

Gerador de Código Intermediário Executável: Linguagem CStr

Eduardo Furtado — 09/0111575
Departamento de Ciência da Computação
Universidade de Brasília

4 de dezembro de 2016

1 Implementação e funcionamento do programa

Nesta iteração do projeto foi finalizado o compilador da linguagem CStr, que gera código intermediário em formato executável utilizando o TAC [1]. Está é a quinta iteração do projeto especificado na primeira entrega, que conta com um analisador léxico feito na segunda entrega, um analisador sintático construído na terceira entrega e um analisador semântico implementado na etapa anterior.

A maior parte da implementação está contida no arquivo “tac.c”, com um arquivo de código auxiliar, “infix_to_postfix.c”, e algumas alterações no arquivo “090111575.y”.

A primeira coisa feita nessa etapa foi me familiarizar com o TAC, através de sua documentação [1].

O passo seguinte foi gerar código auxiliar para a representação em quádruplas, que gerou a necessidade de adicionar novas estruturas de dados ao projeto, e que seriam usadas depois pela função que utiliza a tabela de quádruplas para imprimir o código gerado em um arquivo de saída.

Feito isso, o próximo passo foi gerar código para um programa, o mais simples possível, e depois incrementar a complexidade do código de entrada para imprimir código para a linguagem completa.

Não foi possível imprimir código intermediário executável usando o TAC para tudo o que foi proposto, por limitações da versão 0.11 do TAC [1], para tratar variáveis como ponteiros, algo necessário para implementar o retorno de funções do tipo string, e atribuições de variáveis do tipo string e concatenação de strings. Há mais detalhes na seção 4.

Mais detalhes sobre o desenvolvimento estão nas seções posteriores deste documento, assim como o detalhamento das poucas alterações que sofreu a gramática da linguagem.

Quando se executa o código e não são encontrados erros a árvore de sintaxe anotada, a tabela de símbolos e a tabela de quádruplas são impressos na saída padrão. No caso de erros encontrados a execução segue, porém são mostrados apenas erros na saída final.

O processo foi feito em uma passagem. A ideia era fazer assim, exceto se necessário fazer em duas, o que não se tornou realidade.

Durante todo o desenvolvimento escrevi este relatório e testei o código com diversos arquivos de teste, descritos em seção posterior. Como dito, estes testes orientaram o desenvolvimento durante esta etapa do projeto.

Para compilar e executar o programa foi disponibilizado um script para ambiente Unix, que pode ser executado como descrito no arquivo leia-me.txt. Este arquivo também contém informações sobre como compilar e executar em ambiente Windows.

Outra informação que também está contida no arquivo leia-me.txt é referente a como executar o código gerado, utilizando o TAC.

2 Especificação da gramática da linguagem

A gramática de CStr é baseada em uma versão reduzida da linguagem C [2], com acréscimo do tipo primitivo string e o operador de concatenação (:), e métodos de *string matching* segundo o algoritmo de Knuth-Morris-Pratt (KMP) descrito em [3].

A gramática livre de contexto a seguir descreve a linguagem CStr proposta, com modificações, onde variáveis (símbolos não-terminais) começam com letras maiúsculas, “Function” é a variável inicial e todos os outros símbolos são terminais. A barra vertical | é usada para indicar definições alternativas para um não-terminal.

A ordem em que as operações aparecem determina a precedência de cada operador, onde a primeira tem menor precedência e a última tem a maior precedência.

O problema da ambiguidade das expressões “if-else” foi abordado de acordo com um tutorial [4].

Como discutido em sala de aula, e também segundo as referências [5] e [4], foi utilizado recursão à esquerda, para evitar problemas de falta de memória para a pilha que podem acontecer com o uso de recursão à direita.

Alterações em relação a entrega anterior estariam marcadas na gramática em **amarelo**.

Elementos adicionados estão marcados em **azul**.

Elementos removidos estão riscados (~~exemplo~~) e marcados em **rosa**.

```

Begin → Function
Function → Type Identifier ( FormalArgList ) CompoundStmt
          | Function Type Identifier ( FormalArgList ) CompoundStmt
Identifier → char ( char | digit | - ) *
FormalArgList → ε
              | FormalArg
              | FormalArgList , FormalArg
FormalArg → Type Identifier
ArgList → ε
         | Arg
         | ArgList , Arg
Arg → Factor
    | FunctionCall
    | StringConcat
Declaration → Type Identifier
            | Type Attribution

```

```

Type → int
      | string
Stmt → WhileStmt
      | Expr ;
      | IfStmt
      | CompoundStmt
      | Declaration ;
      | Declaration ;
      | ReturnStmt ;
WhileStmt → while ( Expr ) CompoundStmt
IfStmt → if ( Expr ) CompoundStmt
        | if ( Expr ) CompoundStmt else CompoundStmt
CompoundStmt → { StmtList }
StmtList → StmtList Stmt
          | ε
ReturnStmt → return Factor
Expr → | Rvalue
      | Attribution
      | FunctionCall
      | StringConcat
BooleanExpr → | BooleanRvalue
              | Attribution
              | FunctionCall
              | StringConcat
Attribution → Identifier = Expr
Rvalue → Addition
        | Rvalue Compare Addition
BooleanRvalue → Rvalue Compare Rvalue
Compare → ==
          | <
          | >
          | <=
          | >=
          | !=
Addition → Addition + Multiplication
          | Addition - Multiplication
          | Multiplication
Multiplication → Multiplication * Factor
               | Multiplication / Factor
               | Term
Term → ( Term )
      | - Term
      | + Term
      | Factor
Factor → Identifier

```

	number
	text
FunctionCall →	Identifier (ArgList)
	Identifier . Identifier (ArgList)
StringConcat →	StringConcat : text
	StringConcat : Identifier
	text : text
	Identifier : text
	text : Identifier
	Identifier : Identifier

Poucas alterações foram feitas na gramática em relação à entrega anterior:

A produção da regra “ReturnStmt” foi alterada de “Expr” para “Factor”, com o intuito de poupar muito trabalho na tradução do código. A mudança não gera inconvenientes para os programadores que utilizem a linguagem pois, na prática, precisa-se apenas mover a expressão para uma linha acima do retorno, atribuir a expressão a uma variável e retornar esta variável no “ReturnStmt”.

Foi adicionado a regra “BooleanRvalue”, para reduzir a complexidade da geração de código referente as expressões que aparecem em “ifs” e “whiles”. Essa mudança fez com que a produção “Rvalue” de “BooleanExpr” fosse alterada para a nova regra, “BooleanRvalue”. Outra consequência é que a regra “Rvalue” teve a produção “Rvalue Compare Addition” removida.

3 Semântica da linguagem

A semântica da linguagem segue o que foi apresentado em iterações anteriores do trabalho. Com essa entrega, a seguinte lista foi utilizada para guiar o desenvolvimento:

- A função “main” deve estar presente e outras funções podem ser criadas;
- Os tipos são “INT” e “STRING”;
- Não tem ponteiros de variáveis ou funções;
- Tem operações de comparação entre números: “==” (igualdade), “<” (menor que), “>” (maior que), “<=” (menor ou igual que), “>=” (maior ou igual que);
- Tem operação de laço, no formato: “while (BooleanExpr) statements ”;
- Não há suporte para *overloading* de funções, portanto não é possível declarar funções com nomes repetidos;
- Não é permitido usar variáveis que não foram declaradas;
- Não é permitido chamar uma função não declarada. Para usar uma função ela tem que ser declarada antes de ser usada (aparecer antes no código);
- Não é permitido chamar um método não especificado pela linguagem (algoritmo KMP);

- Não é permitido declarar variáveis mais de uma vez, ou seja, declarar uma variável com nome já usado no escopo;
- Não é permitido chamar uma função com número inválido de argumentos, ou seja, o número de parâmetros passados deve ser corresponder ao número de parâmetros formais da função;
- Não é permitido chamar uma função com algum parâmetro com tipo que difere do tipo especificado pelo parâmetro formal;
- Métodos são relacionados apenas a variáveis do tipo “string”;
- Não é permitido chamar um método de string com número inválido de argumentos, ou seja, o número de parâmetros passados deve ser corresponder ao número de parâmetros formais do método;
- Não é permitido chamar um método de string com algum parâmetro com tipo que difere do tipo especificado pelo parâmetro formal;
- Não há *cast* entre números e strings ou entre strings e números;
- Uma função deve ter um *return statement* sempre alcançável, ou seja, se há escopos de código condicionais, deve haver um retorno no nível primário da função, ou em cada possível ramificação da função;
- O tipo de um *return statement* deve ser compatível com o tipo especificado pela função;
- Uma atribuição de variável deve ter tipo consistente com o da variável;
- Os operadores disponíveis na linguagem são: “+” (Soma), “-” (Subtração), “*” (Multiplicação), “/” (Divisão), “.” (Concatenação);
- Não é permitido fazer operações entre “INT” e “STRING” ou “STRING” e “INT” (tipos diferentes), apenas entre “INT” e “INT” ou “STRING” e “STRING” (mesmos tipos);
- Só é possível realizar a operação de concatenação com strings ou variáveis do tipo string.
- O tipo de expressões de condicionais, “if” e “while”, deve ser resolvido como inteiro - 0 (zero) para falso e diferente de 0 (zero) para verdadeiro;
- Não são permitidas atribuições em expressões de condicionais;
- Não são permitidas concatenações de strings em expressões de condicionais;

4 Código intermediário em formato executável

Para a geração de código intermediário em formato executável está sendo usado o TAC [1]. É gerado código para o que foi proposto, com exceção de alguns casos quando se tem strings envolvidas.

Seria necessário tratar variáveis como ponteiros, algo necessário para implementar o retorno de funções do tipo string, e atribuições de variáveis do tipo string e concatenação de strings.

Para implementar o retorno de função do tipo string, o código gerado iria colocar a referência para uma posição de memória na pilha, e em seguida um POP poderia ser usado para atribuir este endereço de memória a uma variável alvo.

Para implementar a atribuição de variáveis, foi implementado uma versão rudimentar, que trata a string como um vetor e troca cada elemento do vetor alvo pelo elemento correspondente do vetor fonte. Entretanto, essa solução é limitada, pois funciona apenas para strings do mesmo tamanho. Uma solução mais elegante seria trocar o endereço para o qual aponta a variável que faz referência à string.

Para implementar a concatenação de strings, seria alocado memória para uma nova string, já concatenando os dois argumentos envolvidos, e em seguida o endereço que aponta para essa nova região de memória seria utilizado para uma atribuição a variável. Não foi possível implementar assim porque não é possível tratar variáveis como endereços de memória ou alocar memória para o tipo “char” do TAC.

Para solucionar os problemas descritos acima, tentou-se todo tipo de instrução mov suportada pelo TAC (movvv, movvd, movva, movvi, movdv, movdd, movda, movdi, moviv, movid, movia). Não foi encontrado documentação específica sobre as instruções. Provavelmente isso acontece por como está implementado o modelo de memória do TAC 0.11, que utiliza a seção *heap* para armazenar as variáveis estáticas definidas na tabela de símbolos do TAC [1].

Outra tentativa foi alterar, sem acesso à documentação pertinente, o código do TAC 0.11 para implementar o que fosse necessário para solucionar os problemas descritos acima, porem sem sucesso. Por mais que o projeto do TAC tenha sido feito da mesma maneira que este, tradução dirigida por sintaxe e fazendo-se uso de um gerador de analisador LALR, não foi possível obter muito progresso no *branch* sem ter a especificação dos *opcodes*.

Uma observação é que pode-se encontrar linhas de código que funcionam como um *dummy code* antes de *labels*. Trata-se de uma instrução de mov de uma variável temporária para si mesmo, algo que efetivamente não altera a semântica de um código. Isto foi feito como *workaround* de problema identificado, em que o TAC não suportava duas linhas seguidas com *labels*, quando um era o identificador de uma função.

Ao executar o código gerado é possível ver o erro “returning to nowhere”, no código referente a um “ReturnStmnt” na função “main” do programa. Isso é mais uma limitação da versão 0.11 do TAC [1]. É possível modificar o código a mão, para trocar a instrução “return” por “println” e assim ver na saída padrão o que seria retornado pela função “main”.

5 Política de tratamento de erro adotada

Quando ocorre um erro semântico, é exibido na tela a linha em que o erro ocorreu e uma explicação do que gerou o aviso, para que o programador possa fazer a correção. O tradutor continua processando o código e mais erros são mostrados, independente se são semânticos,

sintáticos ou léxicos, e nesse caso a Árvore Abstrata, a Tabela de Símbolos e a tabela de quádruplas não são exibidos no final.

A categorização dos erros é dada conforme a lista apresentada na seção anterior.

O tratamento de erros encontrados pelo Analisador Sintático continua funcionando como na terceira entrega do projeto:

Quando ocorre um erro sintático, é exibido na tela a linha em que o erro ocorreu, qual era o(s) símbolo(s) esperado(s) e qual foi o símbolo encontrado pelo Parser.

Ao encontrar erros sintáticos a execução do Parser continua. Este mecanismo foi feito de acordo com o manual do Bison [6], ou seja, adicionando uma regra de sincronização em lugares estratégicos:

Ao encontrar um erro em “Stmt”, o Parser busca o próximo token “;” ou “{” e retoma a análise logo após este token. No caso de “CompoundStmt”, busca-se o seguinte token “}”. Quando se trata de “Function”, a procura é pelo token “}”.

Dessa maneira garante-se que os erros sintáticos serão exibidos, o que facilita o processo de *debugging*.

Outro ponto a ser explanado é que a Árvore Sintática e a Tabela de Símbolos não são exibidos ao final, quando há erros no código parseado.

O tratamento de erros encontrados pelo Analisador Léxico continua funcionando como na segunda entrega do projeto:

O analisador léxico implementado identifica erros, porém não para com a análise quando isso acontece.

Os erros são mostrados na saída, em meio aos tokens identificados, com um aviso “ERRO!”, seguido da posição do erro no código, linha e coluna), e acompanhado da classificação do erro. Com exceção de comentário não fechado e string grande demais, também é mostrado o pedaço de código referente a aquele erro.

A classificação dos erros léxicos é a seguinte:

- Comentário multilinha não fechado: blocos de código comentados com o delimitador /* e que não são fechados com o delimitador */;
- String não fechada: strings devem estar entre aspas duplas. Se não for encontrado as aspas duplas que fecham essa string, na mesma linha, um erro é identificado;
- String grande demais: foi estipulado que o tamanho máximo para strings é de 9999 caracteres;
- Identificador inválido: um exemplo de identificador inválido é aquele que começa com um número, por exemplo: “2i”;
- Identificador grande demais: foi estipulado que o tamanho máximo para identificadores é de 255 caracteres;
- Caractere inválido: um caractere inexistente na gramática da linguagem CStr. Por exemplo “‘”, pois a linguagem não suporta o tipo char.

6 Funções alteradas/introduzidas

As novas funções, desenvolvidas nessa etapa, foram escritas no arquivo “tac.c”:

- void printTacCode() - função que cria um arquivo de saída cujo nome é o mesmo do arquivo de entrada com a extensão .tac concatenada a ele. Após isso imprime no arquivo de saída a tabela de símbolos TAC, conforme informação contida na tabela TAC e por fim imprime a seção de código TAC, conforme informação contida nas quádruplas;
- int addQuadrupla(char *, char *, char *, char *) - aloca memória para uma nova entrada na no fim da tabela de quádruplas e preenche essa entrada com os dados de uma nova quádrupla. Cada parâmetro corresponde a uma coluna da linha a ser inserida. Retorna 1 se tudo ocorrer bem;
- void printQuadruplas() - função auxiliar para o desenvolvimento, imprime a tabela de quádruplas na saída padrão;
- void addTacTableEntry(char *, char *, char *, char *) - Inicializa a tabela TAC se não estiver inicializada. Em seguida aloca memória para uma nova entrada na no fim da tabela TAC e preenche essa entrada. Os parâmetros são o nome, tipo, valor e escopo da entrada a ser inserida;
- void printTacTableEntries() - função auxiliar para o desenvolvimento, imprime a tabela TAC na saída padrão;
- char * concatenateScopeWithVariable(char *, char *) - Função auxiliar para a função addTacTableEntry(). Recebe uma string com nome de escopo e uma string com o identificador de uma variável e retorna uma string com ambos concatenados, separados por “-”;
- void addReturnQuadrupla(NODE *) - chama a função addQuadrupla() para diferentes casos referentes ao tipo do fator de um “ReturnStmt”. Seu parâmetro é um nodo da árvore sintática anotada;
- char * addQuadruplasForExpressions(NODE *) - chama a função addQuadrupla() para diferentes casos referentes ao tipo de expressões. Seu parâmetro é um nodo da árvore sintática anotada. Retorna a referência para uma variável temporária com o valor da expressão resolvida. Se chama recursivamente;
- void addQuadruplaForComparison(char *, char *, char *) - chama a função addQuadrupla() para diferentes casos referentes ao tipo de expressões booleanas usadas em condicionais. Seus parâmetros são a parte esquerda da comparação, o operador de comparação e a parte direita da comparação;
- char * getLabelForLoop() - retorna um label para utilizar no código TAC referente a laços e empilha os outros labels relacionados a este laço na pilha de labels;
- char * getLabelForIfElse() - retorna um label para utilizar no código TAC referente a if/else e empilha os outros labels relacionados a if/else laço na pilha de labels;

O arquivo “infix_to_postfix.c” contém uma versão extensamente modificada de um algoritmo que transforma expressões da forma *infix* para *postfix* [7]. O algoritmo foi modificado para trabalhar com tokens separados por espaço ao invés de dígitos.

As seguintes funções foram introduzidas na etapa anterior, no arquivo “semantica.c”:

- `int isMainFunctionPresent()` - verifica se a função “main” está presente no código traduzido. Retorna 1 (um) caso a função “main” estiver presente e 0 (zero) em caso contrário;
- `char * strToLower(char *)` - função auxiliar, utilizada em alguns locais do código para alterar o caso de uma string para minúsculas;
- `char * strToUpper(char *)` - função auxiliar, utilizada em alguns locais do código para alterar o caso de uma string para maiúsculas;
- `int functionCallSemanticCheck(NODE *)` - De acordo com a especificação de uma função, verifica se uma chamada para a função tem a quantidade de argumentos corretas e se os tipos desses argumentos estão corretos. Retorna 1 (um) se tudo está OK e 0 (zero) em caso contrário;
- `int checkReturnStatementsTypes(char *, char *)` - Verifica se uma função tem ao menos um “return statement” e verifica o tipo de cada um deles. Retorna 1 (um) se os retornos estão apropriados e 0 (zero) em caso contrário;
- `int saveReturnStmtType(NODE *)` - função que ajuda a função “checkReturnStatementsTypes”. Faz isso guardando o tipo de um “return statement” sempre que for encontrado em uma função, para posterior análise. Essas funções seriam implementadas de maneira diferente, agora que tenho mais experiência com este projeto;
- `int recursivelyLookForReturnInCompoundStmt(NODE *)` - Verifica se uma função tem um *return statement* sempre alcançável, ou seja, verifica se há escopos de código condicionais e caso tenha, verifica se há um retorno no nível primário da função, ou em cada possível ramificação da função. Retorna 1 (um) caso esteja OK e 0 (zero) em caso contrário;
- `char * getRvalueType(NODE *)` - Resolve o tipo de um “Rvalue”, verifica se a operação é válida (verifica se os tipos envolvidos são compatíveis) e retorna a string correspondente. É uma função auxiliar utilizada por várias outras que fazem verificação de tipo;
- `void saveRvalueTypeRecursive(NODE *)` - Função auxiliar da função “getRvalueType”, que efetivamente resolve o tipo de um Rvalue;
- `int methodCallSemanticCheck(NODE *, char *, char *, char *)` - De acordo com as especificações presentes em [3], verifica se uma chamada para um método tem a quantidade de argumentos corretas e se os tipos desses argumentos estão corretos. Também verifica se a variável relacionada ao método é do tipo “string” e se é um método válido. Retorna 1 (um) se tudo está OK e 0 (zero) em caso contrário;
- `int identifierSemanticCheck(char *, char *, char *)` - Verifica se uma variável foi declarada ou se uma função foi especificada. Retorna 1 (um) em caso positivo e 0 (zero) em caso negativo;

- `char * getTypeByMethodIdentifier(char *)` - retorna o tipo de um método, segundo [3];
- `int getFormalArgCountByMethodIdentifier(char *)` - retorna a quantidade de argumentos formais de um método, segundo [3];
- `int fillFormalArgsTypesByMethodIdentifier(char[255][255], char *)` - Função auxiliar de “methodCallSemanticCheck”;
- `int verifyDuplicateFunction(data_for_the_symbol_table*)` - Verifica se uma função foi declarada mais de uma vez. Retorna 1 (um) caso já tenha sido declarada e 0 (zero) se não tiver sido declarada;
- `int verifyDuplicateVariable(data_for_the_symbol_table*, char *)` - Verifica se uma variável foi declarada mais de uma vez. Retorna 1 (um) caso já tenha sido declarada e 0 (zero) se não tiver sido declarada;
- `void saveExpressionType(NODE*)` - Função auxiliar que resolve o tipo de uma expressão e guarda esse tipo em uma variável global;
- `char* getTypeFromTSByIdentifier(char *, char *, char *)` - Retorna o tipo de um identificador, após busca na Tabela de Símbolos.

As seguintes funções foram introduzidas na terceira etapa:

- `void printSymbolTable()` - imprime a Tabela de Símbolos;
- `void addSymbolTableLine(data_for_the_symbol_table*)` - Cria uma nova linha na Tabela de Símbolos e aponta o elemento anterior e o que segue;
- `void setCurrentScope(const char*)` - função que ajuda na construção da tabela de símbolos que mantém registrado qual é o escopo atual;
- `NODE* addTreeNode(description_and_value_data_for_the_tree*)` - aloca memória para adicionar um novo nó à Árvore Sintática;
- `void addNodeToListOfNodeBrothers(NODE*, NODE*)` - função que auxilia a função “addChildToTreeNode”;
- `void addChildToTreeNode(NODE*, NODE*)` - insere nós na Árvore Sintática;
- `void setValueInSymbolTableEntry(char *)` - inclui na tabela de símbolos o valor de uma variável que foi inicializada na declaração;
- `void printSyntaxTree(NODE* root, int indentation_counter)` - imprime a Árvore Sintática;
- `void setLastValueByIdentifierName(char *)` - função auxiliar, necessária pela função “setValueInSymbolTableEntry” para incluir na tabela de símbolos o valor de uma variável que foi inicializada na declaração.

Para lidar com erros sintáticos a função `yerror` de tratamento de erros do Bison foi modificada para comportar-se como descrito na seção 4 deste documento.

As seguintes funções foram introduzidas na etapa do analisador léxico:

`parse_multiple_line_comments()` trata de ignorar caracteres dentro do bloco comentado e contar as colunas em cada linha e as linhas dentro do comentário. Essa função também trata do caso de erro em que o comentário multilinha não for fechado.

`parse_single_line_comments()` trata de ignorar caracteres na linha comentada e incrementa o contador de linhas.

`parse_string()` trata strings, delimitadas por aspas duplas. Identifica erros de strings maiores do que as permitidas pela linguagem (maiores do que 9999 caracteres), ou strings não fechadas. Optou-se por suportar somente strings delimitadas por aspas duplas em uma única linha.

7 Tabela de quádruplas, tabela TAC, árvore sintática e tabela de símbolos

Com esta entrega, três novas estruturas foram utilizadas.

A tabela de quádruplas guarda a representação intermediária a ser usada pelo gerador de código para imprimir o código TAC. Cada entrada na tabela é traduzida para uma ou mais instruções:

```
// Quádruplas
typedef struct quadrupla {
    int index;
    char op[255];
    char arg1[255];
    char arg2[255];
    char result[255];
    struct quadrupla *next;
} quadrupla;
```

A tabela TAC é uma tabela de símbolos utilizada pelo TAC. “Um arquivo `tac` é composto por duas seções: `table` e `code`. A seção `table` é opcional e, se presente, é utilizada para declarar uma tabela de símbolos.” [1].

[1].

```
// Tabela TAC
// “<tipo> <nome> [‘<tamanho do vetor>’] [‘=’ <inicializador>]\n”
typedef struct tac_table_entry_struct {
    char type[255]; // <tipo>
    char name[255]; // <nome> [‘<tamanho do vetor>’]
    char value[255]; // [‘=’ <inicializador>]
    int index;
    struct tac_table_entry_struct *next;
} tac_table_entry_struct;
```

A terceira nova estrutura é uma pilha para gerenciar os labels utilizados no código TAC para controle de fluxo. A solução de pilha foi a mais adequada para o problema de mais de um `while` ou `if/else` anidados.

```

// Pilha de labels
char g_tac_label_pile[255][255];
// posicao no topo da pilha
int g_label_pile_top = -1;

// funcao auxiliar. Push.
void pushPilhaTacLabels(char * elem)
{
    strcpy(g_tac_label_pile[++g_label_pile_top], elem);
}

// funcao auxiliar. Pop.
char * popTacLabel()
{
    return(g_tac_label_pile[g_label_pile_top--]);
}

```

A criação de um nó na árvore se dá quando uma regra é identificada pelo Analisador Sintático. A informação é passada através da estrutura de dados “description_and_value_data_for_the_tree”.

É alocado memória para um nó, em seguida os campos da estrutura são preenchidos com os valores lidos e é feita a conexão com os filhos a serem criados e pai do nó, de acordo com a situação: A quantidade de filhos depende da regra que resultou em um *match*.

Algumas informações presentes na Tabela de Símbolos também foram guardadas na Árvore Sintática, são elas o nome de identificadores e os valores de variáveis, quando possível. O escopo de variáveis e o tipo de funções de variáveis também estão representados na árvore.

A seguir estão as estruturas de dados relacionadas à Árvore Sintática (o texto não está marcado com acentos porque trata-se de código):

```

// Estrutura usada para adicionar um novo no na arvore
typedef struct description_and_value_data_for_the_tree {
    char description[255];
    char value[255];
} description_and_value_data_for_the_tree;

// Arvore Sintatica:
typedef struct tree_node_struct {
    char description[255];
    char value[255];
    int node_brother_count;
    struct tree_node_struct *father, *node_brothers_list, *child;
} NODE;

```

A Tabela de Símbolos é preenchida com o nome dos identificadores de funções, que sempre tem escopo global, campo de valor nulo, e tipo de retorno int ou string.

Além disso também é preenchida com o nome dos identificadores de variáveis, que sempre estão no escopo de alguma função. O campo de valor nulo, caso seja um argumento de função ou variável declarada e não inicializada, ou, no caso em que se trate de uma variável declarada e inicializada, pode contar o valor do número se for do tipo int ou um texto, se for do tipo string. Os dois tipos possíveis são int e string.

A seguir estão as estruturas de dados relacionadas à Tabela de Símbolos (o texto não está marcado com acentos porque trata-se de código):

```

// Estrutura usada para adicionar novas linhas na Tabela de Simbolos:

```

```

typedef struct data_for_the_symbol_table {
    char name[255];
    char type[255];
    char category[255];
    int isFormalArg;
} data_for_the_symbol_table;

// Tabela de Simbolos:
typedef struct symbol_table_struct {
    char name[255];
    char value[255];
    // o escopo pode ser global (no caso de funcoes) ou o nome da funcao em que
    // esta (no caso de variaveis)
    char scope[255];
    char type[255];
    char category[255];
    int isFormalArg;
    struct symbol_table_struct *previous, *next;
} symbol_table_struct;

```

8 Descrição dos arquivos de teste

Para esta etapa, foram construídos arquivos de testes com código de complexidade crescente, a fim de orientar o desenvolvimento. Estes arquivos não possuem erros léxicos, sintáticos ou semânticos, dado que os arquivos de testes feitos nas etapas anteriores também foram usados durante o desenvolvimento para *testing*.

São eles:

- tac1.cstr - A menor porção de código válido léxica, sintática e semanticamente para dar início ao desenvolvimento de impressão de código em formato executável. Basicamente a função main imediatamente retorna um inteiro;
- tac2.cstr - Código de diversas funções que retornam inteiros. Essas funções têm expressões matemáticas para fazer cálculos antes de retornar os valores calculados. Nesse ponto do desenvolvimento já era possível gerar código para funções recursivas, ou qualquer expressão matemática;
- tac3.cstr - Código feito para orientar o desenvolvimento da geração de código intermediário em formato executável relacionado a laços e laços anidados;
- tac4.cstr - Código feito para orientar o desenvolvimento da geração de código intermediário em formato executável relacionado a “if/else” e a “if” ou “if/else” anidados;
- tac5.cstr - Código feito para orientar o desenvolvimento da geração de código intermediário em formato executável relacionado a strings. O desenvolvimento teve que acabar neste ponto, por limitações da versão 0.11 do TAC [1].

Para a etapa anterior, o analisador semântico, foi construído um grande arquivo de teste, `codigo_incorreto3.cstr`, com a maior quantidade possível de erros semânticos, para orientar o

desenvolvimento:

- Linha 9: A variável 'a' não é do tipo string para fazer uma concatenação.
- Linha 9: A variável 'a' não é do tipo string para fazer uma concatenação.
- Linha 13: A variável 'b' tem tipo 'STRING', porém 'INT' tentou ser atribuído a ela.
- Linha 14: A variável 'b' tem tipo 'STRING', porém 'INT' tentou ser atribuído a ela.
- Linha 17: A variável 'b' tem tipo 'STRING', porém 'INT' tentou ser atribuído a ela.
- Linha 23: A variável 'a' não é do tipo string para fazer uma concatenação.
- Linha 23: A variável 'a' não é do tipo string para fazer uma concatenação.
- Linha 24: A variável 'a' não é do tipo string para fazer uma concatenação.
- Linha 25: A variável 'a' não é do tipo string para fazer uma concatenação.
- Linha 27: A variável 'b' tem tipo 'STRING', porém 'INT' tentou ser atribuído a ela.
- Linha 29: A função 'f0' deve retornar 'INT', porém este return statement é do tipo 'STRING'.
- Linha 37: A função 'f2' deve retornar 'INT', porém este return statement é do tipo 'STRING'.
- Linha 45: A função 'f3' deve retornar 'STRING', porém este return statement é do tipo 'INT'.
- Linha 55: A função 'f5' deve retornar 'INT', porém ela não tem nenhum return statement.
- Linha 55: A função 'f5' pode ter um ramo sem return statement (control reaches end of non-void function).
- Linha 63: não é possível fazer uma operação entre INT e STRING.
- Linha 63: não é possível fazer uma operação entre INT e STRING.
- Linha 63: A função 'f7' deve retornar 'INT', porém este return statement é do tipo 'ERROR'.
- Linha 68: não é possível fazer uma operação entre INT e STRING.
- Linha 68: não é possível fazer uma operação entre INT e STRING.
- Linha 68: A função 'f8' deve retornar 'STRING', porém este return statement é do tipo 'ERROR'.
- Linha 84: A função 'f92' deve retornar 'INT', porém este return statement é do tipo 'STRING'.
- Linha 89: A função 'f10' deve retornar 'STRING', porém este return statement é do tipo 'INT'.
- Linha 103: A função 'ifs1' pode ter um ramo sem return statement (control reaches end of non-void function).
- Linha 136: A função 'ifs4' pode ter um ramo sem return statement (control reaches end of non-void function).
- Linha 188: A função 'main' deve retornar 'INT', porém este return statement é do tipo 'STRING'.
- Linha 198: Expressões de condicionais devem ser resolvidas ao tipo 'int'.
- Linha 203: Não são permitidas concatenações de strings em expressões de condicionais.
- Linha 208: Expressões de condicionais devem ser resolvidas ao tipo 'int'.
- Linha 218: Expressões de condicionais devem ser resolvidas ao tipo 'int'.
- Linha 223: Não são permitidas atribuições em expressões de condicionais.
- Linha 230: A função 'main' deve retornar 'INT', porém este return statement é do tipo 'STRING'.
- Linha 234: A variável 'a' já foi declarada.
- Linha 237: A função 'main' deve retornar 'INT', porém este return statement é do tipo 'STRING'.

Linha 239: A função 'main' deve retornar 'INT', porém este return statement e do tipo 'STRING'.

Linha 241: A variável 'b' já foi declarada.

Linha 242: A variável 'b' não é do tipo string para fazer uma concatenação.

Linha 242: A variável 'b' não é do tipo string para fazer uma concatenação.

Linha 242: A função 'main' deve retornar 'INT', porém este return statement e do tipo 'STRING'.

Linha 246: A função 'main' deve retornar 'INT', porém este return statement e do tipo 'STRING'.

Linha 248: A função 'main' deve retornar 'INT', porém este return statement e do tipo 'STRING'.

Linha 248: A variável 'a' não é do tipo string para fazer uma concatenação.

Linha 249: A função 'main' deve retornar 'INT', porém este return statement e do tipo 'STRING'.

Linha 250: A variável 'a' tem tipo 'INT', porém 'STRING' tentou ser atribuído a ela.

Linha 250: A função 'main' deve retornar 'INT', porém este return statement e do tipo 'STRING'.

Linha 255: O método 'kmpPreprocess' requer '0' argumento(s). Foi passado '1' argumento(s).

Linha 257: A função 'main' deve retornar 'INT', porém este return statement e do tipo 'STRING'.

Linha 259: A função 'main' deve retornar 'INT', porém este return statement e do tipo 'STRING'.

Linha 261: A função 'kmpSearch' não existe.

Linha 263: A função 'juca' não existe.

Linha 264: A variável 'a' já foi declarada.

Linha 266: A função 'dois' requer '2' argumento(s). Foi passado '4' argumento(s).

Linha 267: O argumento formal 0 da função 'dois' e do tipo 'INT', porém foi passado um argumento do tipo 'STRING'.

Linha 267: O argumento formal 1 da função 'dois' e do tipo 'INT', porém foi passado um argumento do tipo 'string'.

Linha 269: O argumento formal 0 da função 'f82' e do tipo 'STRING', porém foi passado um argumento do tipo 'int'.

Linha 271: O argumento formal 0 da função 'f82' e do tipo 'STRING', porém foi passado um argumento do tipo 'INT'.

Linha 273: A função 'dois' requer '2' argumento(s). Foi passado '0' argumento(s).

Linha 276: A função 'fun' não existe.

Linha 280: A variável 'a' já foi declarada.

Linha 286: A variável 'a' não é uma string para que se possam usar o método 'função'.

Linha 286: O método 'função' não é um método valido.

Linha 291: A variável 'a' já foi declarada.

Linha 294: A variável 'a' tem tipo 'INT', porém 'STRING' tentou ser atribuído a ela.

Linha 295: A variável 'a' não é do tipo string para fazer uma concatenação.

Linha 295: A variável 'a' tem tipo 'INT', porém 'STRING' tentou ser atribuído a ela.

Linha 296: A variável 'a' não é do tipo string para fazer uma concatenação.

Linha 296: A variável 'a' tem tipo 'INT', porém 'STRING' tentou ser atribuído a ela.

Linha 297: A variável 'a' não é do tipo string para fazer uma concatenação.
Linha 297: A variável 'a' não é do tipo string para fazer uma concatenação.
Linha 297: A variável 'a' tem tipo 'INT', porém 'STRING' tentou ser atribuído a ela.

Este mesmo arquivo também tem alguns erros sintáticos:

Linha 15: unexpected NUMBER, expecting Identifier or TEXT
Linha 17: unexpected '+', expecting ';' ;
Linha 251: unexpected '+', expecting ';' ;
Linha 281: unexpected ':', expecting ';' ;
Linha 299: unexpected NUMBER, expecting Identifier or TEXT

Foi feito também o arquivo `codigo_correto3.cstr`, que se trata de uma versão modificada de `codigo_incorreto3.cstr` para não ter erros.

Foram utilizados também os arquivos de testes, com código correto, desenvolvidos para etapas anteriores:

- `codigo_correto1.cstr` - contém código que seria escrito em uma linguagem C baseada na gramática reduzida que usei de base para o projeto [2]. Este é o mesmo arquivo entregue na etapa anterior.
- `codigo_correto2.cstr` - contém código que seria escrito na linguagem CStr, ou seja, com melhor suporte para trabalhar com strings! Este é o mesmo arquivo entregue na etapa anterior.
- `teste.cstr` - contém código que seria escrito na linguagem CStr, ou seja, com melhor suporte para trabalhar com strings! Este código possui um código com o máximo de elementos sintáticos possíveis, para contemplar as possibilidades da linguagem.
- `teste2.cstr` - contém código que seria escrito na linguagem CStr, ou seja, com melhor suporte para trabalhar com strings! Este código possui um código simples, que tem como intuito visualizar bem a árvore de sintaxe e a tabela de símbolos.

Também foram entregues os arquivos de testes com problemas léxicos no código, os mesmos entregues na segunda etapa:

O arquivo `codigo_incorreto1.cstr` - contém código incorreto que seria escrito em uma linguagem C baseada na gramática reduzida que usei de base para o projeto [2].

- Na linha 3 há um identificador maior do que o permitido pela linguagem (255 caracteres);
- Na linha 4 há um identificador inválido, que começa com um caractere numérico;
- Na linha 5 há um caractere inválido, não contemplado pela gramática da linguagem;
- Na linha 6 há outro caractere inválido, não contemplado pela gramática da linguagem;

- Na linha 7 há caracteres inválidos, não contemplados pela gramática da linguagem, pois tentou-se utilizar o tipo `char`, não suportado pela linguagem;
- Na linha 8 há outro caractere inválido, não contemplado pela gramática da linguagem;
- Na linha 9 há outro caractere inválido, não contemplado pela gramática da linguagem;
- Na linha 10 há um comentário multilinha que não foi fechado, bem como um *Easter egg* para o leitor.

O arquivo `codigo_incorreto2.cstr` - contém código incorreto que seria escrito na linguagem CStr, ou seja, com melhor suporte para trabalhar com strings!

- Na linha 2 há um identificador inválido, que começa com um caractere numérico.;
- Na linha 3 há um caractere inválido, não contemplado pela gramática da linguagem, seguido de um identificador inválido, onde o programador colocou um “f” desnecessário ao final de um número do tipo `float`;
- Na linha 4 há uma string que não foi fechada, pois o programador confundiu aspas duplas com aspas simples;
- Na linha 4 há uma string maior do que os 9999 caracteres que a linguagem permite.

Do ponto de vista sintático, estes arquivos também contém erros. O arquivo `codigo_incorreto1.cstr`, contém:

- Linha 1: “syntax error, unexpected Identifier, expecting INT or STRING”
- Linha 3: “syntax error, unexpected =, expecting Identifier”
- Linha 4: “syntax error, unexpected ;, expecting Identifier”
- Linha 5: “syntax error, unexpected Identifier, expecting ;”
- Linha 5: “syntax error, unexpected ;”
- Linha 6: “syntax error, unexpected Identifier, expecting ;”
- Linha 7: “syntax error, unexpected Identifier, expecting ;”
- Linha 8: “syntax error, unexpected :, expecting ;”
- Linha 64: “syntax error, unexpected \$end”

Similarmente, o arquivo `codigo_incorreto2.cstr`, contém:

- Linha 2: “syntax error, unexpected ;, expecting Identifier”

- Linha 3: “syntax error, unexpected ., expecting ;”
- Linha 5: “syntax error, unexpected RETURN”

Do ponto de vista semântico, o arquivo `codigo_incorreto2.cstr` contém erros:

- A função `main` não existe.
- Linha 15: “A função `'funcao3'` deve retornar `'INT'`, porém ela não tem nenhum `return statement`.”
- Linha 15: “A função `'funcao3'` pode ter um ramo sem `return statement` (control reaches end of non-void function).”
- Linha 20: “A função `'funcao4'` deve retornar `'INT'`, porém ela não tem nenhum `return statement`.”
- Linha 20: “A função `'funcao4'` pode ter um ramo sem `return statement` (control reaches end of non-void function).”
- Linha 25: “A função `'funcao42'` deve retornar `'INT'`, porém ela não tem nenhum `return statement`.”
- Linha 25: “A função `'funcao42'` pode ter um ramo sem `return statement` (control reaches end of non-void function).”
- Linha 33: “A função `'funcao424'` deve retornar `'INT'`, porém ela não tem nenhum `return statement`.”
- Linha 33: “A função `'funcao424'` pode ter um ramo sem `return statement` (control reaches end of non-void function).”
- Linha 42: “A função `'funcao425'` deve retornar `'INT'`, porém ela não tem nenhum `return statement`.”
- Linha 42: “A função `'funcao425'` pode ter um ramo sem `return statement` (control reaches end of non-void function).”

9 Dificuldades encontradas

O principal problema, foi a curva de aprendizado do TAC [1]. Nem tudo pode ser implementado, por suas limitações na versão 0.11 [1], o que resultou em muita tentativa e erro sem acerto.

Para o desenvolvimento foi necessário bastante raciocínio para a tomada de decisões, o que significou em muito *testing* após cada nova funcionalidade adicionada, o que fez essa etapa ser trabalhosa.

Certamente este foi o maior projeto de desenvolvimento que fiz na universidade e provavelmente o projeto que mais envolveu teoria que já fiz na vida. Agora que está terminado sou capaz de contemplar que algo assim foi feito apenas por um grupo muito pequeno de seres humanos. Foi difícil, mas valeu a pena pelo aprendizado e pelo desenvolvimento de minhas habilidades como futuro cientista da computação.

Referências

- [1] Luciano Santos. *TAC, Interpretador de Código de Três Endereços - Manual de Referência*. Brasília, 0.11 edition, Novembro 2014.
- [2] MiniC Grammar. <http://www2.ufersa.edu.br/portal/view/uploads/setores/184/AppendixA.pdf>. [Acessado em 20 de agosto de 2016].
- [3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3 edition, 2009.
- [4] Lexical Analysis With Flex. <http://epaperpress.com/lexand yacc/index.html>. [Acessado em 5 de outubro de 2016].
- [5] Lexical Analysis With Flex. http://aquamentus.com/flex_bison.html. [Acessado em 5 de outubro de 2016].
- [6] Lexical Analysis With Flex. <https://www.gnu.org/software/bison/manual/bison.html>. [Acessado em 5 de outubro de 2016].
- [7] Dr. G T Raju. C Program for Infix to Postfix Conversion. <http://electrofriends.com/source-codes/software-programs/c/data-structures-c/c-program-for-infix-to-postfix-conversion/>. [Acessado em 26 de novembro de 2016].