل شده را با استفاده از متد LoadCompiledGrammar به اين كلاس مي‌دهيم سپس براي پارس كردن و تحليل لغوي و نحوي ورودي در هر مرحله تابع Parse را از نمونه ايجاد شده از كلاس صدا مي‌زنيم با اجراي اين تابع مراحل تحليل لغوي و نحوي تا پردازش و reduce داده شدن يك قانون انجام ميگردد و در نهايت پيامي مبتني بر اطلاعات reduce انجام شده از جمله Token هاي خوانده شده و خود قانون مورد استفاده براي reduce را داخل CurrentRule و CurrentReduction و CurrentToken اعلام مي‌گردد. در زير به شرح اين پيام‌ها مي‌پردازيم.

```
  gpMsgEmpty = 0;      // Nothing
  gpMsgTokenRead = 1;   // Each time a token is read, this message is
generated.
  gpMsgReduction = 2;
          // When the engine is able to reduce a rule,
        // this message is returned. The rule that was
        // reduced is set in the GOLDParser's ReduceRule property.
        // The tokens that are reduced and correspond the
        // rule's definition are stored in the Tokens() property.
  gpMsgAccept = 3;  // The engine will returns this message when the
source
             // text has been accepted as both complete and correct.
             // In other words, the source text was successfully analyzed.
  gpMsgNotLoadedError = 4;    // Before any parsing can take place,
                  // a Compiled Grammar Table file must be loaded.
  gpMsgLexicalError = 5;      // The tokenizer will generate this message
when
                  // it is unable to recognize a series of characters
                  // as a valid token. To recover, pop the invalid
                  // token from the input queue.
  gpMsgSyntaxError = 6;
             // Often the parser will read a token that is not expected
        // in the grammar. When this happens, the Tokens() property
        // is filled with tokens the parsing engine expected to read.
        // To recover: push one of the expected tokens on the input
queue.
  gpMsgCommentError = 7;
        // The parser reached the end of the file while reading a
comment.
        // This is caused when the source text contains a "run-away"
        // comment, or in other words, a block comment that lacks the
        // delimiter.
  gpMsgInternalError = 8;          // Something is wrong, very wrong
```

اين پيغام ها نيز پس از پايان عمل پارس توليد مي‌گردند.

```
  ParseResultEmpty = 0;
  ParseResultAccept = 1;
  ParseResultShift = 2;
  ParseResultReduceNormal = 3;
  ParseResultReduceEliminated = 4;
  ParseResultSyntaxError = 5;
  ParseResultInternalError = 6;
```

TSymbol

اين كلاس براي ذخيره سازي عبارات پاياني، غير پاياني و پاياني ويژه كه بوسيله ماشين DFA و LALR پارسر به كار مي‌روند كاربرد دارد. سمبول ها هم مي‌توانند عبارات پاياني باشند مانند شناسه ها و هم مي‌توانند عبارات غير پاياني مانند قوانين و ساختارهاي برنامه باشند. نمادهاي پاياني كه به وسيله GoldParser به كار مي‌روند در به شرح ذيل مي‌باشند.

```
const

  SymbolTypeNonterminal = 0; / / Normal nonterminal
  SymbolTypeTerminal = 1;   / / Normal terminal
  SymbolTypeWhitespace = 2; / / This Whitespace symbols is a special
terminal
                / / that is automatically ignored the the parsing engine.
                / / Any text accepted as whitespace is considered
                / / to be inconsequential and "meaningless".
  SymbolTypeEnd = 3;      / / The End symbol is generated when the
tokenizer
                / / reaches the end of the source text.
  SymbolTypeCommentStart = 4;
       / / This type of symbol designates the start of a block quote.
  SymbolTypeCommentEnd = 5;
       / / This type of symbol designates the end of a block quote.
```

```
  SymbolTypeCommentLine = 6;
        // When the engine reads a token that is recognized as
    // a line comment, the remaining characters on the line
    // are automatically ignored by the parser.
  SymbolTypeError = 7;
        // The Error symbol is a general-purpose means
    // of representing characters that were not recognized
    // by the tokenizer. In other words, when the tokenizer
    // reads a series of characters that is not accepted
    // by the DFA engine, a token of this type is created.
```

حال به تعريف كلاس TSymbol مي‌پردازيم

```
  TSymbol = class
  private
    FName: String;
    FKind: Integer;
    FTableIndex: Integer;
    function GetText: string;
    function PatternFormat(Source: string): string;
  public
    constructor Create(aTableIndex : integer; aName : string; aKind :
integer);
    property Kind: Integer read FKind;
    // Returns an enumerated data type that denotes
    // the class of symbols that the object belongs to.
    property TableIndex: Integer read FTableIndex;
    // Returns the index of the symbol in the GOLDParser object's
Symbol Table.
    property Name: string read FName;
    // Returns the name of the symbol.
    property Text: string read GetText;
    // Returns the text representation of the symbol.
    // In the case of nonterminals, the name is delimited by angle
brackets,
```

```
    / / special terminals are delimited by parenthesis
    / / and terminals are delimited by single quotes (if special characters
are present).
  end;
```

كلاس ديگري كه براي ذخيره سازي جدول نمادها و مديريت اعمال مرتبط با آن به كار مي‌رود TSymbolTable نام دارد كه در زير به تعريف آن مي‌پردازيم.

```
 TSymbolTable = class
 private
  FList : TObjectList;
  function GetCount: integer;
  function GetItem(Index: integer): TSymbol;
  procedure SetItem(Index: integer; const Value: TSymbol);
 public
  constructor Create;
  destructor Destroy; override;
  procedure Add(Value : TObject);
  procedure Clear;
  property Count : integer read GetCount;
  property Items[Index : integer] : TSymbol read GetItem write
SetItem; default;
 end;
```

علاوه بر نگهداري جدول نمادها GoldParser در ضمن مراحل تحليل لغوي متغيرهاي تعريف شده را داخل كلاسي به نام TVariableList نگهداري مي‌كند. در زير به تعريف اين كلاس مي‌پردازيم.

```
type
 TVariableList = class
 private
  MemberList: TStringList;
  function GetCount: Integer;
  function GetValue(Name: string): string;
```

```
  procedure SetValue(Name: string; Value: string);
  function GetName(Index: Integer): string;
public
  constructor Create;
  destructor Destroy; override;
  procedure Add(Name: string; Value: string);

  procedure Clear;
  property Count: Integer read GetCount;
  property Value[Name: string]: string read GetValue write SetValue;
  property Names[Index : integer] : string read GetName;
end;
```

درحالي كه كلاس Symbol يك مجموعه از عبارات پاياني و غير پاياني را ارائه مي‌كند، كلاس Token يك سري اطلاعات كه در زمان پارس معين شده اند را در مورد اين عبارات را نگهداري و ارائه مي‌كند. براي مثال يك كلاس Symbol ممكن است از يك نوع مشخص مثل identifier باشد ولي يك Token از نوع identifier ممكن است در صورتهاي مختلف و با مقاديري مانند Value1 يا F1 و ... وجود داشته باشد. در زير به تعريف كلاس Token مي‌پردازيم.

```
TToken = class
private
  FState: Integer;
  FDataVar: string;
  FReduction: TReduction;
  FParentSymbol: TSymbol;
  FOwnerStack : TTokenStack;
  function GetKind: Integer;
  function GetName: string;
  procedure SetParentSymbol(Value: TSymbol);
  procedure SetdataVar(Value: string);
  procedure SetReduction(Value: TReduction);
  function GetTableIndex: Integer;
  function GetText: string;
  procedure SetState(Value: Integer);
```

```
public
  constructor Create;
  destructor Destroy; override;
  property Kind : Integer read GetKind;
    // Returns an enumerated data type that denotes the symbol class of the token.
  property Name : string read GetName;
    // Returns the name of the token. This is equivalent to the parent symbol's name.
  property ParentSymbol : TSymbol read FParentSymbol write SetParentSymbol;
    // Returns a reference the token's parent symbol.
  property DataVar : string read FDataVar write SetDataVar;
    // Returns/sets the information stored in the token.
    // This can be either an standard data type or an object reference.
  property Reduction : TReduction read FReduction write SetReduction;
  property TableIndex : Integer read GetTableIndex;
    // Returns the index of the token's parent symbol in the GOLDParser object's symbol table.
  property Text : string read GetText;
    // Returns the text representation of the token's parent symbol.
    // In the case of nonterminals, the name is delimited by angle brackets,
    // special terminals are delimited by parenthesis and terminals
    // are delimited by single quotes (if special characters are present).
  property State : Integer read FState write SetState;
end;
```

GoldParser براي سهولت در استفاده از Token هاي شناسايي شده آنها را داخل Stack نگهداري مي‌كند و براي سهولت دسترسي به اين Token ها از كلاس TTokenStack استفاده مي‌كند. در زر به تعريف كلاس TokenStack مي‌پردازيم.

```
TTokenStack = class
private
  MemberList: TObjectList;
```

```
  OwnedTokens : TObjectList;
  function GetCount: Integer;
  function GetItem(Index: Integer): TToken;
  procedure FreeOwnedTokens;
public
  constructor Create;
  destructor Destroy; override;
  procedure Clear;
  procedure Push(TheToken: TToken);
  function Pop: TToken;
  function Top: TToken;
  property Count: Integer read GetCount;
  property Items[Index: Integer]: TToken read GetItem; default;
end;
```

GoldParser براي سهولت در مطالعه مراحل پارس از كلاسي با نام TReduction براي نگهداري مسير طي شده تا Reduce دادن نماد آغازگر استفاده مي‌كند. GoldParser هنگامي كه Token هاي لازم براي reduce دادن يك عبارت توسط يك قانون را خواند. اطلاعات reduce جاري را داخل نمونه كلاس Reduction موجود در خود نگهداري مي‌كند. اين كلاس شامل تمامي Token هاي rerduce داده شده به همراه مقادير آنها و ساير اطلاعات مي‌باشد. با مشاهده پيام gpMsgReduce در خروجي تابع Parse از كلاس GoldParser مي‌توان از اطلاعات آخرين Reduction كه در داخل خاصيت CurrentReduction از كلاس GoldParser ايجاد شده است استفاده كرد. با استفاده از اين نمونه كه در كلاس TGoldParser تعريف شده است ، مي‌توان تمامي مراحل تجزيه پائين به بالا كه توسط GoldParser انجام شده را پس از پذيرش ورودي مشاهده كرد. از روي اين نمونه است كه مي‌توان درخت پارس را ايجاد نمود. در ذيل به تعريف اين كلاس خواهيم پرداخت.

```
TReduction = class
private
  FTokens: TObjectList;
  FParentRule: TRule;
  FTag: Integer;
  function GetToken(Index: Integer): TToken;
```

```
  function GetTokenCount: Integer;
  procedure SetTag(const Value: Integer);
 public
  constructor Create(const aParentRule : TRule);
  destructor Destroy; override;
  procedure InsertToken(Index : integer; Token : TToken);
  property ParentRule: TRule read FParentRule;
  property TokenCount: Integer read GetTokenCount;
  property Tag: Integer read FTag write SetTag;
  property Tokens[Index: Integer]: TToken read GetToken;
 end;
```

كلاس‌ها و مراحلي كه در بالا به شرح آنها پرداختيم كلاس‌ها و توابع مورد استفاده در عمل تحليل لغوي و نحوي بودند. حال مي‌خواهيم به شرح مراحل تبديل گرامر به جدول LALR(1) و ماشين DFA بپردازيم.

GoldParser براي خواندن و ذخيره اطلاعات گرامر از كلاس GrammarReader استفاده مي‌كند. اين كلاس با خواندن گرامر به فرم GoldParser (كه در ادامه شرح داده خواهد شد) جدول نمادها و DFA و جدول LALR را ايجاد مي‌كند (بوسيله متد DoLoadTables) و همگي آنها به صورت يك فايل با پسوند cgt. ذخيره مي‌كند. اين كار براي هر گرامر مورد استفاده‌اي بايد انجام شود. و كلاس GoldParser از اين گرامر پردازش شده (با پسوند cgt.) براي پارس كردن استفاده مي‌كند. در زير به تعريف كلاس مي‌پردازيم.

```
type

 TGrammarReader = class
 private
  FBufferPos: Integer;
  FBuffer: string;
  FCurrentRecord: Variant;
  FEntryPos: Integer;
  FEntryCount: Integer;
  FStartSymbol: Integer;
  FParser : TGoldParser;
  function ReadUniString: string;
```

```
    function ReadInt16: Integer;
    function ReadByte: Char;
    function ReadEntry: Variant;
    function OpenFile(const FileName: string): boolean;
    function OpenStream(const Stream : TStream) : boolean;
  protected
    function DoLoadTables : boolean;
  public
    constructor Create(aParser : TGoldParser);
    destructor Destroy; override;
    function GetNextRecord: Boolean;
    function RetrieveNext: Variant;
    function LoadTables(const FileName : string): boolean; overload;
    function LoadTables(const Stream : TStream): boolean; overload;
    property Buffer: string read FBuffer;
    property StartSymbol: Integer read FStartSymbol write FStartSymbol;
    property Parser : TGOLDParser read FParser;
  end;
```

حال مي‌توانيد مراحل تشکيل جداول در متد DoLoadTables را ببينيد.

```
function TGrammarReader.DoLoadTables: Boolean;
var Id : String;
    iDummy1, iDummy2, iDummy3, i : integer;
    strDummy : string;
    NewSymbol : TSymbol;
    NewRule : TRule;
    NewFAState : TFAState;
    bAccept : boolean;
    NewActionTable : TLRActionTable;
begin
  Parser.VariableList.Add('Name', '');
  Parser.VariableList.Add('Version', '');
  Parser.VariableList.Add('Author', '');
```

```
 Parser.VariableList.Add('About', '');
 Parser.VariableList.Add('Case Sensitive', '');
 Parser.VariableList.Add('Start Symbol', '');

 Result := False;

 if FHeader = ReadUniString then begin
  while (FBufferPos < Length(FBuffer)) do begin
   Result := GetNextRecord;
   Id := String(RetrieveNext);
   case Ord(Id[1]) of
    RecordIdParameters : begin
     Parser.VariableList.Value['Name'] := RetrieveNext;
     Parser.VariableList.Value['Version'] := RetrieveNext;
     Parser.VariableList.Value['Author'] := RetrieveNext;
     Parser.VariableList.Value['About'] := RetrieveNext;
     Parser.VariableList.Value['Case Sensitive'] := RetrieveNext;
     Parser.VariableList.Value['Start Symbol'] := RetrieveNext;
     FStartSymbol := StrToInt(Parser.VariableList.Value['Start
Symbol']);
    end;
    RecordIdTableCounts : begin
     RetrieveNext;
          // for i := 0 to RetrieveNext - 1 do
      Parser.SymbolTable.Add(nil);
     for i := 0 to RetrieveNext do Parser.CharacterSetTable.Add('');
     RetrieveNext; // for i := 0 to RetrieveNext do
Parser.RuleTable.Add(nil);
     RetrieveNext; // for i := 0 to RetrieveNext do Parser.DFA.Add(nil);
     RetrieveNext;
          // for i := 0 to RetrieveNext do Parser.ActionTable.Add(nil,
          0, 0);
    end;
    RecordIdInitial : begin
     Parser.InitialDFAState := RetrieveNext;
```

```
      Parser.InitialLALRState := RetrieveNext;
    end;
    RecordIdSymbols : begin
      iDummy1 := RetrieveNext;
      strDummy := RetrieveNext;
      iDummy2 := RetrieveNext;

      NewSymbol := TSymbol.Create(iDummy1, strDummy, iDummy2);

      RetrieveNext;
      Parser.SymbolTable.Items[NewSymbol.TableIndex]          :=
NewSymbol;
    end;
    RecordIdCharSets : begin
      iDummy1 := RetrieveNext;
      Parser.CharacterSetTable.Strings[iDummy1] := RetrieveNext;
    end;
    RecordIdRules : begin
      iDummy1 := RetrieveNext;
      NewRule:=TRule.Create(iDummy1,
Parser.SymbolTable.Items[RetrieveNext]);
      RetrieveNext;
      while FEntryPos <= FEntryCount do
        NewRule.AddItem(Parser.SymbolTable.Items[RetrieveNext]);
      Parser.RuleTable.Items[NewRule.TableIndex] := NewRule;
    end;
    RecordIdDFAStates : begin
      NewFAState := TFAState.Create;
      iDummy1 := RetrieveNext;
      bAccept := RetrieveNext;
      if bAccept then NewFAState.AcceptSymbol := RetrieveNext
      else begin
        NewFAState.AcceptSymbol := -1;
        RetrieveNext;
      end;
```

```
        RetrieveNext;
        while FEntryPos <= FEntryCount do begin
          strDummy := RetrieveNext;
          iDummy2 := RetrieveNext;
          NewFAState.AddEdge(strDummy, iDummy2);
          RetrieveNext;
        end;
        Parser.DFA.Items[iDummy1] := NewFAState;
      end;
      RecordIdLRTables : begin
        NewActionTable := TLRActionTable.Create;
        i := RetrieveNext;
        RetrieveNext;
        while FEntryPos <= FEntryCount do begin
          iDummy1 := RetrieveNext;
          iDummy2 := RetrieveNext;
          iDummy3 := RetrieveNext;
          NewActionTable.Add(Parser.SymbolTable[iDummy1],   iDummy2,
iDummy3);
          RetrieveNext;
        end;
        Parser.ActionTables.Items[i] := NewActionTable;
      end;
    end;
  end;
 end;
 Parser.VariableList.Value['Start mbol']:=
Symbol(Parser.SymbolTable.Items[StrToInt(Parser.VariableList.Value['S
tartSmbol'])]).Name;
end;
```

همانطور كه مشاهده مي‌كنيد اين كلاس از دو كلاس TFAStateTable و TLRActionTable براي ايجاد DFA و جدول LALR استفاده مي‌كند. تعريف اين كلاس‌ها به شرح زير مي‌باشد.

```
type

 TFAEdge = class
 private
  FCharacters: Integer;
  FTargetIndex: Integer;
  procedure SetCharacters(const Value: integer);
  procedure SetTargetIndex(const Value: integer);
 public
  property Characters : integer read FCharacters write SetCharacters;
  property TargetIndex : integer read FTargetIndex write
SetTargetIndex;
 end;

 TFAState = class
 private
  FEdges: TObjectList;
  FAcceptSymbol: Integer;
  function GetEdgeCount: Integer;
  function GetEdge(Index: Integer): TFAEdge;
  procedure SetAcceptSymbol(const Value: Integer);
  procedure Add(Characters, Target : integer);
 public
  constructor Create;
  destructor Destroy; override;
  procedure AddEdge(Characters: string; Target: Integer);

  property AcceptSymbol: Integer read FAcceptSymbol write
SetAcceptSymbol;
  property EdgeCount: Integer read GetEdgeCount;
  property Edges[Index : integer] : TFAEdge read GetEdge; default;
 end;

 TFStateTable = class
 private
```

```
  FList : TObjectList;
  function GetCount: integer;
  function GetItem(Index: integer): TFAState;
  procedure SetItem(Index: integer; const Value: TFAState);
 public
  constructor Create;
  destructor Destroy; override;
  procedure Add(Value : TObject);

  property Count : integer read GetCount;
  property Items[Index : integer] : TFAState read GetItem write
SetItem; default;
 end;

type

 TLRAction = class
 private
  FSymbol: TSymbol;
  FAction: Integer;
  FValue: Integer;
  function GetSymbol: TSymbol;
  function GetSymbolIndex: Integer;
 public
  property Value: Integer read FValue;
  property Action: Integer read FAction;
  property Symbol: TSymbol read GetSymbol;
  property SymbolIndex: Integer read GetSymbolIndex;
 end;

 TLRActionTable = class
 private
  FList : TObjectList;
  function GetCount: integer;
  function GetItem(Index: integer): TLRAction;
```

```
  procedure SetItem(Index: integer; const Value: TLRAction);
 public
  constructor Create;
  destructor Destroy; override;
  procedure Add(TheSymbol: TSymbol; Action: Integer; Value: Integer);
  function ActionIndexForSymbol(SymbolIndex: Integer): Integer;

  property Count : integer read GetCount;
  property Items[Index : integer] : TLRAction read GetItem write
SetItem; default;
 end;

 TLRActionTables = class
 private
  FList : TObjectList;
  function GetCount: integer;
  function GetItem(Index: integer): TLRActionTable;
  procedure SetItem(Index: integer; const Value: TLRActionTable);
 public
  constructor Create;
  destructor Destroy; override;

  property Count : integer read GetCount;
  property Items[Index : integer] : TLRActionTable read GetItem write
SetItem; default;
 end;
```

GoldParser براي دريافت گرامر Syntax مخصوصي دارد كه در زير به شرح آن مي‌پردازيم.

ابتدا به تعريف مشخصات گرامر مي‌پردازيم و از عبارات زير براي توصيف گرامر استفاده مي‌كنيم.

```
"Name"    = 'ANSI C'
"Version" = '0.1'
"Author"  = 'Ali Fakeri,Mohammad Falak masir, Hamed Ahmadi'
"About"   = 'we are the best'

"Case Sensitive" = True
"Start Symbol"   = <START>
```

سپس به تعريف مجموعه هاي موجود در گرامر مثل مجموعه اعداد مجموعه كاراكتر ها و ساير مجموعه ها به شكل زير مي‌پردازيم و از يك سري مجموعه موجود به صورت Default (مثل {Digit} {Letter} و {Printable}) براي توسعه تعريف استفاده مي‌كنيم.

```
{Hex Digit}     = {Digit} + [abcdefABCDEF]
{Oct Digit}     = [01234567]
{Id Head}       = {Letter} + [_]
{Id Tail}       = {Id Head} + {Digit}
{String Ch}     = {Printable} - ["]
{Char Ch}       = {Printable} - ['']
```

پس از تعريف مجموعه ها نوبت به تعريف عبارات پاياني مي‌رسد. براي تعريف عبارات پاياني از فرم خلاصه شده به صورت زير استفاده مي‌كنيم.

```
DecLiteral      = [123456789]{digit}*
OctLiteral      = 0{Oct Digit}*
HexLiteral      = 0x{Hex Digit}+
FloatLiteral    = {Digit}*'.'{Digit}+
StringLiteral   = '"'( {String Ch} | '\'{Printable} )* '"'
CharLiteral     = '' ( {Char Ch} | '\'{Printable} )''
Id              = {Id Head}{Id Tail}*
Comment Start   = '/*'
Comment End     = '*/'
Comment Line    = '//'
```

و در پايان به تعريف قوانين گرامر مي‌پردازيم.

در زير نمونه قوانين طراحي شده براي پذيرش جملات به زبان C++ را مشاهده مي‌كنيد.

```
!================================================
Include  Declaration

<Inc Decl> ::= '#' include '<' Id '.' Id '>'

!================================================
Define  Declaration

<Def Decl> ::= '#' define <Value> <Value>

!================================================
Function  Declaration

<Func Proto> ::= <Func ID> '(' <Types>  ')' ';'
         | <Func ID> '(' <Params> ')' ';'
         | <Func ID> '(' ')' ';'

<Func Decl>  ::= <Func ID> '(' <Params>  ')' <Block>
         | <Func ID> '(' <Id List> ')' <Struct Def> <Block>
         | <Func ID> '(' ')' <Block>


<Params>    ::= <Param> ',' <Params>
         | <Param>

<Param>     ::= const <Type> ID
         |     <Type> ID


<Types>     ::= <Type>  ',' <Types>
         | <Type>

<Id List>   ::= Id ',' <Id List>
```

```
          | Id

<Func ID>   ::= <Type> ID
          | ID


!=============================================
Type Declaration
<Typedef Decl> ::= typedef <Type> ID ';'

<Struct Decl>  ::= struct Id '{' <Struct Def> '}'  ';'

<Union Decl>   ::= union Id '{' <Struct Def> '}'  ';'


<Struct Def>   ::= <Var Decl> <Struct Def>
            | <Var Decl>

!=============================================
Variable Declaration

<Var Decl>    ::= <Mod> <Type> <Var> <Var List>  ';'
            |     <Type> <Var> <Var List>  ';'
            | <Mod>      <Var> <Var List>  ';'

<Var>     ::= ID <Array>
          | ID <Array> '=' <Op If>

<Array>    ::= '[' <Expr> ']'
          | '[' ']'
          |

<Var List> ::=  ',' <Var Item> <Var List>
          |
```

```
<Var Item> ::= <Pointers> <Var>


<Mod>    ::= extern
         | static
         | register
         | auto
         | volatile
         | const

!================================================
Enumerations

<Enum Decl>   ::= enum Id '{' <Enum Def> '}' ';'

<Enum Def>    ::= <Enum Val> ',' <Enum Def>
           | <Enum Val>

<Enum Val>    ::= Id
           | Id '=' OctLiteral
           | Id '=' HexLiteral
           | Id '=' DecLiteral


!================================================
Types

<Type>    ::= <Base> <Pointers>

<Base>    ::= <Sign> <Scalar>
         | struct Id
         | struct '{' <Struct Def> '}'
         | union Id
         | union '{' <Struct Def> '}'
         | enum Id
```

```
<Sign>    ::= signed
        | unsigned
        |

<Scalar>  ::= char
        | int
        | short
        | long
        | short int
        | long int
        | float
        | double
        | void

<Pointers> ::= '*' <Pointers>
        |

!================================================
Statements

<Stm>      ::= <Var Decl>
         | Id ':'                    !Label
         | if '(' <Expr> ')' <Stm>
         | if '(' <Expr> ')' <Then Stm> else <Stm>
         | while '(' <Expr> ')' <Stm>
         | for '(' <Arg> ';' <Arg> ';' <Arg> ')' <Stm>
         | <Normal Stm>

<Then Stm>  ::= if '(' <Expr> ')' <Then Stm> else <Then Stm>
         | while '(' <Expr> ')' <Then Stm>
         | for '(' <Arg> ';' <Arg> ';' <Arg> ')' <Then Stm>
         | <Normal Stm>
```

```
<Normal Stm> ::= do <Stm> while '(' <Expr> ')'
          | switch '(' <Expr> ')' '{' <Case Stms> '}'
          | <Block>
          | <Expr> ';'
          | goto Id ';'
          | break ';'
          | continue ';'
          | return <Expr> ';'
          | ';'           !Null statement


<Arg>     ::= <Expr>
          |

<Case Stms> ::= case <Value> ':' <Stm List> <Case Stms>
          | default ':' <Stm List>
          |

<Block>    ::= '{' <Stm List> '}'

<Stm List>  ::=  <Stm> <Stm List>
          |


!
================================================
======
! Here begins the C's 15 levels of operator precedence.
!
================================================
======

<Expr>     ::= <Expr> ',' <Op Assign>
          | <Op Assign>
```

```
<Op Assign>  ::= <Op If> '='  <Op Assign>
          | <Op If> '+='  <Op Assign>
          | <Op If> '-='  <Op Assign>
          | <Op If> '*='  <Op Assign>
          | <Op If> '/='  <Op Assign>
          | <Op If> '^='  <Op Assign>
          | <Op If> '&='  <Op Assign>
          | <Op If> '|='  <Op Assign>
          | <Op If> '>>=' <Op Assign>
          | <Op If> '<<=' <Op Assign>
          | <Op If>

<Op If>     ::= <Op Or> '?' <Op If> ':' <Op If>
          | <Op Or>

<Op Or>     ::= <Op Or> '||' <Op And>
          | <Op And>

<Op And>    ::= <Op And> '&&' <Op BinOR>
          | <Op BinOR>

<Op BinOR>  ::= <Op BinOr> '|' <Op BinXOR>
          | <Op BinXOR>

<Op BinXOR>  ::= <Op BinXOR> '^' <Op BinAND>
          | <Op BinAND>

<Op BinAND>  ::= <Op BinAND> '&' <Op Equate>
          | <Op Equate>

<Op Equate>  ::= <Op Equate> '==' <Op Compare>
          | <Op Equate> '!=' <Op Compare>
          | <Op Compare>

<Op Compare> ::= <Op Compare> '<'  <Op Shift>
```

```
        | <Op Compare> '>'  <Op Shift>
        | <Op Compare> '<=' <Op Shift>
        | <Op Compare> '>=' <Op Shift>
        | <Op Shift>

<Op Shift>  ::= <Op Shift> '<<' <Op Add>
        | <Op Shift> '>>' <Op Add>
        | <Op Add>

<Op Add>    ::= <Op Add> '+' <Op Mult>
        | <Op Add> '-' <Op Mult>
        | <Op Mult>

<Op Mult>   ::= <Op Mult> '*' <Op Unary>
        | <Op Mult> '/' <Op Unary>
        | <Op Mult> '%' <Op Unary>
        | <Op Unary>

<Op Unary>  ::= '!'   <Op Unary>
        | '~'   <Op Unary>
        | '-'   <Op Unary>
        | '*'   <Op Unary>
        | '&'   <Op Unary>
        | '++'  <Op Unary>
        | '--'  <Op Unary>
        | <Op Pointer> '++'
        | <Op Pointer> '--'
        | '(' <Type> ')' <Op Unary>  !CAST
        | sizeof '(' <Type> ')'
        | sizeof '(' ID <Pointers> ')'
        | <Op Pointer>

<Op Pointer> ::= <Op Pointer> '.' <Value>
        | <Op Pointer> '->' <Value>
        | <Op Pointer> '[' <Expr> ']'
```

```
        | <Value>

<Value>    ::= OctLiteral
        | HexLiteral
        | DecLiteral
        | StringLiteral
        | CharLiteral
        | FloatLiteral
        | Id '(' <Expr> ')'
        | Id '(' ')'
        | Id
        | '(' <Expr> ')'
```