Gustavo P. Gouveia (6482819), Victor Lassance (6431325)

Relatório de Compiladores

Segunda Etapa
Definição formal da sintaxe da linguagem de programação CZAR

Texto apresentado à Escola Politécnica da Universidade de São Paulo como requisito para a aprovação na disciplina Linguagens e Compiladores no quinto módulo acadêmico do curso de graduação em Engenharia de Computação, junto ao Departamento de Engenharia de Computação e Sistemas Digitais (PCS).

Universidade de São Paulo

Escola Politécnica

Engenharia de Computação - Curso Cooperativo

Professor: Ricardo Luis de Azevedo da Rocha

São Paulo

2013

Resumo

Este trabalho descreve a concepção e o desenvolvimento de um compilador utilizando a linguagem C. O escopo do compilador se limita a casos mais simples, porém simbólicos, e que servem ao aprendizado do processo de criação e teste de um compilador completo. A estrutura da linguagem escolhida para ser implementada se assemelha a própria estrutura do C, por facilidade de compreensão, porém com algumas peculiaridades trazidas de outras linguagens.

Palavras-chaves: Linguagens, Compiladores, Definição formal da Sintaxe.

Sumário

1 Introdução

Este projeto tem como objetivo a construção de um compilador de um só passo, dirigido por sintaxe, com analisador e reconhecedor sintático baseado em autômato de pilha estruturado.

Em um primeiro momento, foi definida uma linguagem de programação e identificados os tipos de átomos. Para cada átomo foi escrito uma gramática linear representativa da sua lei de formação e um reconhecedor para o átomo. Desse modo, as gramáticas assim escritas foram unidas e convertidas em um autômato finito, o qual foi transformado em um transdutor e implementado como sub-rotina, dando origem ao analisador léxico propriamente dito. Também foi criada uma função principal para chamar o analisador léxico e possibilitar o seu teste.

Nesta etapa, a sintaxe da linguagem, denonimada por nós de CZAR, foi definida formalmente a partir de uma definição informal e de exemplos de programas que criamos, misturando palavras-chave e conceitos de diferentes linguagens de programação. As três principais definições foram escritas na notação BNF[1], Wirth[2] e com diagramas de sintaxe.

Como material de consulta, além de sites sobre o assunto, como por exemplo um que permite verificar a definição em Wirth e criar os diagramas de sintaxe[3], foi utilizado o livro indicado pelo professor no começo das aulas (NETO, 1987), para pesquisa de conceitos e possíveis implementações.

O documento apresenta a seguir as respostas às questões propostas para a segunda etapa, assim como as considerações finais.

[1] Ver http://en.wikipedia.org/wiki/Backus_Naur_Form
[2] Ver http://en.wikipedia.org/wiki/Wirth_syntax_notation
[3] Site: http://karmin.ch/ebnf/index

2 Descrição Informal da Linguagem

O programa é composto por quatro partes, explicadas abaixo de forma simplificada, pois a linguagem será definida de forma completa nos capítulos 4 e 5 nas notações BNF e Wirth, respectivamente:

- Definição do programa:
  - `PROGRAM = IMPORTS DECLS_GLOBAIS DEF_PROCS_FUNCS DEF_MAIN.`
- Inclusão de bibliotecas:
  - `IMPORTS = { ‘<’ IDENT ‘>’ }.`
- Declaração de tipos, variáveis e constantes de escopo global:
  - `DECLS_GLOBAIS = { DEF_TIPO | DECL }.`
  - `DEF_TIPO = ‘struct’ IDENT ‘{’ { DECL } ‘}’.`
  - `DECL = [ ‘const’ ] TIPO IDENT [ ‘=’ EXPR ]`
    `{ ‘,’ IDENT [ ‘=’ EXPR ] } ‘;’.`
- Definição dos procedimentos e funções do programa, que não devem incluir o procedimento principal (chamado main):
  - `DEF_PROCS_FUNCS = { PROC | FUNC }.`
  - `FUNC = TIPO IDENT LIST_PARAMS`
    `‘{’ { INSTR_SEM_RET } "return" EXPR [ ";" ] ‘}’.`
  - `PROC = ‘void’ IDENT LIST_PARAMS ‘{’ { INSTR_SEM_RET } ‘}’.`
  - `LIST_PARAMS = ‘(’ [ [ ‘ref’ ] TIPO IDENT ]`
    `{ ‘,’ [ ‘ref’ ] TIPO IDENT } ‘)’.`
- Definição do procedimento principal (chamado main) - para fins de simplificação, a comunicação entre o programa e o ambiente externo deve ser feito através de arquivos, pois não haverá passagem de parâmetros para a função main:
  - `DEF_MAIN = ‘main’ ‘(’ ‘)’ ‘{’ [ BLOCO ] ‘}’.`

3 Exemplos de Programas na Linguagem

3.1 Exemplo Geral

```
1  <math>
2  <io>
3
4  struct nome_struct {
5    nome_struct eu_mesmo;
6    int a;
7    char b;
8  }
9
10 const int SOU_CONSTANTE_INT = 10;
11 const string SOU_CONSTANTE_STRING = "CONSTANTE_STRING";
12 string sou_variavel = "valor inicial da variavel";
13
14 void soma_como_procedimento (int a, int b, ref int soma) {
15   soma = a + b;
16 }
17
18 int soma_como_funcao (int a, int b) {
19   return a + b;
20 }
21
22 string concatena_chars(int n_chars, char[] caracteres) {
23   string retorno = "";
24   for(int i = 0; i < n_chars; i += 1) {
25       retorno += caracteres[i];
26   }
27   return retorno;
28 }
29
30 void proc_exemplo (char a, int b, int c, int d) {
31   int tmp;
32   char[32] buff;
```

```
33    soma_como_procedimento(b, c, tmp);
34    d = soma_como_funcao(tmp, c) + 5;
35    d = math_exp(SOU_CONSTANTE_INT, 2);
36    io_print(a);
37    io_int_to_str(d, buff);
38    io_print(" gives ");
39    io_print(buff);
40    io_print(" \n pointer to a is: ");
41    buff = a + "caracter";
42    io_print(buff);
43    io_print("bye");
44 }
45
46 main () {
47    proc_exemplo('x', 3, -6, -15);
48 }
```

3.2 Exemplo Fatorial

```
1  <io>
2
3  const int fat_10_rec = 10;
4  const int fat_10_iter = 10;
5  int retorno;
6
7  int fatorial_recursivo(int n) {
8      int retorno = 1;
9          if (n > 1) {
10         retorno = n * fatorial_recursivo (n - 1);
11     }
12     return retorno;
13 }
14
15 int fatorial_iterativo(int n) {
16     int fatorial = 1;
17     while (n > 0) {
18         fatorial = fatorial * n;
19         n = n - 1;
20     }
```

```
21      return fatorial;
22  }
23
24  main () {
25      retorno = fatorial_recursivo(fat_10_rec);
26
27      io_print_int(retorno);
28      io_print(" ");
29      io_print_int(fatorial_iterativo(fat_10_iter));
30  }
```

4 Descrição da Linguagem em BNF

```
1 TODO: Victor
2
3 TODO: Gustavo
```

5 Descrição da Linguagem em Wirth

```
1  PROGRAM          = IMPORTS DECLS_GLOBAIS DEF_PROCS_FUNCS DEF_MAIN
       .
2
3  IMPORTS          = { "<" IDENT ">" }.
4
5  DECLS_GLOBAIS    = { DEF_TIPO | DECL_CONST | DECL_VAR }.
6  DEF_TIPO         = "struct" IDENT "{" { DECL_CONST | DECL_VAR }
       "}".
7  DECL_CONST       = "const" DECL_VAR.
8  DECL_VAR         = TIPO IDENT [ "=" EXPR ] { "," IDENT [ "=" EXPR
       ] } ";".
9
10 DEF_PROCS_FUNCS = { PROC | FUNC }.
11 FUNC             = TIPO IDENT LIST_PARAMS "{" { INSTR_SEM_RET } "
       return" EXPR [ ";" ] "}".
12 PROC             = "void" IDENT LIST_PARAMS "{" { INSTR_SEM_RET }
       "}".
13 LIST_PARAMS      = "(" [ [ "ref" ] TIPO IDENT { "," [ "ref" ]
       TIPO IDENT } [ "," ] ] ")".
14
15 DEF_MAIN         = "main" "(" ")" "{" { INSTR_SEM_RET } "}".
16
17 INSTR_SEM_RET    = DECL_VAR | ATRIB | PROC_CALL | FLOW_CONTROL.
18 ATRIB            = ATRIB_SEM_PV ";".
19 ATRIB_SEM_PV     = VARIDENT OPER_ATRIB EXPR { "," VARIDENT "="
       EXPR }.
20 PROC_CALL        = IDENT "(" [ EXPR ] { "," EXPR } ")" ";".
21 FLOW_CONTROL     = FOR_CONTROL | WHILE_CONTROL | IF_CONTROL.
22 FOR_CONTROL      = "for" "(" DECL_VAR COND ";" ATRIB_SEM_PV ")"
       "{" { INSTR_SEM_RET } "}".
23 WHILE_CONTROL    = "while" "(" COND ")" "{" { INSTR_SEM_RET }
       "}".
24 IF_CONTROL       = "if" "(" COND ")" "{" { INSTR_SEM_RET } "}" ["
       else" "{" { INSTR_SEM_RET } "}"].
```

```
25
26 TIPO               = IDENT_COLCHETES.
27 IDENT_COLCHETES = IDENT { "[" INT "]" }.
28 VARIDENT          = IDENT_COLCHETES { "." VARIDENT}.
29
30 FUNCTION_CALL    = IDENT "(" [ EXPR ] { "," EXPR } ")".
31
32 COND               = COND_TERM { OPER_BOOL COND_TERM}.
33 COND_TERM         = "(" COND ")" | ATOMO_COND {OPER_COMP
       ATOMO_COND}.
34 ATOMO_COND        = VARIDENT | BOOL | INT | "not" ATOMO_COND.
35 BOOL               = "true" | "false".
36
37 OPER_ATRIB        = ["+" | "-" | "*" | "/" | "%"] "=".
38 OPER_BOOL         = "and" | "or".
39 OPER_COMP         = ("=" | "!" | "<" | ">") "=".
40 OPER_ARIT         = "+" | "-".
41 OPER_TERM         = "*" | "/" | "%".
42
43 EXPR               = [ OPER_ARIT ] TERM { OPER_ARIT TERM}.
44 TERM               = "(" EXPR ")" | ATOMO {OPER_TERM ATOMO}.
45 ATOMO              = [ OPER_ARIT ] FUNCTION_CALL | [ OPER_ARIT ]
       INT | STRING | CHAR | [ OPER_ARIT ] FLOAT | BOOL | [
       OPER_ARIT ] VARIDENT.
```

6 Diagrama de Sintaxe da Linguagem

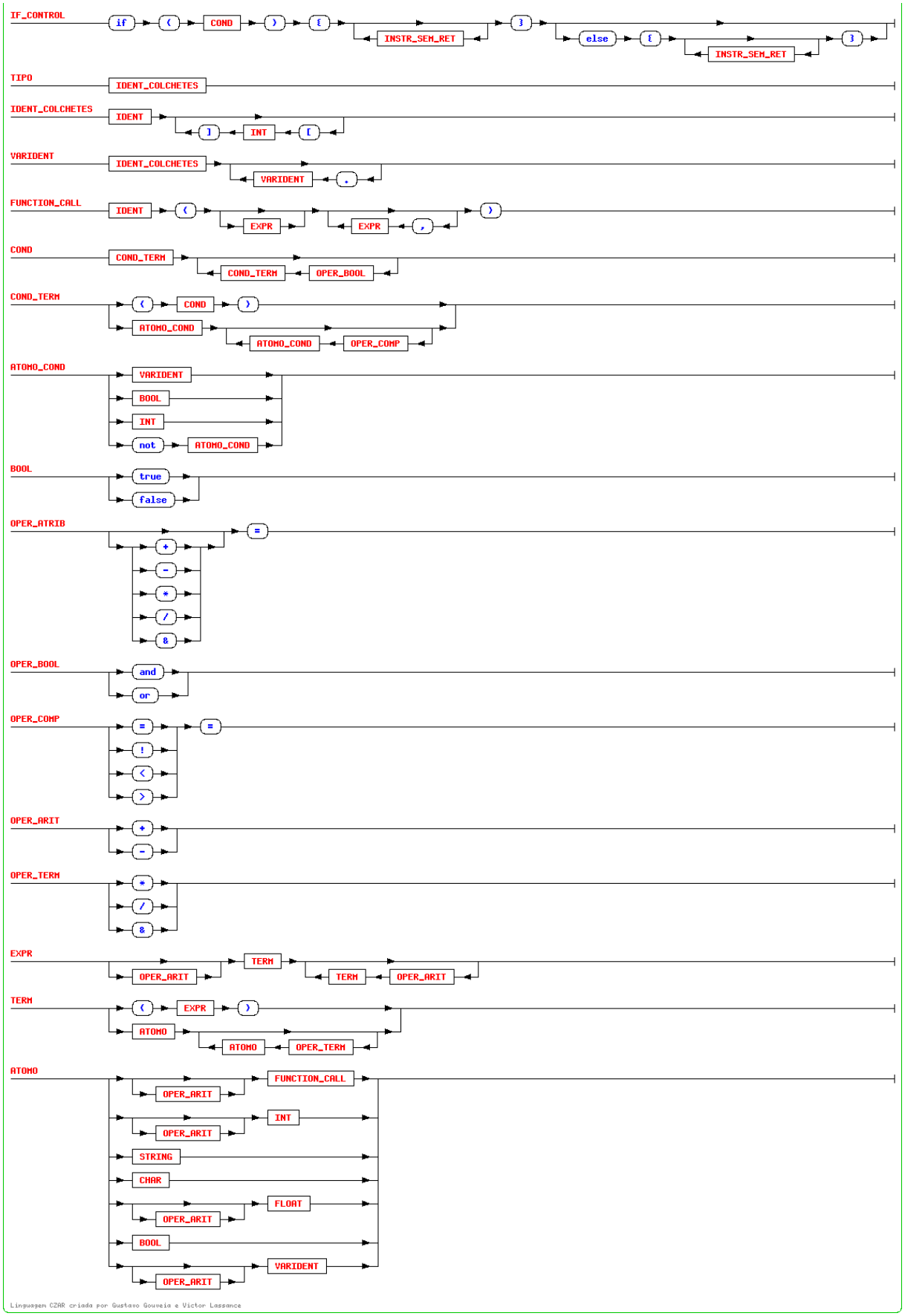

Figura 1 – Diagrama de Sintaxe da Linguagem CZAR

7 Conjunto das Palavras Reservadas

```
1  const
2  struct
3  ref
4  int
5  float
6  string
7  char
8  bool
9  for
10 while
11 if
12 else
13 and
14 or
15 not
16 true
17 false
18 main
19 return
20 void
```

8 Considerações Finais

O projeto do compilador é um projeto muito interessante, porém complexo. Desta forma, a divisão em etapas bem estruturadas permite o aprendizado e teste de cada uma das etapas. Em um primeiro momento, o foco foi no analisador léxico, o que permitiu realizar o *parse* do código e transformá-lo em tokens. Para a realização do analisador, tentamos pensar em permitir o processamento das principais classes de tokens, com o intuito de entender o funcionamento de um compilador de forma prática e didática.

Já na segunda etapa, começamos definindo a linguagem de forma mais livre e geral, partindo para a criação de exemplos de códigos escritos na nossa linguagem com todos os conceitos que deveriam ser implementados. A partir da definição informal e dos exemplos de código, criamos a definição formal na notação BNF, Wirth e com Diagramas de Sintaxe, além de atualizar a lista de palavras-chave. Essa etapa nos fez refletir sobre diversos detalhes de implementação que teremos que definir para o projeto, sendo, portanto, uma etapa crucial no desenvolvimento de um compilador.

Para as próximas etapas, espera-se continuar a atualizar o código e as definições descritas nesse documento quando for necessário, visando agregar os ensinamentos das próximas aulas.

Referências

NETO, J. J. *Introdução à Compilação*. [S.l.]: LTC, 1987. (ENGENHARIA DE COMPUTAÇÃO).