Especificação da Fase Final

Bacharelado em Ciência da Computação
Universidade Federal de São Carlos
Campus Sorocaba
Compiladores

1 Fase Final

Nesta última fase vocês realizarão a análise léxica, sintática, semântica e geração de código C para a linguagem descrita pela gramática da Seção 3, agora completa.

2 Entrega

Data: **06/06/16**

Estrutura:

Entregue um arquivo compactado .zip ou tar.gz com o formato de nome: Final_Nome1_Nome2.zip, contendo apenas os arquivos de extensão .java e testes realizados, respeitando a seguinte estrutura básica:

Final_Nome1_Nome2/AST: Classes da árvore sintática.
Final_Nome1_Nome2/AuxComp: SymbolTable, tratamento de erro, etc.
Final_Nome1_Nome2/Lexer: Classes do analisador léxico.
Final_Nome1_Nome2/Testes: Arquivos de teste (Opcional).
Final_Nome1_Nome2/Compiler.java: Arquivo principal e analisador sintático.
Final_Nome1_Nome2/Main.java: Arquivo que contém a função void main().

Em Nome poder ser NomeSobrenome, para diferenciar alunos com nomes iguais.

Para esta fase, seu compilador deve ser baseado no compilador 10 estudado na disciplina.

Seu Compiler.java deve implementar um método compile(), chamado em Main.java.

Adicione um cabeçalho com nome e RA dos integrantes da dupla em todos os arquivos *.java que forem entregues.

Você pode criar outros arquivos e pastas, respeitando esta estrutura básica, contanto que seja possível compilar via linha de comando com apenas um javac Main.java.

3 Gramática

A seguir estão listados alguns elementos da notação de descrição da gramática:

- {} A regra de produção pode ser repetida 0 ou mais vezes.
- [] A regra de produção é opcional.
- | Separa regras de produção alternativas.

As regras de produção aparecem com as letras inicias maiúsculas.

Note que os terminais da linguagens estão descritos entre aspas simples, como por exemplo ';' na regra de produção VariableDecl e os terminais que são palavras-chave da linguagem estão, também, em negrito.

Não confunda os terminais '{' '}' com o símbolo de notação { }, pois este determina o número de repetições de certa regra de produção e aquele marca o início e final de bloco. O mesmo vale para '[' ']'.

A regra CharConstant, na regra Factor, indica que seu compilador deve ser capaz de ler um caractere ASCII, como é um char em C, mas não precisa dar suporte a todos aqueles que utilizam a barra, por exemplo, um '\b'.

Já a regra StringConstant, indica que seu compilador dever ser capaz de ler um conjunto de caracteres entre aspas duplas, "", como são as constantes string em C. Uma construção como `"um exemplo de string"` deve ser aceita pelo compilador.

A gramática da linguagem está detalhada abaixo:

Program ::= Decl { Decl }
Decl ::= FunctionDecl
VariableDecl ::= Variable ';'
Variable ::= Type Ident
Type ::= StdType | ArrayType
StdType ::= '**int**' | '**double**' | '**char**'
ArrayType ::= StdType '[' IntNumber ']'
FunctionDecl ::= Type Ident '(' Formals ')' StmtBlock
| '**void**' Ident '(' Formals ')' StmtBlock
Formals ::= [Variable { ',' Variable }]
StmtBlock ::= '{' { VariableDecl } { Stmt } '}'
Stmt ::= Expr ';' | ifStmt | WhileStmt | BreakStmt
| ReturnStmt | PrintStmt
IfStmt ::= '**if**' '(' Expr ')' '{' { Stmt } '}' ['**else**' '{' { Stmt } '}']
WhileStmt ::= '**while**' '(' Expr ')' '{' { Stmt } '}'
ReturnStmt ::= '**return**' [Expr] ';'
BreakStmt ::= '**break**' ';'
PrintStmt ::= '**print**' '(' Expr { ',' Expr } ')' ';'
Expr ::= SimExpr [RelOp Expr]

SimExpr ::= [Unary] Term { AddOp Term }
Term ::= Factor { MulOp Factor }
Factor ::= LValue ':=' Expr
| LValue
| Number
| Call
| '(' Expr ')'
| '**readInteger**' '(' ')' | '**readDouble**' '(' ')' | '**readChar**' '(' ')'
| CharConstant | StringConstant
LValue ::= Ident | Ident '[' Expr ']'
Call ::= Ident '(' Actuals ')'
Actuals ::= Expr { ',' Expr }
Ident ::= Letter { '_' | Letter | Digit }
Number ::= IntNumber ['.' IntNumber]
IntNumber ::= Digit { Digit }
RelOp ::= '=' | '!=' | '<' | '<=' | '>' | '>='
AddOp ::= '+' | '-' | '||'
MulOp ::= '*' | '/' | '%' | '&&'
Unary ::= '+' | '-' | '!'
Digit ::= '0' | '1' | ... | '9'
Letter ::= 'A' | 'B' | ... | 'Z' | 'a' | 'b' | ... | 'z'

4 Análise Léxica

Nesta fase seu compilador será capaz de analisar tokens mais complexos, por exemplo, a palavra reservada '**void**', no lugar de apenas '**v**'. Portanto, fique atento às palavras reservadas que devem ser identificadas pelo seu analisador léxico.

- **Lista de palavras reservadas:**

```
void      readChar          int          if
main      readDouble        char        else
print      readInteger      double      while
break      return
```

Também não permita a utilização de palavras reservadas fora de seu contexto de uso, por exemplo, não deve haver variável com nome void.

- **Identificadores:**

Considere identificadores válidos quaisquer sequências de letras, números e sublinhado(_), mas iniciando obrigatoriamente com uma letra.

Considere também que identificadores nesta linguagem são *case-sensitive*, ou seja, IF não é o mesmo que if! O mesmo vale quando for analisar a declaração de variáveis.

- **Operadores e símbolos de pontuação:**

`+ - * / % < <= > >= := = != && || ! ; , . [] () {}`

- **Comentários:**

Comentário simples: Iniciam com um // e duram até o final da linha, da mesma forma que a linguagem C.

Em múltiplas linhas: Começam com um /* e terminam com um */.

Tudo que estiver contido no comentário deve ser ignorado pelo seu compilador, isto é, não precisa haver classe na ASA para representá-lo.

- **Tipos Básicos:**

Os tipos básicos são int, double e char.

O compilador só precisa ser capaz de identificar números em base decimal.

Para o tipo double, considere apenas números com ponto flutuante. Não é necessário se preocupar em suportar números com notação exponencial.

Para o tipo char, considere caracteres entre aspas simples, como na linguagem C.

Pode até haver suporte à '\0', '\n' e '\t', mas uma construção como 'abc' deve gerar erro!

- **String**

Não há um tipo básico para strings, elas devem ser vetores de `char` como na linguagem C. Mas seu compilador deve dar suporte às constantes string, no caso de um `print("Informe um número")`, por exemplo.

- **Observação:**

Não é necessário que seu compilador implemente explicitamente as regras de produção Ident, Number e IntNumber. Basta que o analisador léxico seja capaz de produzir os tokens equivalentes. Fica a seu critério escolher o que for mais conveniente.

5 Análise Semântica

- **Equivalência entre tipos:**

Cada variável deve estar associada a um tipo básico específico e só deve receber valores deste tipo.

Dois tipos básicos são equivalentes apenas se forem exatamente o mesmo tipo.

Não permita que um vetor seja atribuído a uma variável simples, pois em C vetores são ponteiros. Exemplo:

```
1 void main () {
2
3     int a;
4     int[10] vet;
5
6     a := vet; //Considere errado!
7     a := vet[1]; //Isto e correto
8 }
```

Listing 1: Atribuição de vetor à variavel comum

- **Atribuição:**

Quando houver uma atribuição, lembre-se de verificar se os dois lados da expressão tem o mesmo tipo.

Devido a regra Factor, pode haver identificadores, números ou expressões que aparecem sozinhas em uma linha de código, por exemplo:

a; (1+2); 10; 'c'; "erro";

Essas expressões são possíveis sintaticamente, mas são incorretas semanticamente e seu compilador deve emitir erro.

- **Variáveis:**

Para que sejam utilizadas no código, as variáveis precisam ter sido declaradas e quando forem utilizadas, deve-se respeitar o tipo com que foram declaradas.

Não esqueça que deve diferenciar letras minúsculas e maiúsculas.

Agora que há suporte à funções, variáveis declaradas como parâmetro não devem ter nome igual às variáveis declaradas no escopo do bloco da função.

- **Vetores:**

Os vetores, quando declarados, devem ter um número inteiro que define seu tamanho e tamanhos negativos devem ser proibidos.

Considere que a faixa de valores para indexar um vetor começa em 0. Sintaticamente é até possível que seja aceito o código `v[1.0] = 0;` mas deve ser tratado como um erro semântico.

- **Break:**

O comando `break` deve aparecer somente dentro de blocos de código que estejam aninhados a um bloco `while`, mesmo que indiretamente, como ocorre quando um `break` está dentro de um `if`, que por sua vez está em um `while`.

- **Print:**

O comando `print()` pode imprimir expressões de qualquer tipo básico ou as constantes `char` e `string`. Mas não deve ser utilizado sem tomar argumentos.

- **Return:**

O `return` pode ser usado dentro de qualquer função, mas o tipo de expressão retornada deve ser igual ao tipo de retorno com que a função foi declarada.

Nos casos em que a função tiver tipo de retorno `void`, o `return` pode ser usado com a expressão vazia, por exemplo, `return ;` está correto.

- **Operadores:**

A operação de negação, !, tem comportamento indefinido para o tipo double, pelo menos para o compilador gcc, então permita apenas para os tipos char e int;

O operador % também não deve ser aplicado ao tipo double;

- **Declaração de Função**

Todo programa desta linguagem deve possuir obrigatoriamente uma função `main`, sem parâmetros. Note que agora esta análise não é mais sintática. O controle sobre a existência ou não desta função passa a ser semântico.

Não permita que duas funções tenham o mesmo nome. Esta verificação é similar ao que é feito com as variáveis.

- **Chamada de Função**

Em uma chamada de função, a ordem e o tipo dos parâmetros deve ser idêntica ao do protótipo com que a função foi declarada.

Funções que estejam em atribuições devem ter tipo de retorno compatível com a variável em questão e funções do tipo `void` não devem estar em atribuições.

- **Strings**

A única operação envolvendo as constantes `string` deve ser a atribuição a um vetor do tipo `char`, mas deve haver verificação de tamanho para evitar problemas com a execução do código C. As demais operações devem gerar erro.

6 Mensagens de Erro

Nesta fase as mensagens devem ser significativas, principalmente para informar precisamente o erro semântico identificado.

Use o formato o formato a seguir para as mensagens de erro:

\n<nome do arquivo>:<número da linha de erro>:<mensagem de erro>\n<linha do código com erro>

Em <nome do arquivo> não é para conter o caminho do arquivo!

Caso não consiga adicionar <nome do arquivo> à mensagem, deixe o "Error at line" do padrão do compilador.

Em <linha do código com erro> pode manter o padrão do compilador, de imprimir o código próximo ao ponto onde o token estava no momento que o erro ocorreu.

Na ocorrência de erro, faça o compilador lançar exceção java.lang.RuntimeException com a mensagem de erro no formato citado.

7 Exemplo de Código

Um programa nesta linguagem, que seu compilador deve aceitar, está exemplificado listagem 2 abaixo:

```
1  int compara(int[100] vet1, int[100] vet2, int tam) {
2      int i;
3
4      i := 0;
5      while (i < tam) {
6
7          if (vet1[i] != vet2[i]) {
8              break;
9          }
10
11         i := i + 1;
12     }
```

```
13
14      return (i = tam);
15  }
16
17  void main() {
18      int[10] vet1;
19      int[10] vet2;
20      int tam;
21      int i;
22
23      tam := 0;
24      while(tam = 0 || tam > 10) {
25          print("Informe o tamanho dos vetores: ");
26          tam := readInteger();
27      }
28
29      print("Leitura do vetor 1\n");
30      i := 0;
31      while(i < tam) {
32          vet1[i] := readInteger();
33          i := i + 1;
34      }
35
36      print("Leitura do vetor 2\n");
37      i := 0;
38      while(i < tam) {
39          vet2[i] := readInteger();
40          i := i + 1;
41      }
42
43      if (compara(vet1, vet2, tam)) {
44          print("Vetores iguais\n");
45      } else {
46          print("Vetores diferentes\n");
47      }
48
49      return ;
50  }
```

Listing 2: Exemplo de código válido

8 Geração de Código C

Para a geração de código em C, utilize o arquivo Main.java que recebe o arquivo de entrada e o arquivo de saída via linha de comando. Implemente obrigatoriamente o método genC(), chamado em sua classe principal da AST. Porém algumas classes da AST, mais simples, podem ficar sem o método genC().

Se o código não possuir erros, escreva o arquivo correspondente em linguagem C para o arquivo de saída definido via linha de comando. Caso haja erros, imprima-os na tela no formato pedido na seção 6.

A geração de código C estará correta se o arquivo gerado com extensão .c compilar no compilador gcc. Isso implica que não serão avaliados espaçamentos e indentações em seu código. E também será possível implementar certas traduções de maneiras diferentes, sobretudo leitura e escrita.

Os testes para geração de código C quando compilados e executados produzirão saídas que serão comparadas com diff, ignorando diferenças de espaçamentos, mas é importante deixar ao menos um espaço simples entre qualquer elemento impresso para tela ou arquivo.

O código abaixo é uma possível tradução para o exemplo 2.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int compara(int vet1[100], int vet2[100], int tam) {
5      int i;
6
7      i = 0;
8      while (i < tam) {
9
10         if (vet1[i] != vet2[i]) {
11             break;
12         }
13
14         i = i + 1;
15     }
16
17     return (i == tam);
18 }
19
20 void main() {
21     int vet1[10];
22     int vet2[10];
```

```
23      int tam;
24      int i;
25
26      tam = 0;
27      while (tam == 0 || tam > 10) {
28          printf("Informe o tamanho dos vetores(limite
                =10): ");
29          scanf("%d", &tam);
30      }
31
32      printf("%s", "Leitura do vetor 1\n");
33      i = 0;
34      while(i < tam) {
35          scanf("%d", &vet1[i]);
36          i = i + 1;
37      }
38
39      printf("%s", "Leitura do vetor 2\n");
40      i = 0;
41      while(i < tam) {
42          scanf("%d", &vet2[i]);
43          i = i + 1;
44      }
45
46      if (compara(vet1, vet2, tam)) {
47          printf("%s", "Vetores iguais\n");
48      } else {
49          printf("%s", "Vetores diferentes\n");
50      }
51
52      return ;
53  }
```

Listing 3: Exemplo de código gerado por genC()