Projeto - Especificação Fase 3
COMPILADORES

1 Problema

Nesta fase final serão realizadas a análise léxica, sintática, semântica e geração de código c para a linguagem descrita pela gramática da Seção 2, agora completa.

2 Gramática

Essa seção define a gramática a ser implementada. As palavras reservadas e símbolos da linguagem são exibidos entre aspas simples. Elementos da notação da gramática:

- Uma sequência de símbolos entre { e } pode ser repetida zero ou mais vezes;
- Uma sequência de símbolos entre [e] é opcional;
- Regras de produção alternativas são separadas por |.
- Comentários devem ser iniciados por '#' e terminam com '\n'.
- Palavras reservadas não podem ser utilizadas como nomes de identificadores.
- . indica qualquer caracter.
- ^ representa a operação de potenciação.
- Esta linguagem é *case sensitive*.

Program	::= 'program' Name ':' FuncDef {FuncDef} 'end'
Name	::= Letter{Letter | Digit}
Letter	::= 'a' | ... | 'z' | 'A' | ... | 'Z'
Digit	::= '0' | ... | '9'
FuncDef	::= 'def' Name '(' [ArgsList] ')' : Type '{'Body'}'
ArgsList	::= Type NameArray {',' Type NameArray}
NameArray	::= Name['['Number']']
Body	::= [Declaration] {Stmt}
Declaration	::= Type IdList {';' Type IdList} ';'
Type	::= 'int' | 'float' | 'string' | 'boolean' | 'void'
IdList	::= NameArray {',' NameArray}
Stmt	::= SimpleStmt | CompoundStmt
SimpleStmt	::= ExprStmt | PrintStmt | BreakStmt | ReturnStmt | FuncStmt
ExprStmt	::= Name ['['Atom']'] '=' (OrTest | '['OrList']') ';'
OrTest	::= AndTest {'or' AndTest}
AndTest	::= NotTest {'and' NotTest}
NotTest	::= ['not'] Comparison
Comparison	::= Expr [CompOp Expr]
Expr	::= Term {('+' | '-') Term}
Term	::= Factor {('*' | '/') Factor}
Factor	::= [Signal] AtomExpr {'^' Factor}
AtomExpr	::= Atom [Details]
Atom	::= Name | Number | String | 'True' | 'False'
Details	::= '['(Number | Name)']' | '(' [OrList] ')'
Number	::= [Signal] Digit{Digit} ['.' Digit{Digit}]
Signal	::= '+' | '-'
String	::= "' . "'
CompOp	::= '<' | '>' | '==' | '>=' | '<=' | '<>'
OrList	::= OrTest {',' OrTest}
PrintStmt	::= 'print' OrTest {',' OrTest}';'
BreakStmt	::= 'break' ';'
ReturnStmt	::= 'return' [OrTest]';'
FuncStmt	::= Name'(' [OrList] ')";'
CompoundStmt	::= IfStmt | WhileStmt | ForStmt
IfStmt	::= 'if' OrTest '{' {Stmt} '}' ['else' '{' {Stmt} '}']
WhileStmt	::= 'while' OrTest '{' {Stmt} '}'
ForStmt	::= 'for' Name 'inrange' '('Number ',' Number')' '{' {Stmt} '}'

3 Detalhes da linguagem

- *Loop for*: inrange(x, y) possui incremento de $+1$ quando $x < y$ e decremento de -1 quando $x > y$. Exemplo de acordo com a gramática e equivalência em c:

  `for x inrange(0,3)` → `for(x=0;x<3;x++)`
  `for x inrange(3,0)` → `for(x=3;x>0;x--)`

- *Arrays* válidos na linguagem:

```
int a[3]; a=[1,2,3];
int b[3]; b[0]=1;
```

- Funções válidas na linguagem:
```
def myFirstFunc(int a, float b, boolean c, string d):int {
     return 10;
}
def mySecondFunc(int vet1[5], int v):int {
     return vet1[0];
}
def main():void {
     return ;
}
```

- Chamada de função:
```
int vet1[5], vet2[5], res;
vet1=[1,2,3,4,5];
vet2=[5,4,3,2,1];
res = myFirstFunc(3, b, 'True', d) * 10;
mySecondFunc(vet1, vet2[3]); # Passa vet1 inteiro e posição 3 de vet2
```

- Deve haver suporte no `print` para '\n' e '\t':
```
print 'Exemplo:\t10\n';
```

4 Análise léxica

O compilador produzido nesta fase deve ser capaz de analisar *tokens* mais complexos, por exemplo, a palavra reservada `print`, ao invés de `R`. Para isso é importante atenção com as palavras reservadas que devem ser identificadas pelo seu analisador léxico.

```
and      boolean  break   def      elif   else  end  False
float    for      if      inrange  int    not   or   print
program  return   string  True     while
```

Os identificadores são compostos por um conjunto de letras e números iniciando com uma letra. Exemplos de identificadores válidos: `var1, x213, myVar, Num`. Exemplos de identificadores inválidos: `123a, var#, Var$, num_1`. Outrossim, palavras reservadas são inválidas como nome de identificadores.

5 Análise semântica

- Equivalência entre tipos: Cada variável deve estar associada a um tipo básico específico e só deve receber valores deste tipo. Não deve ser permitido que uma variável do tipo `int` receba um valor do tipo `float` ou que um vetor seja atribuído a uma variável simples.

- Atribuição: Os dois lados da atribuição devem ter o mesmo tipo. Sintaticamente é possível uma atribuição do tipo: `var = 'string' ^ 2;` Esta, porém, é uma expressão inválida semanticamente e deve gerar um erro. Outra regra possível sintaticamente é possuir um *break* fora de um *loop*, isto também deve gerar um erro semântico.

- Variáveis: Devem ser declaradas antes de serem usadas no código e, por ser uma linguagem *case sensitive*, deve haver diferenciação entre maiúsculas e minúsculas.
- Vetores: Possuem tamanho definido na declaração e, apesar de sintaticamente possível, números negativos devem ser proibidos.
- *For loop*: Deve aceitar somente variáveis ou números do tipo inteiro no *inrange*, mesmo sendo sintaticamente possível o emprego de valores reais, strings e valores booleanos.
- Declaração de função: Todo programa desta linguagem deve, obrigatoriamente, possuir uma função chamada `main` do tipo `void` sem parâmetros. Não deve ser permitido que duas funções tenham o mesmo nome e nem que uma função tenha o mesmo nome que uma variavel.
- Chamada de função: A ordem dos parâmetros na chamada de uma função deve ser idêntica a do protótipo com o qual a função foi declarada. No caso do retorno de uma função ser atribuído a uma variável, ambos devem ter o mesmo tipo. Funções do tipo `void` não devem estar em atribuições.
- Returno de função: Apesar de sintaticamente possível, não deve ser permitido o emprego de uma função sem retorno. O `return` é obrigatório em **todos** os tipos de função, inclusive nas funções do tipo `void`, cujo retorno deve ser expresso da seguinte forma: `return ;`
- AtomExpr: Apesar de ser sintaticamente possível, não devem possíveis construções do gênero: `13[a]`, `'sou string'(a,b,c)`, `'True'[0]`.

6 Mensagens de erro

As mensagens de erro devem ser escritas de forma a identificar o erro encontrado pelo programa. **Devem ser escritas na tela e não em um arquivo** e devem obedecer o seguinte formato:

```
<Nome do arquivo>, <Número da linha de erro>, <Mensagem>\n
<Linha do código com erro>\n
```

Exemplo:

```
"erroSintatico4.in, 7, Falta {.\nif x < 0 \n"
```

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

Favor seguir o padrão descrito, pois a correção será automatizada. A mensagem de erro deve ser sucinta, porém o formato fica a critério da dupla.

7 Geração de código em c

Se o arquivo de teste possuir erro, este deve ser impresso na tela seguindo o formato descrito na Seção 6. Neste caso, o arquivo .c não deve ser gerado, nem mesmo vazio.

Já para os casos de testes isentos de erros, o arquivo em linguagem c será considerado correto se for possível compilá-lo com o seguinte comando:

```
gcc -o acertoSintatico1 acertoSintatico1.c
```

Vale destacar que é possível obter diferentes traduções para um mesmo arquivo de entrada. Por tanto, o que será levado em consideração na correção é a saída do mesmo. **Os arquivos .c devem produzir saídas idênticas àquelas dos arquivos de entrada, pois serão comparadas através do comando `diff`.**

8 Informações importantes

- O projeto de compiladores será composto de três fases (F_1, F_2 e F_3), sendo esta a fase 3 (fase final). A nota final do projeto será: $(F_1 + F_2 + F_3)/3$.
- Será atribuída uma nota de 0 a 10 ao trabalho. Esta será calculada automaticamente de acordo com a porcentagem de casos de testes nos quais o compilador entregue passar.
- Receberão nota **zero** os trabalhos com plágio e/ou que não compilem.
- O código deve ser implementado em Java.

9 Entrega - Fase 3

- **Data de entrega**: 26/06/2017.
- Construir a árvore AST.
- Implementar o genC.
- Implementar o analisador léxico fora do Compiler.
- Submeter um arquivo zip chamado nome1_nome2_fase3.zip com a seguinte estrutura:
  - `nome1_nome2_fase3/src`: Seu código (Main.java, Compiler.java).
  - `nome1_nome2_fase3/src/AST`: Classes da árvore sintática.
  - `nome1_nome2_fase3/src/Lexer`: Classes do analisador léxico.
  - `nome1_nome2_fase3/src/Aux`: SymbolTable.java, CompilerError.java.
  - `nome1_nome2_fase3/testes/acertos`: Arquivos de teste corretos (Opcional).
  - `nome1_nome2_fase3/testes/erros`: Arquivos de teste errados (Opcional).
  - `nome1_nome2_fase3/genC`: Arquivos .c gerados (Opcional).

  Outros arquivos e pastas podem ser criados desde que seja possível compilar com o comando dado abaixo.
- Ao descomprimir o arquivo `.zip` deverá ser criado um diretório `nome1_nome2_fase3`.
- Adicionar cabeçalho com nome e RA dos integrantes da dupla em todos os arquivos `*.java`.

- A partir da pasta `nome1_nome2_fase3`, deve ser possível compilar o projeto com o seguite comando:

  ```
  javac -classpath src/ src/Main.java
  ```

  e executar com:

  ```
  java -classpath src/ Main testes/acertos/acerto1.in genC/acerto1.c
  ```

- **O não seguimento do padrão aqui estabelecido implicará em um desconto de 0.5 na nota final desta fase.**

10 Dúvidas

Em caso de dúvidas sobre o trabalho entrar em contato com tunes.vanessa@gmail.com.