Universidade de Coimbra

Departamento de Engenharia Informática

Compiladores

Compilador para linguagem iJava

Autor:
David Cardoso
Número: 2011164039

Autor:
Bruno Caceiro
Número: 2008107991

27 de Maio de 2014

Índice

1 Introdução

Este projecto consiste no desenvolvimento de um compilador para a linguagem *iJava* (imperative Java), que consiste num pequeno subconjunto da linguagem Java (versão 5.0). Os programas da linguagem *iJava* são constituídos por uma única classe (a principal), contendo necessariamente um método *main*, e podendo conter outros métodos e atributos, todos eles estáticos e (possivelmente) públicos.

O projecto foi estruturado em 3 fases, primeiramente foi feita a Análise Lexical, implementada na linguagem *C* e utilizando a ferramenta *lex*. A segunda fase consistiu na análise sintática, com a construção da árvore de sintaxe abstrata e análise semântica (tabelas de símbolos, deteção de erros semânticos). No final foi feita a geração de código.

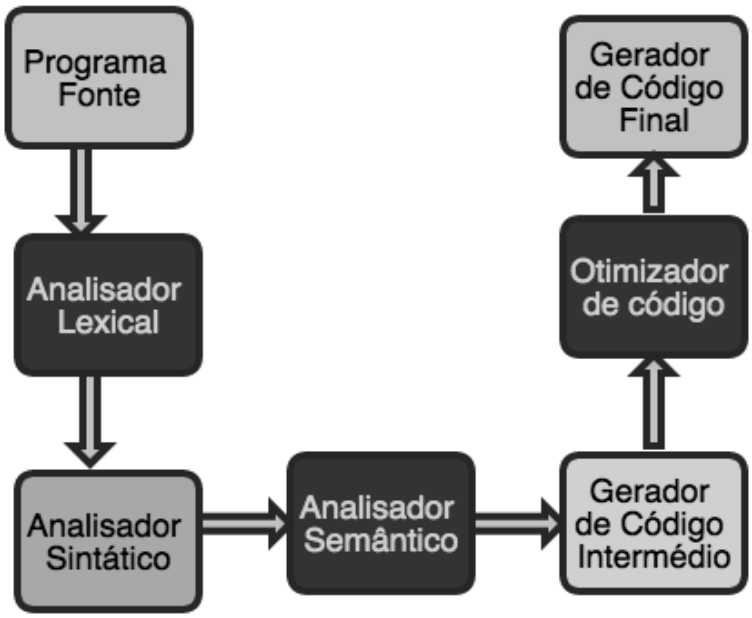

Figura 1: Fases de Compilação

2 Análise Lexical

A Análise Lexical consiste em analisar a entrada de linhas de caracteres e produzir uma sequência de símbolos (*tokens*) que podem ser manipulados mais facilmente por um *parser*. É uma forma de verificar um determinado alfabeto, neste caso o alfabeto da linguagem *iJava*. Esta análise pode ser dividida em três fases:

- Extração e classificação de *tokens*;
- Eliminação de delimitadores e comentários;
- Tratamento de erros;

2.1 Tokens

- **ID:** Sequências alfanuméricas começadas por uma letra, onde os símbolos "_"e "$"contam como letras. Maiúsculas e minúsculas são consideradas letras diferentes
- **INTLIT:** Sequências de dígitos decimais e sequências de dígitos hexadecimais (incluindo a-f e A-F) precedidas de "0x"
- **BOOLLIT:** "true"| "false"
- **INT:** "int"
- **BOOL:** "boolean"
- **NEW:** "new"
- **IF:** "if"
- **ELSE:** "else"
- **WHILE:** "while"
- **PRINT:** "System.out.println"
- **PARSEINT:** "Integer.parseInt"
- **CLASS:** "class"
- **PUBLIC:** "public"
- **STATIC:** "static"
- **VOID:** "void"
- **STRING:** "String"
- **DOTLENGTH:** ".length"
- **RETURN:** "return"
- **OCURV:** "("
- **CCURV:** ")"
- **OBRACE** "{"
- **CBRACE:** "}"
- **OSQUARE:** "["
- **CSQUARE:** "]"

Foi necessário separar alguns *tokens* devido às diferentes prioridades que cada operador tem.

- **OP1:** "&&"
- **OP1OR:** "| |"
- **OP2:** "<"| ">"| "<="| ">="
- **OP2EQS:** "=="| "!="
- **OP3:** "+"| -"
- **OP4:** "*"| "/"| "%"
- **NOT:** "!"
- **ASSIGN:** "="
- **SEMIC:** ";"
- **COMMA:** ","
- **RESERVED:** O *iJava* é um subconjunto da linguagem *Java*, como tal, existe um conjunto de funcionalidades que embora não sejam suportadas, têm de ser consideradas. Assim, foi necessário tratar todo um conjunto de palavras reservadas de forma a permitir que sejam lexicalmente válidas mas não sintaticamente.
  - abstract | assert | break | byte | case | catch | char | const | continue | default | do | double | enum | extends | final | finally | float | for | goto | implements | import | instanceof | interface | long | native | package | private | protected | short | strictfp | super | switch | synchronized | this | throw | throws | transient | try | volatile | null | ++ | –

2.1.1 Tratamento de Erros

Se forem detectados erros lexicais no ficheiro de entrada então é impressa uma mensagem de erro no *stdout*:

- "Line<num linha>,col<num coluna>:illegal character('<c >'\n)"
- "Line<num linha>,col<num coluna>:unterminated comment\n"

3 Análise Sintática e Semântica

3.1 Gramática

A gramática é a maneira formal de especificar a sintaxe de uma linguagem. Desenvolver uma gramática não ambígua é um dos passos mais importantes para o sucesso do compilador. Para a gramática da linguagem *iJava* usámos a notação **BNF** *(Backus Naur Form)*. A gramática que nos foi dada era ambígua e por isso tivémos de efectuar diversas alterações para permiter a análise sintática ascendente com o *yacc*.
Algumas das alterações que efectuámos foram:

- Usar recursividade à direita, nas regras onde era possível existirem uma ou mais repetições
- Criação de estados adicionais para as regras que continham *tokens* opcionais

3.1.1 Estado Final da Gramática

A gramática final, utilizada no *yacc* é a seguinte apresentada.

START $\longrightarrow$ $CLASS$ ID $OBRACE$ field_or_method_declaration $CBRACE$

field_or_method_declaration $\longrightarrow$ $FieldDecl$ field_or_method_declaration
| $MethodDecl$ field_or_method_declaration

$FieldDecl$ $\longrightarrow$ $STATIC$ $VarDecl$ VarDecl_REPETITION

$MethodDecl$ $\longrightarrow$ $PUBLIC$ $STATIC$ method_type_declaration ID $OCURV$ FormalParams $CCURV$ $OBRACE$ VarDecl_REPETITION statement_declaration_REPETITION $CBRACE$

method_type_declaration $\longrightarrow$ Type
| $VOID$

FormalParams $\longrightarrow$ Type ID several_FormalParams
| $STRING$ $OSQUARE$ $CSQUARE$ ID

several_FormalParams $\longrightarrow$ $COMMA$ Type ID several_FormalParams

VarDecl_REPETITION $\longrightarrow$ VarDecl VarDecl_REPETITION

VarDecl $\longrightarrow$ Type ID several_var_decl_in_same_instructionOPTIONAL $SEMIC$

several_var_decl_in_same_instructionOPTIONAL $\longrightarrow$ $COMMA$ ID several_var_decl_in_same_instruct

Type $\longrightarrow$ INT $OSQUARE$ $CSQUARE$

| $BOOL$ $OSQUARE$ $CSQUARE$
| INT
| $BOOL$

statement_declaration_REPETITION $\longrightarrow$ Statement statement_declaration_REPETITION

Statement $\longrightarrow$ $OBRACE$ several_statement $CBRACE$

| IF $OCURV$ Expr $CCURV$ Statement %$prec$ IFX
| IF $OCURV$ Expr $CCURV$ Statement $ELSE$ Statement
| $WHILE$ $OCURV$ Expr $CCURV$ Statement
| $PRINT$ $OCURV$ Expr $CCURV$ $SEMIC$
| ID array_indexOPTIONAL $ASSIGN$ Expr $SEMIC$
| $RETURN$ return_expression $SEMIC$

several_statement $\longrightarrow$ Statement several_statement

array_indexOPTIONAL $\longrightarrow$ $OSQUARE$ Expr $CSQUARE$

return_expression $\longrightarrow$ Expr

IndexableExpr $\longrightarrow$ ID

| $INTLIT$ $BOOLLIT$ ID $OCURV$ Args_OPTIONAL $CCURV$
| $OCURV$ Expr $CCURV$
| Expr $DOTLENGTH$
| IndexableExpr $OSQUARE$ Expr $CSQUARE$
| $PARSEINT$ $OCURV$ ID $OSQUARE$ Expr $CSQUARE$ $CCURV$

Expr $\longrightarrow$ Expr $OP1$ Expr %$prec$ $OP1$

| Expr $OP1OR$ Expr %$prec$ $OP1OR$
| Expr $OP4$ Expr %$prec$ $OP4$
| Expr $OP3$ Expr %$prec$ $OP3$
| Expr $OP2$ Expr %$prec$ $OP2$
| Expr $OP2EQS$ Expr %$prec$ $OP2EQS$
| $OP3$ Expr %$prec$ NOT

| *NOT* Expr %*prec NOT*
| *NEW INT OSQUARE* Expr *CSQUARE*
| *NEW BOOL OSQUARE* Expr *CSQUARE*
| IndexableExpr

Args_OPTIONAL ⟶ Args

Args ⟶ Expr comma_expr

comma_expr ⟶ *COMMA* Expr comma_expr

3.2 Árvore de Sintaxe Abstrata

Para a árvore de sintaxe abstrata optámos por utilizar um nó genérico (*Node*) com a seguinte estrutura:

```
/* General Node */
typedef struct _Node
{
    //Type of the Node (to identify the type of the node)
  NodeType n_type;

    //Type of the Struct (Int, Void, String,...)
  Type type;

    //Id or list of id's
    listID* id;

    //The tree next nodes (the case of if)
    struct _Node* n1;
    struct _Node* n2;
    struct _Node* n3;

    //Next node
    struct _Node* next;

    //Literals (to store the values)
    char* value;

    char isStatic;
}Node;


/* Linked list of ID's (for multiple declaration of variables) */
typedef struct _idList
{
  char* id;
  struct _idList* next;
}listID;
```

Listing 1: Estruturas para representação de Nós da Árvore

3.2.1 Exemplo

Considerando o seguinte programa:

```
1 class gcd {
2  public static void main(String[] args) {
3      int x;
4  }
5 }
```

Listing 2: Programa Exemplo

A árvore gerada é a seguinte:

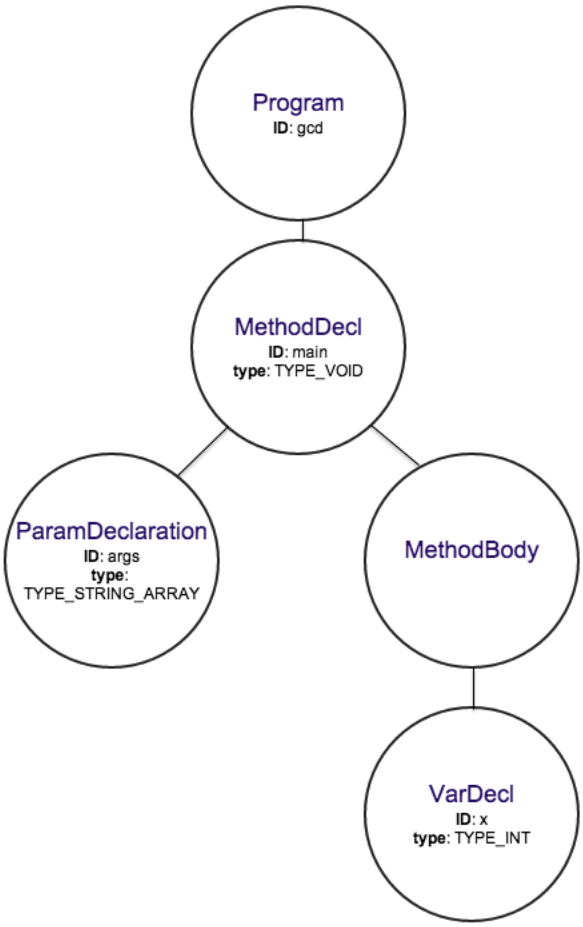

Figura 2: Árvore de Sintaxe Abstrata para o Programa Exemplo

3.3 Análise Semântica

A Análise Semântica tem como principais objectivos a ligação das definições de variáveis com a sua utilização, a verificação da correcção de tipos, declarações e chamadas de funções.

3.4 Tabela de Símbolos

Para representação da tabela de símbolos, foram utilizadas as seguintes estruturas:

```
/* Table Node */
typedef struct _TableNode
{
    //Type of the Node (to identify the type of the node)
    TableType n_type;

    //Type of the Struct (Int, Void, String,...)
    Type type;

    //ID or list of id's
    listID* id;

    //Next node
    struct _TableNode* next;

    //If is a param
    char isParam;
}TableNode;

/* Table */
typedef struct _Table{
    TableNode* table;
    struct _Table* next;
}Table;
```

Listing 3: Estruturas para representação da Tabela de Símbolos

3.5 Tratamento de Erros Semânticos

1º Passagem: Verificar se as variáveis existem (incluindo nomes de funções). Percorrer 2 scopes, o scope da função que chamou essa variável, caso não seja encontrado aí vai se procurar no scope global, caso não exista aí, então é porque estamos a chamar uma merda que não existe e então dá erro.

2ª Passagem (tudo em simultâneio) Percorrer a árvore e analisar os statements (esquerdo/direito), depois ramifica-se para analisar os filhos