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

2.1 Motivação

A motivação desde trabalho é a construção da linguagem descrita na seção de Objetivos do trabalho [Nal]. 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 adicionado 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.

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.

2.2 Descrição da análise léxica

Para implementação 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), 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".

3 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 do contexto para formar uma árvore da estrutura gramatical [ALSU07]. Para a construção do nosso analisador utilizaremos da ferramenta *Bison*[Cor].

3.1 Descrição da análise sintática

Para a construção do analisador, primeiramente precisamos adaptar nosso analisador léxico para enviar *Tokens* para nosso programa em Bison. Esses *Token* por sua vez serão especificados no arquivo de cabeçalho gerado pelo *Bison* ao compilar nosso código e passado pela estrutura *yyvall* 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 transição 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 nossa regras feitas, o próximo passo é a criação da nossa árvore sintática. A cada derivação 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.

3.2 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 seria 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.

3.3 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 o *parse*. 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.

4 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].

4.1 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 o parser de construção da árvore é chamado.

Para realizar a checagem, será adicionado 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 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;

4.2 Funções e tratamentos de erros semânticos

Para cada erro semântico foi feito 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 existe, criamos e adicionamos na tabela, se já existe, subimos 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, subimos um erro de variável não declarada, caso ache, seguimos a execução criando o nó da árvore e seguindo em frente.

DEMAIS REGRAS AINDA EM DESENVOLVIMENTO

5 Dificuldades e melhorias

Até o momento estou tendo dificuldades com as ambiguidades da gramática. Não estou conseguindo retirar as warnings mostradas pelo Bison. Existem 6 shift/reduce para tratar.

Apenas criando escopo de funções, ainda não está sendo criado escopo para If, Else, For e afins.

erros semânticos implementados até o momento: Redeclaração de variável e utilização de variável não declarada.

6 Instruções de execução e testes

Para execução do analisador deve-se:

- 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 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 sintáticos e léxicos;
- o *example4.txt* e *example5.txt* que são os arquivos que possuem erros semânticos;
- 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*.

Referências

[ALSU07] A Aho, M Lam, R Sethi, and J Ullman. Compilers: Principles, techniques and tools. 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 20-Novembro-2020; url: https://github.com/troydhanson/uthash].

[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].

7 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* > | ε
3 < *var-declaration* >: **TYPE ID ;**
4 < *func* >: **TYPE ID (** < *params-list* > **)** < *comp-statement* >
5 < *params-list* >:< *params* > | ε
6 < *params* >:< *params* > **,** **TYPE ID** | **TYPE ID**
7 < *comp-statement* >: **{** < *statement-list* > **}**
8 < *comp-inline* >: < *comp-statement* > | < *statement* >
9 < *statement-list* >:< *statement-list statement* > | ε
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**(< *var* >);
14 < *writeln-statement* >: **WRITELN**(**STR**); | **WRITELN**(**CHAR**); | **WRITELN**(< *var* >);
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* >
21 < *return-statement* >: **RETURN** < *simple − expr* > **;** | **RETURN ;**
22 < *var* >: **ID**
23 < *relop* >: **<=** | **<** | **>** | **>=** | **==** | **!=**
24 < *op-expr* >:< *op-expr* > < *op* > < *term* > | < *op-expr* > < *log* > < *term* > | < *term* >
25 < *op* >: **+** | **-** | **/** | *****
26 < *log* >: && | || | !
27 < *term* >: **(** < *simple-expr* > **)** | < *var* > | < *call* > | **INT** | **FLOAT** | **ELEM** | **SET** | **EMPTY**
28 < *call* >: **ID (** < *args* > **)**
29 < *args* >:< *arg-list* > | ε
30 < *arg-list* >:< *simple-expr* > **,** < *arg-list* > | < *simple-expr* >

ID = < *letter* > (< *letter* > | < *digit* > | < *underscore* >)*
INT = $-?$ < *digit* >$^+$
FLOAT = $-?$(< *digit* >)$^+$.(< *digit* >)$^+$
TYPE = (*int* | *float* | *set* | *elem*)
STR = [$\wedge$"]*
CHAR = [$\wedge'$]$^+$

QUOTES = ”
$< letter >= a \mid ... \mid z \mid A \mid ... \mid Z$
$< digit >= 0 \mid ... \mid 9$
$< whitespace >= \backslash n \mid \mid \backslash t \mid \backslash r$
$< comment >= /*.^**/ \mid //.*$
$< underscore >= \_$