Obscurecedor de Código Java - Análise sintática

Lucas Neto Moreira - 09/0122755
Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

1 de novembro de 2013

Resumo

No processo de obscurecimento de código java, é importante que a análise sintática seja feita de forma precisa para que o compilador tenha total controle sobre as estruturas do código fonte.

1 Implementação

Foram utilizados dois geradores de parsers para a implementação do analisador léxico. Num primeiro momento a gramática foi implementada utilizando o programa Byacc/J [2] , o programa se integra ao analisador léxico Jflex[3] utilizando um método abstrato chamado NextToken , que deve ser implementado pelo usuário, o que retira do parser todo o trabalho de integração com o analisador léxico. Ao fim da implementação da gramática no Byacc/J os esforços foram voltados para a construção da arvore de parsing, mas a criação dos nodos da arvore foi impossibilitada por uma característica do projeto do parser. O programa disponibiliza uma classe chamada ParserVal , que encapsula os tipos possíveis para cada nodo na arvore, o que dificulta bastante a extensão do parser, e portanto foi necessário migrar a gramática para outro parser.

O parser Escolhido para a migração da gramática foi o Cup [1], o programa se integra sem dificuldade ao analisador léxico. A criação da arvore de parsing e feita utilizando nodos específicos para cada redução, os nodos foram gerados utilizando o programa Classgen[5], que recebe uma descrição muito similar a uma gramática livre de contexto e gera para cada redução uma classe com as informações necessárias e diferentes construtores para cada produção diferente em cada regra.

O desenvolvimento foi feito utilizando como base o projeto exemplo disponibilizado na pagina do Cup [1], que contem uma distribuição do Jflex e do Classgen, juntamente a um script Ant que guia a construção do parser.

2 gramática da linguagem

A implementação da gramática foi baseada na especificação oficial do java 5[4] ,foi escolhido um subconjunto bastante reduzido da gramática oficial para viabilizar a construção do parser a tempo. As principais regras da gramática são:

1. `Program ::= ClassDeclaration`

   A regra inicial descreve a maior unidade de compilação no escopo do parser , este token é importante pois a redução desta regra marca o fim da construção da arvore e permite que o parser execute métodos do usuário sobre a arvore resultante , esta característica facilita bastante a validação da arvore e o tratamento de erro no parser.

2. `ClassDeclaration ::= ACCESS_MODIFIER KEYWORD_CLASS IDENT {classStatements }`

   A declaracão de uma classe e a mais simples possível , não é permitida herança ou implementação de interfaces, todos os modificadores de acesso são permitidos.

3. `ClassStatements ::= ClassStatement ClassStatements | ClassStatement`

   este tipo de construção foi utilizado em diversos momentos na construção da gramática para implementar listas de tokens do mesmo tipo , a partir daqui esta construção sera omitida, e serão descritos apenas os tokens que contem não-terminais.

4. `ClassStatement ::= FieldVarDeclaration | MethodDeclaration`

   internamente a cada classe e possível declarar variáveis de instância e métodos.

5. `FieldVarDeclaration ::= ACCESS_MODIFIER TYPE IDENT;`

   Declaracões de variáveis de classe são feitas uma por statement para facilitar a posterior modificação do controle do programa na geração de código intermediário.

6. `MethodDeclaration ::= ACCESS_MODIFIER TYPE IDENT ( ListParam ) {Statements}`

Declaracões de métodos recebem um modificador de acesso , um tipo de retorno é uma lista de parâmetros, a produção do corpo do método fica por conta da mesma regra.

7. `parametro ::= TYPE:t IDENT:i`

   Parâmetros para métodos são construídos por um tipo e um identificador , a construção de listas e utilizada na redução das declaracões de métodos.

8. `statement ::= Expression | VarDefinition`

   Parâmetros para métodos são construídos por um tipo e um identificador , a construção de listas e utilizada na redução das declaracões de métodos.

9. `VarDefinition ::= TYPE identList`

   Declaracões de variareis locais podem ser feitas em bloco, para o propósito da geração de código intermediário e importante manter informações sobre o escopo de cada variável , o que pode ser feito utilizando apenas os nomes das variáveis.

10. `Expression ::= NumericExpression`

    Expressoes são statements que alteram o estado da classe.

11. ```
    NumericExpression ::= UN_OP NumericExpression
      | IDENT
      | NumericExpression ASS_OP NumericExpression
      | numericExpression EXP_OP numericExpression
    ```

    O token de Expressões numéricas cobre todas as expressões sobre constantes , literais e variáveis que podem ser feitas utilizando operadores de atribuição e operações aritméticas.

3 Criação da Arvore de Parsing

Cada redução feita pelo parser gera um nodo na arvore de parsing , os nodos armazenam outros nodos que correspondem as outras produções a ultima redução marca a criação da raiz da arvore. Desta forma a geração da arvore é análoga à de uma gramática S-atribuída.
    ```

4 Metodologia de Teste

Foram feitos testes para cada regra , substituindo a produção da regra inicial pela regra em questão de forma a construir a sub-arvore equivalente à regra, o arquivo final de teste contem tokens que forçam a redução de todas as regras da gramática.

5 Principais Arquivos

Os arquivos que geram o parser são:

- cl.ast.cl

  Arquivo de definições para o Classgen.

- cup.Parser.cup

  Arquivo de definições para o Cup. Gera o parser.

- flex.Scanner.jflex

  Arquivo de definições para Jflex , gera o analisador léxico.

- input.test

  Arquivo de teste , contem o código java utilizado na geração da arvore.

- build.xml

  script ant que gera o parser. O parser pode ser criado e executado pelo comando "ant run", é preciso ter o ant instalado (sudo apt-get install ant).

6 Dificuldades

A principal dificuldade encontrada na implementação do parser foi a criação da arvore, foi necessário migrar a implementação da gramática para um outro parser, o que causou um overhead de tempo significativo, incluindo a busca e escolha por um novo parser e a adaptação do analisador léxico para a integração com o novo parser.

Referências

[1] Scott Hudson. Cup, lalr parser generator in java. Technical report, Georgia Tech, 2012.

[2] Tomas Hurka. Byacc/j. Technical report, unaffiliated, 2008.

[3] Gerwin Klein. Jflex , the fast lexical analyser generator. Technical report, unaffiliated, 2009.

[4] Oracle. Java manual,chapter 18. syntax. Technical report, Oracle, 2011.

[5] Sebastian Winter. Classgen. Technical report, Technical University of Munich, 2002.