Universidade Federal de Santa Catarina
Ciências da Computação
INE5426 - Construção de Compiladores

Karla Aparecida Justen

Construção de Compiladores
Etapa 4 - Análise Semântica + Gerador de código

Pasquinho

Introdução

Como objetivo inicial da disciplina de Construção de Compiladores os alunos deveriam escolher uma linguagem de programação e construir passo a passo seu compilador. A linguagem a ser explorada foi o Pascal. A decisão de escolha veio do fato do aluno Bruno ter tido seu primeiro contato com esta linguagem de programação na disciplina de Introdução à Computação ainda quando cursava Engenharia Química. Outros fatores decisivos para a escolha da mesma são sua gramática simplificada e ao mesmo tempo com algumas construções únicas, como o **goto**, algo não existente nas linguagens de programação atuais.

Fatos interessantes sobre o Pascal:

- Criada pelo suíço Niklaus Wirth
- Derivada do Algol-60
- Programação Modular e Estruturada
- Facilita criação de procedimentos com baixo acoplamento e alta coesão.
- Criada especialmente para ensinar programação estruturada e nas fábricas de software de Niklaus.

Estrutura Básica do Pascal

Um programa em Pascal é composto de constantes e variáveis globais, procedimentos e funções re-entrantes e um programa principal. Procedimentos não retornam valores, funções sim. Tanto em procedimentos quanto em funções os parâmetros podem ser passados por referência ou por valor. É possível passar vetores e matrizes com o tamanho, mas não a quantidade de dimensões, especificado no tempo de execução.

Procedimentos e funções podem conter, dentro de seu escopo, novos procedimentos e funções. Dentro de qualquer parte do programa também podem ser criados blocos com os comandos BEGIN e END, que também possuem seu próprio escopo. (retirado da Wikipédia [2]).

Números pares entre dois valores inteiros [editar | editar código-fonte]

```
program pares;

var
 x, y: integer;

begin
 writeln('Digite os dois valores');
 readln(x, y);
 if (x mod 2)<> 0 then
  x := x + 1;
 while x<= y do
 begin
  writeln(x, ' - ');
  x := x + 2;
 end;
 writeln('Fim da Lista');

end.
```

Figura 1 - Exemplo de Programa em Pascal [2]

Gramática da Linguagem

Inicialmente foi tomada como base para a definição léxica da linguagem a definição EBNF do Pascal [1] disponível online.

Programs and Blocks

program
    program-heading block "."
program-heading
    program identifier "(" identifier-list ")" ";"
block
    declaration-part statement-part
declaration-part
    [label-declaration-part]
    [constant-definition-part]
    [type-definition-part]
    [variable-declaration-part]
    procedure-and-function-declaration-part
label-declaration-part
    label label { "," label } ";"
constant-definition-part
    const constant-definition ";" { constant-definition ";" }

Figura 2 - Definição EBNF [1]

O passo seguido foi transformar esta definição para a definição de EBNF aceita pelo Railroad [3]. O Railroad é uma ferramenta que cria diagramas de sintaxe para gramáticas livre de contexto definidas em EBNF.

Porém ao colocar essa gramática no contexto de aplicação ao ANTLR4, não foi possível compilá-la, devido a um emaranhado de recursões à esquerda, após ajustes feitos a gramática ficou complicada de entender. O que levou o grupo a recorrer aos recursos apresentados pela gramática apresentada no Pascal para ANTLR [4]. Com base nessa referência, foi refeita a análise para converter essa gramática nos conformes da EBNF, apresentado no Anexo 1, sendo possível gerar os diagramas, como alguns apresentados na FIGURA 3. O conjunto completo está num compacto entregue junto com este relatório. Por fim, foi possível realizar testes, apresentado a seguir.

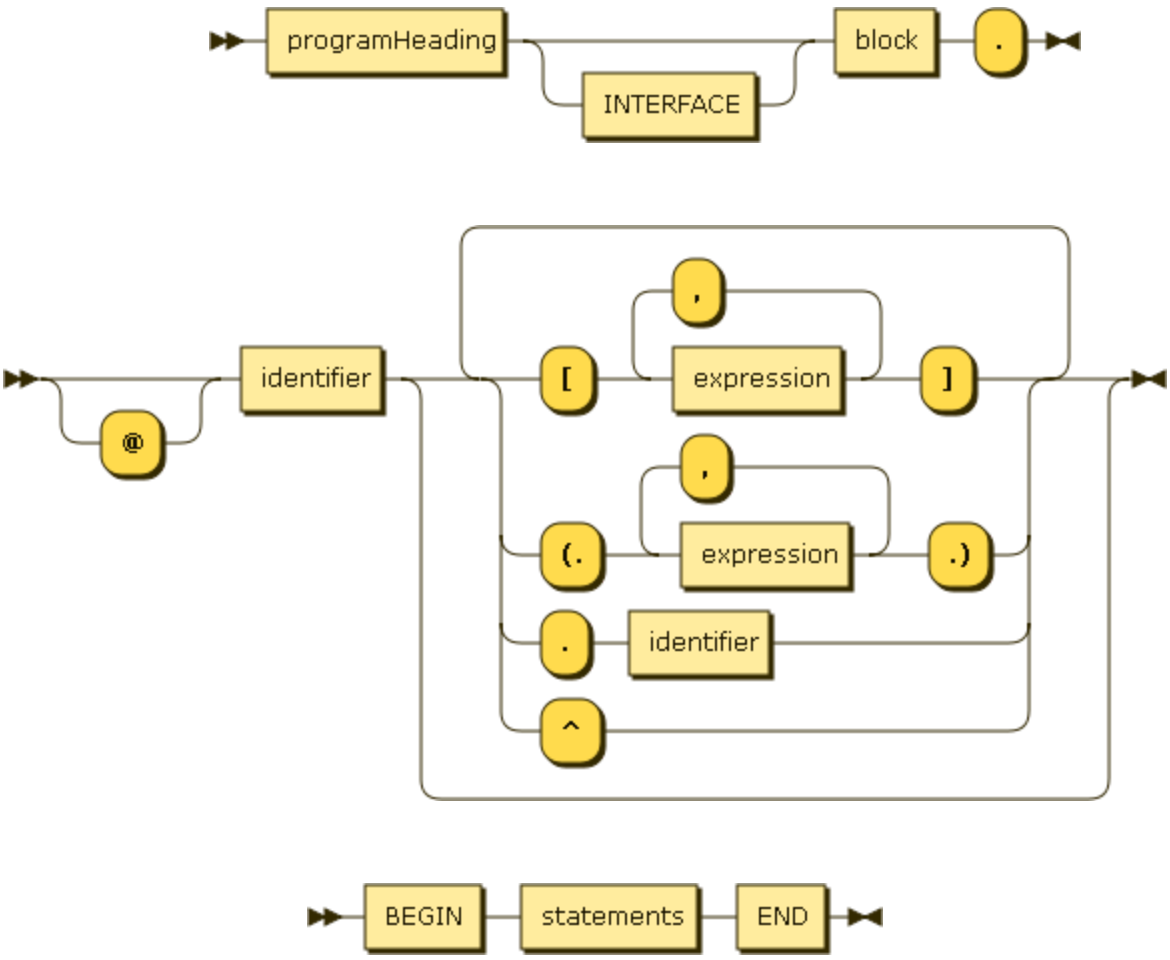

Figura 3 - Exemplos de Diagramas de Sintaxe gerados pelo Railroad

Desenvolvimento

Para desenvolver a parte semântica foi estudado e replicado o trabalho da sequência de vídeos “Let's build a compiler - Compilerbau ANTLR Tutorial”, apresentado por Yankee [8]. Assim iremos desenvolver a gramática replicada de [8] para ficar semelhante com a gramática que propomos até agora.

Mas como as abordagens utilizadas são bastantes diferentes de como foi feito o trabalho até agora, o material apresentado nesta etapa será um pouco diferente do proposto pelo professor. Onde ao mesmo tempo que é feita a análise semântica também é desenvolvida a geração de código para execução.

A evolução do compilador replicado de [8] está disponivel em [10].

Enquanto em GitHub-Let-s-build-a-compiler---Pascal [9] está o projeto que aprimoramos a partir de [8].

Para gerar códigos executáveis, é tido como auxilio o Jasmim. Para isso é gerado um arquivo formado por instruções Java bytecode [11].

Para aprimorar a gramática são seguidos os seguintes passos:
1º Adicionar no arquivo “Compiler/test/de/letsbulidacompiler/compiler/CompilerTest.java” um exemplo do que se pretende acrescentar na gramática;

2º Compilar o Projeto Compiler: Run AS → TestNG Test. Assim, os novos testes acrescentados provavelmente acusarão erro. É isso o esperado.

3º Alterar a gramática salva como “Parser/grammar/Demo.g4”. Salvar as alterações.

4º Com o terminal em “Parser/grammar” executar o seguinte comando:

```
$ java -jar ../lib/antlr-4.7-complete.jar -package de.letsbuildacompiler.parser -o
../src/de/letsbuildacompiler/parser -no-listener -visitor Demo.g4
```

Isso irá atualizar a análise léxica e sintática gerada automaticamente pelo ANTLR4. Por isso é preciso, em seguida, sobre o projeto Parser: dar Refresh.

5º Agora é possível editar o “Compiler/src/de/letsbuildacompiler/compiler/MyVisitor.java”, fazendo @Override sobre os apelidos dados aos tokens, tratando as novas regras semânticas.

6º Novamente compilar o Projeto Compiler: Run AS → TestNG Test. Caso tudo tenha sido implementado corretamente, os novos testes acrescentados passarão no teste.

Gramática final apresentada em [8] e no .zip 22 de [10] :

```
grammar Demo;

program: programPart+;

programPart: statement           #MainStatement
               | functionDefinition #ProgPartFunctionDefinition
               ;

statement: println ';'
         | varDeclaration ';'
         | assignment ';'
         | branch
         | print ';'
         ;

branch: 'if' '(' condition=expression ')' onTrue=block 'else' onFalse=block
      ;

block: '{' statement* '}' ;

expression: left=expression '/' right=expression #Div
              | left=expression '*' right=expression #Mult
          | left=expression '-' right=expression #Minus
          | left=expression '+' right=expression #Plus
          | left=expression operator=('<' | '<=' | '>' | '>=') right=expression
#Relational
              | left=expression '&&' right=expression #And
              | left=expression '||' right=expression #Or
              | number=NUMBER  #Number
              | txt=STRING #String
              | varName=IDENTIFIER #Variable
              | functionCall #funcCallExpression
              ;

varDeclaration: 'int' varName=IDENTIFIER ;

assignment: varName=IDENTIFIER '=' expr=expression;

println: 'println(' argument=expression ')';

print: 'print(' argument=expression ')';


functionDefinition: 'int' funcName=IDENTIFIER '(' params=parameterDeclaration ')' '{'
statements=statementList 'return' returnValue=expression ';' '}' ;

parameterDeclaration: declarations+=varDeclaration ( ',' declarations+=varDeclaration)*
                                |
                                ;
```

```
statementList: statement* ;

functionCall: funcName=IDENTIFIER '(' arguments=expressionList ')' ;

expressionList: expressions+=expression ( ',' expressions+=expression)*
                      |
                      ;

IDENTIFIER: [a-zA-Z] [a-zA-Z0-9]* ;
NUMBER: [0-9]+;
WHITESPACE: [ \t\n\r]+ -> skip;
STRING: '"' .*? '"' ;
```

Regras Semânticas:

- ordem de execução da expressão numérica; resolvido de forma sintática;
- declaração de variáveis numéricas;
- exceções implementadas:
    - não permitir declarar variável sem tipo;
    - não deixar imprimir variável sem estar declarada;
    - não permitir declarar a mesma variável (com mesmo nome) mais de uma vez;
    - permitir ter variáveis dentro de um método, com o mesmo nome de uma variável fora do método;
    - reconhecer como declaração dos parâmetros de métodos como variáveis, acrescentando na tabela de símbolos;
    - garantir que métodos chamados já foram declarados e implementados;
    - ter duas funções com o mesmo nome, apenas se diferenciando pelos parâmetros;
    - não permite que a mesma função seja definida mais de uma vez (nome e parâmetros iguais);
    - fez if funcionar;
    - operations <, <=, >, >=

Alterações feitas:

Estrutura Básica do Pascal

1. BEGIN e END nas condicionais, sendo mais restrito que o Pascal. Pois precisa sempre ter BEGIN e END, independente da quantidade de *statements* tiverem;

2. Declaração de *functions* mais parecidas que pascal; ainda não há separação da parte de declaração de variáveis, por exemplo.

```
function x(int i): int;
BEGIN
      writeln(i);
      return i;
END;
```

3. Renomear print para write;

4. Renomear println para writeln;

5. Iniciei a estruturação do cabeçalho:

```
program compiler;

function randomNumber(): int;
BEGIN
  int i;
  i = 4;
  return i;
END;

writeln(randomNumber());
```

6. **While**

Sintaxe

```
while <condição> do
 begin
   <comandos>;
   <comandos>;
 end;
```

7. **Forma de declaração de variáveis**

```
NOME: int;
```

```
function
NomeDaFunção(Parâmetro1: Tipo; ParâmetroN: Tipo): Tipo de retorno;
var
  ...
begin
  ...
End;
```

8. **Ordenação do bloco para declaração de Variáveis**

   Criei o token “blockVarDeclaration” acrescentei na construção do toke principal, “program” e adaptei o método “visitProgram” no MyVisitor.

9. **Ordenar Funções**

   (fixar após declaração de variáveis)

10. **Declaração de Constantes**

    Diferenciei a declaração de ‘=’para ‘:=’ pois não consegui encontrar uma maneira de não haver confunsão com ‘assigment’.

```
program  media_notas;
CONST
 PI := 3.1415926;
VAR
 NOME: string;
BEGIN ...
END.
```

Importante! Para funcionar adequadamente na classe DemoParser.java, localizada no projeto Parser/src/de.letsbuildacompiler.parser, é preciso fazer as seguintes alterações:

- Na linha 1160: onde está "`public Token constValue;`"
  trocar por: "`public ExpressionContext constValue;`"
- Na linha 1185: onde está
  "`((ConstDeclarationContext)_localctx).constValue = match(NUMBER);`"
  trocar por: "`((ConstDeclarationContext)_localctx).constValue = expression(0);`"

11. **Permitir Comentários**
    Comentários são feitos entre chaves, { … }.

12. **Salvar console em arquivos**
    Criei "createFile(String result)" na classe MyVisitor.java e chamo ela no final do método "visitProgram()".

Para executar os arquivos gerados, é preciso primeiro usar o Jasmim para para converter o arquivo .j para .class, para isso é preciso usar o seguinte comando:

```
java -jar ../Compiler/lib/jasmin.jar <exemplo>.j
```

Com o .class criado, para executar:

```
java <exemplo>
```

13. **Case** (incompleto)

```
CASE seletor OF
     alvo1 : BEGIN
               ... instruções ...
             END;

     alvo2 : comando2;

     alvo3 : BEGIN
               ... instruções ...
             END;
     ELSE comando4;
END;
```

14.

Referências:

[1] Definição EBNF: http://www.fit.vutbr.cz/study/courses/APR/public/ebnf.html

[2] Wikipédia Pascal: https://pt.wikipedia.org/wiki/Pascal_(linguagem_de_programa%C3%A7%C3%A3o)

[3] Railroad: http://bottlecaps.de/rr/ui

[4] Pascal para ANTLR: https://github.com/antlr/grammars-v4/tree/master/pascal

[5] http://eli-project.sourceforge.net/pascal_html/pascal-.html

[6] https://github.com/antlr/antlr4/blob/master/doc/getting-started.m

[7] Livro ANTLR: https://moodle.ufsc.br/pluginfile.php/2197138/mod_resource/content/1/The%20Definitive%20ANTLR%204.pdf

[8] "Let's build a compiler - Compilerbau ANTLR Tutorial", apresentado por Yankee: https://www.youtube.com/watch?v=2uvKTmfPNzE&index=1&list=PLOfFbVTfT2vbJ9qiw_6fWwBAmJAYV4iUm

[9] https://github.com/karlajusten/-Let-s-build-a-compiler---Pascal

[10] https://drive.google.com/drive/folders/0B1nNj03qFNnYWG01cU5sWW5HTEk?usp=sharing

[11] https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings