# Tradutor para Linguagem C com tratamento de conjuntos

Afonso Dias de Oliveira Conceição Silva - 140055771

UnB, Universidade de Brasília

## 1 Objetivos do trabalho

O objetivo deste trabalho consiste na implementação de um tradutor para uma nova linguagem projetada para incluir facilidade no tratamento de conjuntos em programas escritos em C. Este trabalho será dividido em 4 etapas, sendo estas: Implementação do analisador léxico, Implementação do analisador sintático, Implementação do analisador semântico e Implementação do gerador de código intermediário.

## 2 Motivação

A motivação desde trabalho é a construção da linguagem descrita [Nal] na seção de Objetivos do trabalho. Para tal, foi escolhido um subconjunto da linguagem C e feita a introdução de uma nova primitiva de dados para conjuntos e de operações sobre esta nova primitiva sob conjuntos ou *SET*. A criação dessa nova estrutura de dados ajuda na facilidade de realizar operações e verificações sobre conjuntos. As demais primitivas e comandos de C têm semântica padrão. Para isso será adicionada uma nova declaração de tipo chamada *SET*, sem tipo associado aos seus elementos. A declaração de uma variável do tipo *elem* é polimórfica. Como restrição em relação à sintaxe usual de C, a declaração de variáveis não poderá ser seguida de atribuição.

Como política de tratamento de erro, o tradutor deve tentar relatar todos os erros, não podendo abortar a execução após o primeiro erro [Nal].

As novas operações sobre conjuntos são descritas a seguir:

- Atribuição: a atribuição é feita por referência.
- Pertinência: o operador infixo *in* tem dois parâmetros, uma expressão e um conjunto; o resultado da operação é verdadeiro se o elemento pertencer ao conjunto.
- Tipagem: o operador is_set tem um único parâmetro que é uma variável polimórfica; retorna verdadeiro se o tipo do elemento for *set*.
- Inclusão de elemento a um conjunto: a operação primitiva *add* tem um parâmetro, uma expressão de pertinência. Se o resultado da expressão de pertinência for verdadeiro, o conjunto na expressão não é alterado; caso contrário, o valor da expressão é incluído no conjunto.

- Remoção de elemento de um conjunto: a operação primitiva *remove* tem um parâmetro, uma expressão de pertinência. Se o resultado da expressão de pertinência for falso, o conjunto na expressão não é alterado; caso contrário, o valor da expressão é removido do conjunto.
- Seleção: a operação primitiva *exists* tem um parâmetro, uma expressão de pertinência; caso o conjunto não seja vazio, a variável da expressão de pertinência assume o valor de um elemento qualquer do conjunto; caso o conjunto seja vazio, o comportamento é indefinido.
- Iteração: o comando *forall* tem duas partes, uma expressão de pertinência e um comando ou bloco de comandos; o comando ou bloco de comandos executa após iteração por todos os elementos pertinentes ao conjunto.

## 3 Analisador Léxico

A primeira fase da implementação é fazer o analisador léxico para a linguagem descrita no objetivo do trabalho. Para a implementação será usado o gerador de analisador léxico *flex* [WE].

O analisador léxico deve receber como entrada um arquivo texto e deve ser capaz de gerar e apresentar a sequência de *tokens* referentes à linguagem sendo traduzida. Caso a entrada não pertença à linguagem, o analisador deve relatar erro, indicando localização.

### 3.1 Descrição da análise léxica

Para geração do analisador, primeiramente são criadas expressões regulares com base na definição formal da linguagem no primeiro bloco, este chamado de *declarations*. Após isto, são criadas regras, no segundo bloco, *transition rules*, para gerar ações a partir do reconhecimento das expressões regulares. Tais ações, neste programa, são necessárias para visualizar os *tokens* e seus lexemas. Para lidar com tratamentos de erros, símbolos não reconhecidos são expostos ao final da sequência de *tokens*, contendo o número da linha e da coluna em que foram encontrados.

Os tokens identificados neste analisador léxico incluem: INT (números inteiros), FLOAT (números flutuantes), STRING (cadeia de caracteres), CHAR(carácter), EMPTY (para atribuição de conjunto vazio), TYPE (tipo), OP (operadores matemáticos), ASSIGN (atribuição), RELOP (operadores relacionais), LOG (operadores lógicos), ID (identificadores). Vale ressaltar que comentários no código fonte são ignorados pelo analisador léxico para gerar a sequência de *tokens*. O código do analisador léxico se encontra no arquivo "lexical.l".

## 4 Analisador Sintático

A segunda fase do projeto é realizar a implementação do analisador sintático, que é o responsável pela análise sintática do código. O analisador sintático utiliza

de *Tokens* pré determinados e de regras descritas em uma gramática livre de contexto para formar uma árvore da estrutura gramatical [ALSU07]. Para a construção do nosso analisador utilizaremos da ferramenta *Bison* [Cor].

### 4.1 Descrição da análise sintática

Para a construção do analisador, primeiramente precisamos adaptar nosso analisador léxico para enviar *Tokens* para o prgrama gerado pelo *Bison*. Esses *Token* por sua vez serão especificados no arquivo de cabeçalho gerado pelo *Bison* ao compilar nosso código e passado pela estrutura *yyval* dentro da variável *token* descrita na diretiva *union*.

Com a primeira parte feita, agora podemos classificar nossos *Tokens*, criar nossas regras e nossas estruturas de dados para representar a árvore, a tabela de símbolos e o escopo. As regras de produção são as mesmas regras descritas na nossa GLC, as variáveis da gramática seguirão as regras de derivação até chegar nas folhas ou terminais, que serão nossos *Tokens*. A variável inicial será o nosso *program*.

Tendo nossas regras descritas, o próximo passo é a criação da nossa árvore sintática. A cada redução realizada, é adicionado um novo nó da árvore seguindo a abordagem aprendida em aula chamada *bottom-up*. Essa abordagem é a criação da árvore feita de baixo para cima, ou seja, das folhas até a raiz. Além da criação da nossa árvore, também é feita a construção da tabela de símbolos.

### 4.2 Implementação da árvore sintática

Para a criação da AST, criamos utilizando a diretiva *%type* as variáveis que são os identificadores das nossas regras da GLC. Essas variáveis são ponteiros de *struct* dos nós da árvore que implementamos junto da diretiva *%union*, essa por sua vez faz com que possamos sobrescrever a estrutura do *yyval* (estrutura utilizada na integração entre o *flex* e o *bison*). Nosso ponteiro de nós de árvore possui uma estrutura recursiva para poder apontar para seus filhos, nós os chamamos de *left* e *right*, que são respectivamente as derivações a esquerda e a direita daquele determinado nó, e eles são representados como ponteiros para nós. Além disso, a estrutura do nó também possui uma variável para guardar o tipo e o valor do nó em questão.

Para realizar a atribuição de endereços aos ponteiros dos nós, utilizamos dos ponteiros $$ e $$i, onde i são números naturais inteiros e positivos, disponibilizados pelo *Bison* durante a aplicação das regras de transição. Por exemplo, se temos uma regra que se deriva em *VAR + VAR* os ponteiros para cada terminal seriam respectivamente $1, $2 e $3, e o $$ é utilizado para enviar esses nós da árvore para o escopo de cima da derivação, caso seja necessário, afim de conseguir estruturar a árvore completa no $$ da variável inicial.

Para mostrar a árvore, foi criada a função *print parser tree*, essa função realiza uma busca em largura mostrando as informações do nó atual, seu filho a esquerda e depois o da direita. A cada altura da árvore é impresso um sinal de espaçamento para melhor visualização.

### 4.3 Implementação da tabela de símbolos

A tabela de símbolos foi implementada por meio de uma *hash table* utilizando a biblioteca *uthash* [Han] para facilitar a busca por símbolos caso seja necessário. A tabela vai sendo preenchida por meio das regras de transição encontradas durante a análise do código fonte. As informações contidas nela são:

- *Key*: (concatenação do nome do símbolo e do escopo que a função se encontra)
- *Name*: Nome do símbolo
- *Type*: (int — float — elem — set ).
- *SymbolType*: Tipo do objeto (variável, parâmetro ou função).
- *Scope*: (Recebe o nome da função em que foi declarada ou global)
- *Params*: (Recebe a lista de parâmetros – tipo e nome – de uma função)
- *Handle* necessário para a criação da *hash table* segundo a implementação do uthash.

A função *print symbol table* foi adicionada no código para imprimir a tabela de símbolos ao final da execução do programa, informando símbolo e as demais informações descritas na estrutura acima.

## 5 Analisador Semântico

A terceira parte do tradutor é a análise semântica. O analisador semântico utiliza de informações da árvore e da tabela de símbolos para checar a consistência do código de acordo com a definição da linguagem [ALSU07].

### 5.1 Implementação do analisador semântico

Para a implementação dessa parte do tradutor, foi decidido fazer o processo de tradução em um passo. Portanto, tornou-se necessária a implementação de funções adicionais para a checagem sintática do código à medida que a construção da árvore é chamado.

Para realizar a checagem, serão adicionadas as seguintes regras:

- Erro de redeclaração: símbolo com mesmo nome no mesmo escopo;
- Erro de não declaração: símbolo utilizado não foi declarado anteriormente;
- Erro tipos incompatíveis: operando da esquerda e da direita com tipos incompatíveis;
- Erro de tipo de retorno: return com tipo diferente do da função;
- Tipos incompatíveis com operadores aritméticos: utilização de algum operando de tipo incompatível com operador aritmético;
- Passagem de parâmetros de tipo incompatível: parâmetros passados de tipos distintos do da declaração da função;
- Erro de operações aritméticas com set: Inviabilizar operações do tipo SET.
- Erro de número de argumentos: o número de argumentos passado na chamada da função é diferente do número de parâmetros esperado;
- Erro de main não declarada: caso a função main não seja encontrada na tabela de símbolos, o erro é gerado;

### 5.2 Funções e tratamentos de erros semânticos

Para cada erro semântico foi feita uma nova lógica para realizar sua checagem. A seguir iremos comentar a solução proposta para cada lógica realizada.

**Erro de redeclaração:** Para esse erro foi utilizada uma nova checagem na função de adicionar símbolo na tabela. Utilizamos da função *HASH_FIND_STR* da biblioteca *Uthash* [Han] para procurar se o símbolo que estamos analisando já existe nela. Caso não exista, criamos e adicionamos na tabela, se já existe, relatamos um erro semântico de redeclaração de variável.

**Erro de não declaração:** Para esse erro foi criada uma nova função para procurarmos um símbolo na tabela utilizando a função *HASH_FIND_STR* da biblioteca *Uthash* [Han] e retornarmos caso encontrado. Adicionamos nas regras de derivação que possuem *ID* a chamada dessa nova função. Caso essa nova função não encontre o símbolo sendo analisado, relatamos um erro de variável não declarada, caso ache, seguimos a execução criando o nó da árvore e seguindo em frente.

**Erro de main não declarada:** Para esse erro apenas utilizamos da função *HASH_FIND_STR* da biblioteca *Uthash* [Han] para procurar se na pilha do escopo existe a função main.

**Erro de tipos incompatíveis em operações:** Para esse erro criamos uma função chamada *define_type* onde ela irá verificar o tipo do nó da esquerda e da direita. Caso esses tipos não sejam vazios, iremos validar se esses tipos fazem parte da conversão implícita. Se não fizerem parte, ou seja, não forem do tipo elem, int ou float, relatamos o erro. Caso sejam, chamamos a nova função *implicit_conversion* que irá ver o par de tipos e criar um novo nó do tipo a esquerda da operação.

**Erro de passagem de parâmetros de tipo incompatível:** Para esse erro criamos uma nova estrutura de dados para armazenar a lista de argumentos, além disso criamos uma função para checar se o tipo que está sendo passado está correto fazendo uma comparação com a tabela de símbolos. Caso a lista de argumentos não seja vazia, comparamos um a um com os argumentos presentes na tabela de símbolos. Caso esses tipos façam partes dos tipos com conversão, chamamos a função *implicit_args_conversion* que tem a lógica semelhante ao comentado na regra anterior. Caso o tipo do argumento passado não faça parte da conversão, relatamos o erro.

**Erro de número de argumentos** Para esse erro utilizamos parte da explicação do erro anterior, quando estamos fazendo a checagem do tipo dos argumentos comparando com a tabela de símbolos, temos um contador para o argumento e o parâmetro. Caso esses contadores fiquem com valores diferentes, relatamos o erro.

**Erro do tipo do retorno** Para esse erro chamamos uma função de checagem do tipo do retorno dentro da regra de transição do *RETURN_STMT*. Nessa checagem procuramos na tabela de símbolo pela função que está tendo seu retorno analisado. Comparamos o seu retorno com o tipo da função, caso não sejam do mesmo tipo, tentamos aplicar a conversão implícita. Caso não seja possível a conversão, relatamos o erro.

**Erro de operações aritméticas com set** Para esse erro apenas verificamos na função *define_type* se a operação que estamos validando é aritmética e se algum dos dois nós analisados é do tipo Set, caso seja relatamos o erro.

## 6 Gerador de Código intermediário

Nessa última parte do trabalho iremos utilizar todas as partes feitas anteriormente e implementar um gerador código intermediário utilizando a ferramenta TAC [LS].

TAC, ou o interpretador de três endereços, é composto por uma sequência de instruções envolvendo operações binárias ou unárias e uma atribuição. O nome "três endereços"está associado à especificação, em uma instrução, de no máximo três variáveis: duas para os operadores binários e uma para o resultado. Assim, expressões envolvendo diversas operações são decompostas nesse código em uma série de instruções, eventualmente com a utilização de variáveis temporárias introduzidas na tradução. Dessa forma, obtém-se um código mais próximo da estrutura da linguagem *assembly*. Nesse trabalho iremos usar uma ferramenta que irá interpretar o código intermediário que estamos gerando no fim da análise do código fonte.

Nessa parte foi adicionado um campo chamado *instruction* na estrutura de dados do nó da árvore. Esse campo está servindo como auxiliar para guardar a respectiva função do *assembly* que será escrita ao analisar aquele nó. Por exemplo, quando chega uma expressão aritmética, analisamos que tipo de expressão é (soma, subtração, multiplicação ou divisão), caso seja uma soma, colocamos no campo *instruction* o texto *"add"* para ser utilizada na hora da geração do código intermediário. Além dessa mudança na estrutura do nó, foi implementado uma função nova para criar o arquivo TAC, escrever a parte da *.table* recuperando as variáveis que estão na tabela de símbolos, uma função para escrever a parte do *.code* que irá verificar o tipo do nó que está sendo analisando e escrever no arquivo TAC a respectiva instrução em assembly. Também temos uma nova função recursiva que analisa expressões aritméticas sendo elas simples ou não. Para a partes dos registradores que serão utilizados, foi criado uma variável global que servirá de contador para o uso dos registradores. Por fim, essa geração do código intermediário foi feito em duas passagens, ou seja, a gente faz a geração do código após ter a árvore sintática já criada. Vale ressaltar que o TAC não é criado em casos de erros.

Nesse trabalho foi implementado o gerador de código intermediário para funções aritméticas e para as funções *write* e *writeln* com CHAR, STRING ou Variáveis. Ainda não foi feita a parte de conversão implícita de tipo, porém está feita no semântico

### 6.1 Gerador de código intermediário para SET e ELEM

Para gerar o código intermediário do ELEM, podemos fazer a conversão implícita sempre que necessário. Para o SET, já que é uma estrutura nova que

pode ter seu tamanho alterado, precisaremos de um contador para cada declaração de variável do tipo SET. Toda vez que uma função é chamada com essa variável como argumento, esse contador será incrementado. Dessa forma, é possível saber na parte da geração de código qual será o tamanho dessas variáveis. Como esse contador, podemos criar na seção de *.table* essas variáveis SET com seus tamanhos.

## 7 Dificuldades e melhorias

Com ajuda do laboratório consegui localizar os locais que estão acontecendo os 6 shifts/reduce, porém não consegui removê-los sem causar outros erros.

Ainda estou tendo dificuldade na criação dos escopos de if, else e forall. Não foi implementado o for simples. Apenas foi implementado a parte as função aritméticas no TAC e as função write e writeln com CHAR, STRING ou Variáveis

## 8 Instruções de execução e testes

Para execução do analisador deve-se:

- Garantir que o seu flex está na versão 2.6.4 ou acima;
- Garantir que o seu Bison está na versão 3.0.4 ou acima;
- Executar o *make*;
- executar o seguinte comando: *./program.out* e após esse comando (na mesma linha), passar o arquivo a ser analisado que se encontra dentro da pasta exemplos;
- exemplo de execução: ./program.out ./exemplos/example1.txt;
- Para executar o TAC faça: ./tac ./code.tac;

Para essa entrega serão utilizados 7 arquivos de teste:

- o *example.txt* e o *example1.txt*, onde são os arquivos sem erros;
- o *example2.txt* e *example3.txt* que são os arquivos que possuem erros todos os tipos de erros;
  - símbolo não reconhecido na linha 11 coluna 5 e erro de uso inadequado de parêntese e ponto e vírgula na linha 11 coluna 11;
  - Erro de tipo do retorno na linha 4 coluna 2; símbolo não reconhecido na linha 7 e 8 coluna 9; operação com set na linha 11 coluna 6; argumento do tipo errado na linha 12 coluna 18; variável do tipo int recebendo returno de função do tipo set na linha 13 coluna 11;
- o *example4.txt* e *example5.txt* que são os arquivos que possuem erros semânticos;
  - Erro de variáveis não declaradas na linha 4 coluna 15, 5 coluna 21, 7 coluna 24 e 8 coluna 25;
  - Erro de variável redeclarada na linha 16 coluna 8 e 17 coluna 2;
- Foi criado o *exampleMaster.txt* que contém todos os códigos de exemplo dados na descrição da linguagem com pequenas alterações para não ter mais de uma *main*. Esse arquivo não contém erros.
- Foi criado o *exampleTac.txt* para agrupar as funções que estão gerando o código intermediário.

## Referências

[ALSU07] A Aho, M Lam, R Sethi, and J Ullman. Compilers: Principles, techniques and tools, addison-wesley 2nd edition. 2007.

[Cor] Robert Corbett. Gnu bison. [Online; acessado 12-Março-2021; url: https://www.gnu.org/software/bison/].

[Deg] Jutta Degener. Ansi C yacc grammar. [Online; acessado 18-fevereiro-2021, url: https://www.lysator.liu.se/c/ANSI-C-grammar-y.html ].

[Han] Troy D. Hanson. uthash. [Online; acessado 04-Abril-2021; url: https://github.com/troydhanson/uthash].

[LS] Cláudia Nalon Luciano Santos. Tac - the three address code interpreter. [Online; acessado 05-Maio-2021; url: https://github.com/lhsantos/tac].

[Nal] Cláudia Nalon. Objetivos do trabalho de tradutores. [Online; acessado 04-Abril-2021; url: https://aprender3.unb.br/mod/page/view.php?id=294131].

[WE] Vern Paxson Will Estes. Flex. [Online; acessado 16-fevereiro-2021; url: https://github.com/westes/flex].

## 9 Gramática

Gramática baseada na linguagem C exemplificada na referência [Deg] , e adicionando as novas funções e propriedades de conjunto.

1 < *program* >:< *declaration-list* >
2 < *declaration-list* >:< *declaration-list* >< *var-declaration* > | < *declaration-list* > < *func* > | < *var-declaration* > | < *func* > | $\varepsilon$
3 < *var-declaration* >: **TYPE ID ;**
4 < *func* >: **TYPE ID (** < *params-list* > **)** < *comp-statement* >
5 < *params-list* >:< *params* > | $\varepsilon$
6 < *params* >:< *params* > **, TYPE ID** | **TYPE ID**
7 < *comp-statement* >: **{** < *statement-list* > **}**
8 < *comp-inline* >: < *comp-statement* > | < *statement* >
9 < *statement-list* >:< *statement-list statement* > | $\varepsilon$
10 < *statement* >: < *variable−declaration* > | < *expr* > | < *conditional-statement* > | < *iteration-statement* > | < *return-statement* > | < *set-func* > | < *read-statement* > | < *write-statement* > | < *writeln-statement* >
11 < *variable − declaration* >: **TYPE ID**
12 < *read-statement* >: **READ**(< *var* >);
13 < *write-statement* >: **WRITE**(**STR**); | **WRITE**(**CHAR**); | **WRITE**(< *term* >);
14 < *writeln-statement* >: **WRITELN**(**STR**); | **WRITELN**(**CHAR**); | **WRITELN**(< *term* >);
15 < *set-func* >: **ADD (** < *in-statement* > **)** | **REMOVE (** < *in-statement* > **)** | **IS_SET (** < *in-statement* > **)** | **EXISTS (** < *in-statement* > **)**
16 < *in-statement* >: < *simple-expr* > **IN** < *simple-expr* >
17 < *exp* >:< *var* > **=** < *expr* > | < *simple-expr* > **;**
18 < *simple-expr* >:< *op-expr* >< *relop op-expr* > | < *op-expr* > | < *in-statement* >
19 < *conditional-statement* >: **IF (** < *simple − expr* > **)** < *comp-inline* > | **IF (** < *simple − expr* > **)** < *comp-inline* > **ELSE** < *comp-inline* >
20 < *iteration-statement* >: **FORALL (** < *in − statement* > **)** < *comp-inline* > | **FOR (** < *for − expression* > **)** < *comp-inline* >
21 < *for-expression* >:< *variable − declaration* > **;** < *simple − expr* > **;** < *simple − expr* > **;**
22 < *var* >: **ID**
23 < *return-statement* >: **RETURN** < *simple − expr* > **;** | **RETURN ;**
24 < *var* >: **ID**
25 < *op-expr* >:< *op-expr* >< *op* >< *term* > |< *op-expr* >< *log* >< *term* > | < *not* >< *term* > | < *term* >
26 < *relop* >:< *LE* > | < *L* > | < *G* > | < *GE* > | < *EQ* > | < *NE* >
27 < *LE* >: **<=**
28 < *L* >: **<**
29 < *G* >: **>**
30 < *GE* >: **>=**
31 < *EQ* >: **==**
32 < *NE* >: **!=**
33 < *op* >:< *add − op* > | < *sub* > | < *div* > | < *mult* >
34 < *add − op* >: **+**
35 < *sub* >: **-**

36 $< div >$: /
37 $< mult >$: *
38 $< log >:< and > \mid < or > \mid < not >$
39 $< and >$: &&
40 $< or >$: ||
41 $< not >$: !
42 $< term >$: ( $< simple\text{-}expr >$ ) | $< var > \mid < call >$ | **INT** | **FLOAT** | **ELEM** | **SET** | **EMPTY**
43 $< call >$: **ID** ( $< args >$ )
44 $< args >:< arg\text{-}list > \mid \varepsilon$
45 $< arg\text{-}list >:< simple\text{-}expr >$ , $< arg\text{-}list > \mid < simple\text{-}expr >$

**ID** = $< letter > (< letter > \mid < digit > \mid < underscore >)^*$
**INT** = $-?< digit >^+$
**FLOAT** = $-?(< digit >)^+.(< digit >)^+$
**TYPE** = $(int \mid float \mid set \mid elem)$
**STR** = $[\wedge"]^*$
**CHAR** = $[\wedge']^+$
**QUOTES** = "
$< letter >= a \mid ... \mid z \mid A \mid ... \mid Z$
$< digit >= 0 \mid ... \mid 9$
$< whitespace >= \backslash n \mid \mid \backslash t \mid \backslash r$
$< comment >= /*.^**/ \mid //.*$
$< underscore >= \_$