Universidade Estadual de Maringá
Ciência da computação

RELATÓRIO - COMPILADOR RASCAL

Fernando Silva Silvério RA: 98936
Héctor Dorrighello Giacon RA: 99450

Maringá
2021

1 - Introdução

Segue presente neste relatório o detalhamento de todas as atividades realizadas para o desenvolvimento do compilador para a linguagem *Rascal*. O relatório será dividido entre as seguintes seções: Decisões de projeto; Cumprimento das Etapas; Descrição dos Módulos e Compilação e execução.

2 - Decisões de Projeto

Para a implementação do código fonte do compilador a princípio a escolha se voltou para a linguagem C++ pois acreditava-se que devido ao fato da mesma contar com estruturas de dados por padrão haveria uma certa facilidade para a construção do código. Entretanto a integração do C++ com o *Bison* (ferramenta utilizada para realizar a análise sintática) trouxe bastante problemas. Devido ao contratempo mencionado foi decidido alterar a linguagem para C.

Com essa decisão o trabalho em sua totalidade foi implementado utilizando a linguagem C, o *Bison* e o *Flex* para a análise léxica.

3 - Cumprimento das Etapas

Considerando a primeira etapa da compilação, a fase léxica, não houve qualquer problema no reconhecimento dos tokens definidos pela linguagem estabelecida.

Na etapa da análise sintática a gramatica desenvolvida não apresentou qualquer erro de *"shift/reduce"* o que possibilitou chegar na terceira etapa com a estrutura consolidada.

A geração da árvore sintática foi realizada com sucesso, mesmo mostrando ser a parte mais complexa até então, considerando a dificuldade para lidar com a estrutura de nós necessários para a formação da mesma.

No tocante à análise semântica, apesar de mostrar as mensagens esperadas em alguns dos casos de teste o compilador apresenta problemas com a análise de expressões considerando o lado direito de uma expressão binária, a dupla de desenvolvimento do compilador encontrou o local do erro mas não soube encontrar a razão que impede o funcionamento esperado da mesma, acredita-se que a lógica do código implementado esteja correta porém não se sabe a razão do mal funcionamento.

Mas considerando os locais onde não dependem de analise de expressão, funcionaram corretamente, como é o caso de verificar se uma variável que esta sendo utilizada está declarada. O exemplo mostrado na figura 1, a variável *h* não está declarada, mostrado corretamente pelo erro na figura 2. A falha de segmentação subsequente se deve à analise de expressão necessária no restante do código.

```
1   program cmdIf;
2
3     var x, y: integer;
4
5   begin
6     h := 10;
7     read(x, y);
8     if x <= y then
9       write(x + y)
10    else
11      write(x * y);
12  end.
```

Figura 1 – Código com variável não declarada.

```
Sucesso sintatico!

Variável h não declarada

./teste.sh, linha 7: 47124 Falha de segmentação  ./rascal tests/cmdIf.ras
```

Figura 2 – Erro semântico mostrado pelo compilador *Rascal*.

Um outro erro mostrado corretamente é para procedimentos. Seguindo o mesmo principio do exemplo 1, n exemplo 3 o procedimento *proc1* não está declarado, e portanto, deve ser retornado um erro informando, como mostra a figura 4. Lembrando que este erro segue o mesmo padrão para funções, mas como uma chamada de função depende da analise de expressão, então ela não funciona corretamente. A falha de segmentação subsequente se deve à analise de expressão necessária no restante do código.

```
1   program procSimples;
2
3     var x, y: integer;
4
5     procedure p();
6       var a, b: integer;
7     begin
8       a := 2 * x;
9       b := a + 1;
10      y := b + y;
11    end;
12
13  begin
14    proc1();
15    x := 5;
16    y := 10;
17    p();
18    write(x, y);
19  end.
```

Figura 3 – Código com procedimento não declarado.

```
Sucesso sintatico!

Procedimento proc1 não está declarado

./teste.sh, linha 7: 47891 Falha de segmentação  ./rascal tests/procSimples.pas
```

Figura 4 – Erro semântico mostrado pelo compilador *Rascal*.

E por fim, outro erro mostrado corretamente pelo nosso compilador *Rascal* é o fato de ter o número incorreto de parâmetros em um procedimento. Neste caso, o procedimento está declarado mas como mostrado na figura 5, o procedimento p não possui parâmetros mas na sua chamada, na linha 14 do código, ele possui dois parâmetros, o que está incorreto e devidamente mostrado pelo compilador na figura 6. E seguindo o exemplo anterior, este caso vale o mesmo para funções mas como dependem de analise de expressões, esta parte fica comprometida. A falha de segmentação subsequente se deve à analise de expressão necessária no restante do código.

```
1   program procSimples;
2
3     var x, y: integer;
4
5     procedure p();
6       var a, b: integer;
7     begin
8       a := 2 * x;
9       b := a + 1;
10      y := b + y;
11    end;
12
13  begin
14    p(x,y);
15    x := 5;
16    y := 10;
17    write(x, y);
18  end.
```

Figura 5 – Código com número incorreto de parâmetros na chamada do procedimento.

```
Sucesso sintatico!

Número incorreto de parâmetros no procedimento p

./teste.sh, linha 7: 48603 Falha de segmentação  ./rascal tests/procSimples.pas
```

Figura 6 – Erro semântico mostrado pelo compilador *Rascal*.

Infelizmente a geração do código para a máquina *MEPA* ficou comprometida devido ao problema apresentado na etapa anterior, fazendo com que essa funcionalidade não esteja presente no compilador desenvolvido. Todavia os locais no código semântico onde a tradução seria feita estão comentados para caso semanticamente funcionasse, ali os códigos da *MEPA* seriam inseridos em uma matriz de *String's*, no qual cada linha corresponde a uma linha no arquivo de assembly. Ao final da execução de *analisaBloco*, dentro de *analisaPrograma*, a função *gerarArquivoMepa* é chamada, se não houver erros, o conteúdo da matriz de *String's* é passado para um arquivo e assim a *MEPA* seria chamada.

4 - Descrição dos Módulos

4.1 - Organização do Analisador Léxico

Para implementar o analisador léxico, foi utilizada a ferramenta *Flex* que possui a função de gerar o autômato que recebe os tokens da linguagem, ou seja o próprio analisador léxico. Essa etapa é encontrada no arquivo *flex.lex*, onde além de haver a especificação dos tokens e seus significados na linguagem Rascal vale mencionar também a presença de *regex* (expressões regulares) para especificar os identificadores, números inteiros, comentários, quebra de linha e espaço na linguagem.

4.2 - Organização do Analisador Sintático

Em relação à organização do analisador sintático, foi utilizado a ferramenta *Bison*, no qual as regras implementadas no arquivo *bison.y* refletem à sintaxe da linguagem Rascal. Em regras comuns como 'programa' a ação necessária foi chamar a função correspondente para se gerar um nó da arvore sintática abstrata. Já em casos onde houve recursão, como em 'lista_identificadores', foi necessário utilizar o conceito de lista encadeada juntamente à recursão da regra.

4.3 - Geração da AST

A geração da árvore sintática abstrata se dá juntamente com a ferramenta *Bison* no qual em cada regra há a sua ação correspondente, que é a chamada de uma função que irá construir o nó da arvore naquele momento da execução. A árvore em si se da pelo fato de que cada nó possui nós internos até que ao fim existam somente *tokens*. Além disso há estruturas que possuem característica de herança na gramática e para esses casos foi utilizado *enum* juntamente com *union*. Todos essa estrutura está nos arquivos *ast.c* e *ast.h*.

4.4 - Modulo Semântico

Para a etapa semântica é necessário percorrer a arvore sintática abstrata gerada na etapa anterior. Para isso foram criadas funções que percorrem as *struct's* da arvore seguindo a sua implementação, analisando cada tipo de nó e verificando se as variáveis, funções e procedimentos estão declarados, visíveis no escopo, se possuem os parâmetros corretos bem como a inserção de tais estruturas na tabela de símbolos. A tabela em si foi implementada como uma pilha e o seu conteúdo contempla todos os parâmetros possíveis

necessários para variáveis, funções, parâmetros formais e procedimentos, podendo ser percorrida como uma lista para procurar um elemento ou um atributo. A implementação deste módulo está nos arquivos *semantico.c* e *semantico.h* e da tabela de simbolos em *tabelaSimbolos.c* e *tabelaSimbolos.h*.

4.5 - Geração de Código *MEPA*

Como houve problema na execução do código semântico, mencionado anteriormente, a geração de código para a máquina *MEPA* foi abordada apenas mostrando no código onde cada parte seria escrita, essa evidenciação foi feita por meio de comentários nos locais onde a analise semantica foi validada e dita correta. Mas estruturalmente no código possui uma flag para caso haja erros, e caso não haja, um arquivo é aberto com o nome do *<programa>.ras* através da função 'gerarArquivoMepa'. Mediante a isso, o que foi feito deste módulo está no arquivo *semantico.c* e *semantico.h*.

5 - Compilação e Execução

Para compilar o projeto e gerar o programa compilador *Rascal*, é necessário utilizar o programa *Make* pelo terminal. Após utilizar esse comando, o compilador será gerado e poderá ser executado. Caso a compilação já tenha sido feita e deseja-se apagar os arquivos gerados, basta executar o seguinte comando:

```
make clean
```

após a chamada desse comando, os arquivos gerados serão apagados e o comando *make* estará disponível novamente.

Em relação aos testes fornecidos pelo professor, eles estão dentro da pasta do projeto, em *tests*, e podem ser executados individualmente através do comando:

```
./rascal /tests/<nome do programa>
```

ou podem ser executados todos de uma vez sequencialmente através do arquivo *runTests.sh*, através do comando:

```
./runTests.sh
```