Especificação Fase 3

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

20 de Maio de 2015

1 Terceira fase

A terceira fase do trabalho consiste na implementação de um compilador que realiza a análise léxica, sintática e semântica da gramática da linguagem, além de gerar código em linguagem C.

2 Entrega

Data: **14/06/15**
Você deve entregar todos os arquivos .java com a mesma estrutura do Compilador 10, isto é, com o pacote AST, Lexer com as análises léxica, sintática e emissão de erros separadas. No diretório tests, você deve enviar todos os testes realizados.

3 Gramática

Esta seção define a gramática da linguagem. A gramática utilizada é uma versão simplificada da linguagem de programação Pascal, baseada em [1].

Em cada linha, há a descrição de uma regra de produção da gramática. As palavras-chave da linguagem estão em letras maiúsculas. Os símbolos da linguagem são mostrados entre apóstrofos. Os símbolos não-terminais da gramática são aqueles descritos por palavras em letras minúsculas.

Uma sequência de símbolos entre `[` e `]` é opcional, enquanto que uma sequência de símbolos entre `{` e `}` pode ser repetida zero ou mais vezes. Qualquer sequência de caracteres no arquivo fonte encontrado entre `{` e `}` deve ser tratado como um comentário.

É importante notar que o símbolo `.` presente na quarta regra de produção do símbolo não-terminal `factor` representa uma *string* de tamanho variável. Essa *string* pode conter qualquer caractere menos o apóstrofo, pois este é um símbolo da linguagem.

O *statement* `procfuncstmt` na regra de produção do `stmt` deve ser uma chamada a um procedimento, pois procedimentos não possuem retorno.

O `returnstmt` somente deve possuir `expr` quando dentro de uma função. Caso esteja dentro de um procedimento, não deve haver `expr`.

```
prog          ::= PROGRAM pid ';' body '.'

body          ::= [dclpart] compstmt

dclpart       ::= VAR dcls [subdcls] | subdcls

dcls          ::= dcl {dcl}

dcl           ::= idlist ':'  type ';'

idlist        ::= id {',' id}

type          ::= stdtype | arraytype

stdtype       ::= INTEGER | REAL | CHAR | STRING

arraytype     ::= ARRAY '[' intnum '..'  intnum ']' OF stdtype
```

```
subdcls        ::= subdcl {subdcl}
subdcl         ::= subhead ';' body ';'
subhead        ::= FUNCTION pid args ':'  stdtype | PROCEDURE pid args
args           ::= '(' [dcls] ')'
compstmt       ::= BEGIN stmts END
stmts          ::= stmt {';' stmt} ';'
stmt           ::= ifstmt
                 | whilestmt
                 | assignstmt
                 | compstmt
                 | readstmt
                 | writestmt
                 | writelnstmt
                 | returnstmt
                 | procfuncstmt
ifstmt         ::= IF expr THEN stmts [ELSE stmts] ENDIF
whilestmt      ::= WHILE expr DO stmts ENDWHILE
assignstmt     ::= vbl ':=' expr
readstmt       ::= READ '(' vblist ')'
writestmt      ::= WRITE '(' exprlist ')'
writelnstmt    ::= WRITELN '(' [exprlist] ')'
returnstmt     ::= RETURN [expr]
procfuncstmt   ::= pid '(' [exprlist] ')'
vblist         ::= vbl {',' vbl}
vbl            ::= id ['[' expr ']']
exprlist       ::= expr {',' expr}
expr           ::= simexp [relop expr]
simexp         ::= [unary] term {addop term}
term           ::= factor {mulop factor}
factor         ::= vbl
                 | num
                 | '(' expr ')'
                 | '"'.'"'
                 | procfuncstmt
id             ::= letter {letter | digit}
pid            ::= letter {letter | digit}
num            ::= intnum ['.'  intnum]
```

```
intnum          ::= digit {digit}

relop           ::= '=' | '<' | '>' | '<=' | '>=' | '<>'

addop           ::= '+' | '-' | OR

mulop           ::= '*' | '/' | AND | MOD | DIV

unary           ::= '+' | '-' | NOT
```

4 Instruções

O seu compilador deve se basear no Compilador 10 visto em sala de aula. Isto é, a partir de um programa descrito na gramática acima, o seu compilador deve fazer a análise léxica, sintática e semântica e gerar código em linguagem C. Observe que seu programa deve:

- gerar uma mensagem de erro no caso de uma situação que o compilador não esperava (erro léxico, sintático ou semântico presente no código fonte);
- caso contrário, gerar um arquivo de saída de extensão `.c` com o código em C equivalente ao código do arquivo de entrada.

Referências

[1] Alfred Aho, Monica Lam, Ravi Sethi e Jeffrey Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Technical, 1986.