Gerador de Código Intermediário

Lucas Mafra Chagas - 12/0126443

Prof.ª Dra. Cláudia Nalon

Resumo Esse projeto tem como intuito construir um analisador sintático para um subconjunto da linguagem C, mostrando sua nova primitiva de linguagem, sua descrição, instrução e gramática.

Keywords: Tradutores · Compiladores · Mini-C.

1 Objetivo

Tendo como base a metodologia de ensino baseada em projeto, determinou-se algumas diretrizes que o discente da Ciência da Computação necessita seguir para alcançar um melhor entendimento do conteúdo apresentado. Dessa forma, as diretrizes definidas consistem na implementação de um analisador de um subconjunto da linguagem C. Por fim, este projeto é constituído por quatro etapas: Implementação do analisador léxico; Implementação do analisador sintático; Implementação do analisador semântico; e Implementação do gerador de código intermediário.

2 Motivação

Um compilador é um programa que lê um programa escrito numa linguagem, conhecida como linguagem fonte, e o traduz num programa equivalente numa outra linguagem, conhecida como linguagem alvo. Essa ação é feita observando possíveis erros presentes no programa fonte[1].

Sendo assim, a linguagem projetada para o desenvolvimento do trabalho final segue as tratativas básicas da linguagem C e a adição de uma nova primitiva de dados, que é declarado como *set*. A declaração de uma variável desse tipo segue o molde de C, sem ter um tipo associado aos seus elementos, onde o *set* tem como papel ser composto por *elem*, onde elem é polimórfico, podendo assumir o papel de *int*, *float* ou de *set* [6].

Além disso, o novo tipo de dado da linguagem, o *set* possui operações específicas, como verificação de tipagem, *is_ set*, a inclusão de elemento em um conjunto, *add*, a remoção de elemento de um conjunto, *remove*, a seleção dentro desse conjunto, *exists*, e a iteração dentro desse conjunto, *forall* [6].

3 Descrição da Análise Léxica

A análise léxica é a primeira fase de um projeto de compilador. O analisador léxico é responsável por converter uma sequência de caracteres em uma sequência de lexemas, removendo qualquer comentário ou espaço em branco. O analisador léxico é responsável por varrer o código fonte do programa e identificar cada token, um por um [7].Durante essa fase, para uma melhor compreensão da linguagem desenvolvida, é possível construir uma gramática livre de contexto que determina o seu funcionamento. Além da gramática, a construção de uma tabela de símbolos classificando os nomes das variáveis, as funções, as classes e as constantes também auxilia no compreendimento [5].

No projeto desenvolvido, além das regras e expressões regulares, o analisador léxico necessitou da estruturação de algumas funcionalidades para a preparação do algoritmo. Essas novas estruturas servem de base para a construção da tabela de símbolos na próxima etapa. Para o tratamento de erros, foi construída uma função que destaca um token inválido ao ser encontrado, mostrando a linha e a coluna onde esse erro aconteceu.

4 Descrição da Análise Sintática

A análise sintática é a segunda fase de um projeto de compilador. O analisador sintático é responsável por garantir que os lexemas identificados pelo analisador léxico sejam descritos de forma adequada aos seus programas [2]. Os tokens, que vem agrupados nos lexemas, possuem dois componentes: os seus nomes e seus valores de atributos. Os nomes dos tokens aparecem como símbolos terminais da gramática da linguagem, e seus atributos direcionam para a tabela de símbolos que possuem mais informações sobre os tokens [1].

Nessa segunda etapa, além da revisão do analisador léxico, foi necessário construir uma árvore sintática abstrata, que representa a estrutura sintática hierárquica do programa, e a estrutura da tabela de símbolos, que são estruturas de dados utilizados pelo compilador para obter informações sobre as construções do programa-fonte.

Para a simplificação da construção da árvore sintática abstrata, foi utilizado o conceito de árvore binária para facilitar o desenvolvimento desta etapa, onde a derivação de um símbolo terminal pode crescer para a esquerda ou direita. Nesta estrutura de dados, os nodos intermediários representam os símbolos não terminais, os nodos folhas representam os tokens presentes no código fonte, e a raiz representa o programa analisado.

Para os nodos intermediários, foram armazenados apenas as informações consideradas necessárias para a construção da árvore, mostrando apenas as derivações que chegam num símbolo terminal. Nos nodos folhas, foram armazenados as suas informações, mostrando o valor atribuído aquele símbolo terminal.

No projeto, foi utilizado hash para a construção das Tabelas de Símbolos. Dentro dessa hash, foi armazenada o símbolo terminal, com o seu valor, sua tipagem e seu escopo. O intuito desse armazenamento é garantir com que seja

possível fazer a análise semântica das variáveis presentes no código, passando as informações de declarações para o uso do analisador semântico.

5 Descrição da Análise Semântica

A análise semântica é a terceira fase de um projeto de compilador. O analisador semântico é responsável por coletar as informações dos tokens e colocar em tabela de símbolos, performar verificação de tipo, verificação de etiqueta e controle de fluxo. O analisador semântico produz algum tipo de representação do programa, seja o código do objeto ou uma representação intermediária do programa [4] [1].

Nessa terceira etapa, além da revisão do analisador sintático, foi necessário otimizar a tabela de símbolos, para fazer a verificação de tipo, de etiqueta e um controle melhor do fluxo. Além disso, o programa foi otimizado para um compilador de duas etapas. Para isso, o código foi construído para construção da árvore abstrata apenas ao fim das passagens de leitura do código.

Para garantir o pleno funcionamento da análise semântica, é necessário verificar se os seguintes pontos estão presentes no trabalho: a conversão implícita de tipos, a verificação da chamada da função ou variável após a sua declaração, as regras de escopo, as regras para passagem de parâmetros e a existência de uma função main.

A conversão implícita de tipos na linguagem desenvolvida segue os mesmos parâmetros do C, onde em expressões envolvendo operadores com operandos de tipos diferentes, os valores dos operandos são avaliados e convertidos para o mesmo tipo antes da operação ser executa. Nesse caso, os tipos mais simples, como int, são convertidos para os tipos mais complexos, como o float.

No projeto, devido a uma dificuldade na etapa anterior, não foi possível realizar essa etapa por completo. Para sua finalização, falta, dentro das operações, coletar os tipos presentes na tabela de símbolo e realizar, nos operandos, a conversão de tipo, mostrando na árvore sintática abstrata anotada.

A verificação da chamada de uma função ou variável após a sua declaração tem o papel de verificar a checagem de tipos e a unicidade do identificador. Nesse caso, é observado se a função ou variável foi declarada apenas uma vez, se foi declarada ou definida antes do seu primeiro uso, se é declarada mas nunca utilizada, a que essa declarações se refere, se os tipos de uma expressão são compatíveis e se as dimensões apresentam uma consistência com o que foi declarado.

Assim como na etapa da conversão implícita de tipos, não foi possível finalizar essa parte do projeto. Para sua conclusão, falta verificar com clareza se aquelas variáveis ou funções já foram declaradas dentro do escopo de cada tabela de símbolo. Caso não tenham sido declaradas, é importante mostrar os erros semânticos para o usuário, informando sua localização.

Por se tratar de uma linguagem que faz parte de um subconjunto da linguagem C, as suas regras de escopo também respeitam as regras de escopo do C, sendo um escopo estático. Na execução do projeto, é necessário verificar se as variáveis são locais ou globais, e como elas se comportam em cada ambiente.

Caso uma variável seja declarada globalmente e localmente, dentro de funções as variáveis locais irão sobrepor o valor da variável global.

A função main é considerada o local de início da execução de um programa C. ela é única no código fonte e difere dos outros identificadores. Para isso, é necessário verificar se ela está presente no programa e se ela não é declarada mais de uma vez.

6 Gerador de Código Intermediário

Como parte final do *front end* de um compilador, o gerador de código intermediário é responsável por receber a entrada de sua fase predecessora, o analisador semântico, na forma de uma árvore de sintaxe anotada. Essa árvores sintática deve ser convertida numa representação linear.

7 Descrição dos Arquivos de Teste

Os arquivos de teste buscam apresentar códigos sem erros e códigos que possuem algum erro léxico, sintático ou semântico. Os códigos inválidos buscam mostrar erros observados nas três etapas do projeto, seja com a introdução de um caractere inexistente, uma sequência de comandos que não respeitam a gramática do projeto ou até mesmo um erro semântico no programa fonte, como declaração de mais de uma variável, passagem de parâmetros erradas ou a não inclusão de uma função main.

No código incorreto1.mf, o código apresenta erros léxicos e sintáticos. Os erros são os seguintes: um sinal de divisão fora de posição na linha 3, coluna 5, falta de ponto e vírgula na linha 12, coluna 13, apontando para uma declaração de um for na linha 13, coluna 7, uma operação de soma a mais dentro do for na linha 13, coluna 29 e um símbolo de porcentagem não definido pelo léxico na linha 14, coluna 14. Alguns erros a mais aparecem por conta dessas faltas, como uma declaração de chaves na linha 3, coluna 5, por conta da divisão fora de lugar.

No código incorreto2.mf, o código apresenta erros léxicos e sintáticos. Os erros são os seguintes: declaração de EMPTY logo após um identificador sem tipagem na linha 3, coluna 14, função forall sem a operação de conjuntos na linha 5, coluna 15, e a não inclusão de um parenteses para abrir uma função na linha 19, coluna 19.

8 Instruções para Compilação

O projeto está rodando sobre os seguintes requerimentos:

- gcc (Ubuntu 9.3.0-17ubuntu1 20.04) 9.3.0
- flex 2.6.4
- bison (GNU Bison) 3.7

Para compilar, é necessário executar os seguintes comandos:

1. $ make
2. $ ./mafralang tests/corretoX.mf, onde X representa o número do exemplo, para rodar os códigos corretos;
3. $ ./mafralang tests/incorretoX.mf, onde X representa o número do exemplo, para rodar os códigos incorretos.

Depois de compilar o código, ele gerará no próprio terminal a árvores sintática abstrata e a tabela de símbolos. Os erros apareceram no terminal, informando a linha e a coluna onde ocorreu, além da tabela de símbolos.

Referências

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., USA (2006)
2. Grune, D., Jacobs, C.J.H.: Parsing Techniques: A Practical Guide. Ellis Horwood, USA (1990)
3. Kernigham, B.W., Ritchie, D.M.: The C programming language. Prentice hall of India (1973)
4. Knuth, D.E.: Semantics of context-free languages. Mathematical systems theory **2**(2), 127–145 (1968)
5. Louden, K.C.: Compiler construction. Cengage Learning (1997)
6. Nalon, C.: Trabalho prático - descrição da linguagem (2021), https://aprender3.unb.br/mod/page/view.php?id=294131, visitou em 26-02-2021
7. Ritchie, D.M., Kernighan, B.W., Lesk, M.E.: The C programming language. Prentice Hall Englewood Cliffs (1988)

A Gramática

A gramática foi feita a partir das adaptações da gramática apresentada por Brian W. Kernighan e Dennis M. Ritchie no livro K&R [3]. Os terminais são apresentados entre aspas simples e em caixa alta, enquanto as variáveis são apresentadas entre parênteses.

⟨*program*⟩ ::= ⟨*translation_unit*⟩

⟨*translation_unit*⟩ ::= ⟨*external_declaration*⟩
| ⟨*translation_unit*⟩ ⟨*external_declaration*⟩

⟨*external_declaration*⟩ ::= ⟨*function_definition*⟩
| ⟨*declaration*⟩

⟨*function_definition*⟩ ::= ⟨*declaration_specifiers*⟩ ⟨*declarator*⟩ ⟨*declaration_list*⟩ ⟨*compound_statement*⟩
| ⟨*declaration_specifiers*⟩ ⟨*declarator*⟩ ⟨*compound_statement*⟩

⟨*declaration_list*⟩ ::= ⟨*declaration*⟩
| ⟨*declaration_list*⟩ ⟨*declaration*⟩

⟨*declaration*⟩ ::= ⟨*declaration_specifiers*⟩ ';'
| ⟨*declaration_specifiers*⟩ ⟨*init_declarator_list*⟩ ';'

⟨*declaration_specifiers*⟩ ::= 'TYPE'
| 'TYPE' ⟨*declaration_specifiers*⟩

⟨*init_declarator_list*⟩ ::= ⟨*declarator*⟩
| ⟨*init_declarator_list*⟩ ',' ⟨*declarartor*⟩

⟨*declarator*⟩ ::= ⟨*direct_declarator*⟩

⟨*direct_declarator*⟩ ::= 'ID'
| 'MAIN'
| '(' ⟨*declarator*⟩ ')'
| ⟨*direct_declarator*⟩ '(' ')'
| ⟨*direct_declarator*⟩ '(' ⟨*identifier_list*⟩ ')'
| ⟨*direct_declarator*⟩ '(' ⟨*parameter_list*⟩ ')'

⟨*identifier_ list*⟩ ::= 'ID'
| ⟨*identifier_ list*⟩ ',' 'ID'

⟨*parameter_ list*⟩ ::= ⟨*parameter_ declaration*⟩
| ⟨*parameter_ list*⟩ ',' ⟨*parameter_ declaration*⟩

⟨*parameter_ declaration*⟩ ::= ⟨*declaration_ specifiers*⟩ ⟨*declarator*⟩
| ⟨*declaration_ specifiers*⟩ ⟨*abstract_ declarator*⟩

⟨*abstract_ declarator*⟩ ::= ⟨*direct_ abstract_ declarator*⟩

⟨*direct_ abstract_ declarator*⟩ ::= '(' ⟨*abstract_ declarator*⟩ ')'
| '(' ')'
| ⟨*direct_ abstract_ declarator*⟩ '(' ')'
| ⟨*direct_ abstract_ declarator*⟩ '(' ⟨*parameter_ list*⟩ ')'

⟨*coumpound_ statement*⟩ ::= '{' '}'
| '{' ⟨*block_ item_ list*⟩ '}'

⟨*block_ item_ list*⟩ ::= ⟨*block_ item*⟩
| ⟨*block_ item_ list*⟩ ⟨*block_ item*⟩

⟨*block_ item*⟩ ::= ⟨*declaration*⟩
| ⟨*statement*⟩

⟨*statement*⟩ ::= ⟨*expression_ statement*⟩
| ⟨*compound_ statement*⟩
| ⟨*conditional_ statement*⟩
| ⟨*iteration_ statement*⟩
| ⟨*input_ statement*⟩
| ⟨*output_ statement*⟩
| ⟨*return_ statement*⟩

⟨*expression_ statement*⟩ ::= ';'
| ⟨*expression*⟩ ';'

⟨*conditional_ statement*⟩ ::= 'IF' '(' ⟨*expression*⟩ ')' ⟨*statement*⟩
| 'IF' '(' ⟨*expression*⟩ ')' ⟨*statement*⟩ 'else' ⟨*statement*⟩

⟨*iteration_ statement*⟩ ::= 'FOR' '('⟨*expression_ statement*⟩ ⟨*expression_ statement*⟩ ⟨*expression*⟩')' ⟨*statement*⟩
| 'FOR' '(' ⟨*expression_ statement*⟩ ⟨*expression_ statement*⟩')' ⟨*statement*⟩
| 'FOR' '('⟨*declaration*⟩ ⟨*expression_ statement*⟩ ⟨*expression*⟩')' ⟨*statement*⟩
| 'FOR' '('⟨*declaration*⟩ ⟨*expression_ statement*⟩')' ⟨*statement*⟩
| 'FORALL '(' ⟨*expression*⟩ ')' ⟨*statement*⟩

⟨*input_ statement*⟩ ::= 'READ' '('⟨*expression*⟩ ')'

⟨*output_ statement*⟩ ::= 'WRITE' '(' ⟨*expression*⟩ ')'
| 'WRITELN' '(' ⟨*expression*⟩ ')'

⟨*return_ statement*⟩ ::= 'RETURN' ⟨*expression*⟩ ';'

⟨*set_ expression_ list*⟩ ::= ⟨*is_ set_ expression*⟩
| ⟨*remove_ expression*⟩
| ⟨*add_ expression*⟩
| ⟨*exists_ expression*⟩

⟨*is_ set_ statement*⟩ ::= 'IS_SET' '(' 'ID' ')'

⟨*remove_ statement*⟩ ::= 'REMOVE' '(' ⟨*expression*⟩ ')'

⟨*add_ statement*⟩ ::= 'ADD' '(' ⟨*expression*⟩ ')'

⟨*exists_ statement*⟩ ::= 'EXISTS' '(' ⟨*expression*⟩ ')'

⟨*expression*⟩ ::= ⟨*expression*⟩ ',' ⟨*assignment_ expression*⟩
| ⟨*assign_ expression*⟩

⟨*assign_ expression*⟩ ::= ⟨*unary_ expression*⟩ '=' ⟨*assignment_ expression*⟩
| ⟨*relational_ expression*⟩

⟨*relational_ expression*⟩ ::= ⟨*arithmetic_ expression*⟩
| ⟨*relational_ expression*⟩ '==' ⟨*arithmetic_ expression*⟩
| ⟨*relational_ expression*⟩ '<' ⟨*arithmetic_ expression*⟩
| ⟨*relational_ expression*⟩ '≤' ⟨*arithmetic_ expression*⟩

| ⟨*relational_ expression*⟩ '>' ⟨*arithmetic_ expression*⟩
| ⟨*relational_ expression*⟩ '≥' ⟨*arithmetic_ expression*⟩
| ⟨*relational_ expression*⟩ '!=' ⟨*arithmetic_ expression*⟩

⟨*arithmetic_ expression*⟩ ::= ⟨*logical_ expression*⟩
| ⟨*arithmetic_ expression*⟩ '+' ⟨*logical_ expression*⟩
| ⟨*arithmetic_ expression*⟩ '-' ⟨*logical_ expression*⟩
| ⟨*arithmetic_ expression*⟩ '*' ⟨*logical_ expression*⟩
| ⟨*arithmetic_ expression*⟩ '/' ⟨*logical_ operation*⟩

⟨*logical_ expression*⟩ ::= ⟨*set_ expression*⟩
| '!' ⟨*set_ expression*⟩
| ⟨*logical_ expression*⟩ '&&' ⟨*set_ expression*⟩
| ⟨*logical_ expression*⟩ '||' ⟨*set_ expression*⟩

⟨*set_ expression*⟩ ::= ⟨*cast_ expression*⟩
| ⟨*set_ expression*⟩ 'IN' ⟨*cast_ expression*⟩

⟨*cast_ expression*⟩ ::= ⟨*unary_ expression*⟩
| '(' ⟨*type_ name*⟩ ')'

⟨*type_ name*⟩ ::= ⟨*specifier_ qualifier_ list*⟩
| ⟨*specifier_ qualifier_ list*⟩ ⟨*abstract_ declarator*⟩
| ⟨*speificer_ qualifier_ list*⟩ ⟨*declarator*⟩

⟨*specifier_ qualifier_ list*⟩ :: 'TYPE' ⟨*specifier_ qualifier_ list*⟩
| 'TYPE'

⟨*unary_ expression*⟩ ::= ⟨*postfix_ expression*⟩
| ⟨*set_ expression_ list*⟩
| ⟨*function_ expression*⟩
| '-' ⟨*cast_ expression*⟩

⟨*function_ expression*⟩ ::= 'ID' '(' ⟨*initializer_ list*⟩ ')'

⟨*postfix_ expression*⟩ ::= ⟨*primary_ expression*⟩
| '(' ⟨*type_ name*⟩ ')' '{' ⟨*initializer_ list*⟩ '}'
| '(' ⟨*type_ name*⟩ ')' '{' ⟨*initializer_ list*⟩ ',' '}'

⟨*initializer_list*⟩ ::= ⟨*initializer*⟩
| ⟨*initializer_list*⟩ ',' ⟨*initializer*⟩

⟨*initializer*⟩ ::= ⟨*assignment_expression*⟩
| '{' ⟨*initializer_list*⟩ '}'
| '{' ⟨*initializer_list*⟩ ',' '}'

⟨*primary_expression*⟩ ::= 'ID'
| 'INTEGER'
| 'FLOAT'
| 'STRING'
| 'EMPTY'
| '(' ⟨*expression*⟩ ')'