

Analizador léxico¹

Alexandre Yuji Kajihara¹

Universidade Tecnológica Federal do Paraná – UTFPR

DACOM – Departamento Acadêmico de Computação do Curso de Bacharelado em Ciência da Computação
Campo Mourão, Paraná, Brasil

¹alexandre.ykz@gmail.com

Resumo

Para desenvolver um compilador é necessário várias etapas, e uma delas é o analisador léxico, que tem como objetivo identificar as peculiaridades de uma determinada linguagem de programação. O método utilizado foi definir uma linguagem de programação que no nosso caso foi o T++, para que possamos montar o nosso analisador léxico. Depois de definido a linguagem verificamos como se representa um identificador, um comentário, um espaço na linguagem escolhida. Também averiguamos quais são palavras reservadas, símbolos, tipos que a linguagem possui. Após isso, definimos a linguagem de programação para desenvolver o analisador que foi a linguagem C, a ferramenta que irá nos auxiliar que foi o Flex e por fim, o JFLAP que foi utilizado para montarmos o nosso autômato. O resultado que tivemos foi que conseguimos identificar os identificadores, palavras reservadas, tipos, comentários, símbolos, espaços e valores do código-fonte. A conclusão é que essa primeira etapa para se construir um compilador é bem simples, já que apenas temos que identificar o que é cada caractere, palavra, símbolo ou valor do código-fonte. O relatório está dividido em oito partes que são: introdução, objetivo, fundamentação, materiais, procedimentos e resultados, discussão dos resultados, conclusões e referências.

1. Introdução

Para montar o analisador léxico foi necessário saber como é composta a linguagem T++, como quais são as palavras reservadas, símbolos, tipos, valores, como descrever os identificadores, comentários, espaço. Além disso, precisamos lembrar alguns conceitos como de autômatos finitos, expressões regulares e algoritmos. Com esses conhecimentos representamos as expressões regulares com base nas características

que a linguagem foi definida, representamos o autômato com todas as peculiaridades da linguagem, e com o algoritmo conseguimos identificar a composição da linguagem.

2. Objetivo

O objetivo de desenvolver o analisador léxico é conseguir identificar o que representa certas palavras ou símbolos dos código-fonte de um programa feito em T++. Por exemplo, quando encontramos a palavra repita no código-fonte devemos imprimir na tela que repita é uma palavra reservada da linguagem. Identificando o que cada palavra ou símbolo representa será útil para próxima etapa para se desenvolver um compilador que é o analisador sintático.

3. Fundamentação

Os fundamentos necessários para implementar esse o analisador léxico foi expressões regulares, autômato finito e algoritmos. A expressão regular auxiliou na representação as peculiaridades da linguagem, como identificadores na linguagem T++ tem que obrigatoriamente começar com uma letra, e podem ter n letras ou n dígitos. Com os autômatos conseguimos representar todas as características da linguagem em um único autômato. Em relação, com o algoritmo precisamos ter o conhecimento de linguagem de programação que foi escolhida, que no nosso caso foi a linguagem C, em que só usamos seus recursos básico, como um laço de repetição e a função de imprimir na tela. Além da linguagem C, tivemos que aprender a utilizar o LEX que é uma ferramenta que auxilia a gerar o analisador léxico, porém a estrutura de um arquivo LEX é algo bem simples.

4. Materiais

Os materiais utilizados foram um *laptop* com o Sistema Operacional Ubuntu 16.10 64 bits com a configuração Intel Core i7-6500U, 16 GB de memória RAM (*Random Access Memory*), um compilador para linguagem C que foi o GCC na versão 6.2.0, o LEX na

1 Trabalho desenvolvido para a disciplina de BCC36B – Compiladores

versão 2.6.1, gedit na versão 3.22.0, alguns códigos em T++ e o JFLAP versão 7.0 para montar o autômato. Com o gedit que é um editor de texto escrevemos alguns códigos na linguagem T++ e o arquivo scanner.l que é o que é aceito pelo LEX. Com o GCC e LEX, criamos nosso analisador léxico que aceita linguagem T++.

5. Procedimentos e resultados

Para começar o analisador léxico descobrimos quais seriam as peculiaridades da linguagem T++. Em que a mesma só aceita valores naturais, inteiros, flutuantes, exponenciais, arranjos unidimensionais e bidimensionais. Além disso, em uma função quando não é declarado nenhum tipo, podemos assumir que é uma função do tipo void, ou seja, não retorna nada. Ainda sobre as características da linguagem temos que as variáveis são locais ou globais, temos que os comentários começam com chaves e terminam com chaves, que os identificadores começam obrigatoriamente com uma letra e podem ter n letras ou n dígitos, e que temos quatro tipos de espaço, que é o espaço em branco, \n, \r e o \t. As duas últimas características da linguagem, seria as palavras reservada que seriam: se, então, senão, fim, repita, flutuante, retorna, até, leia, escreve, inteiro e os símbolos presentes na linguagem T++ que são: adição, subtração, multiplicação, divisão, igual, vírgula, atribuição, maior, maior igual, menor, menor igual, diferente, abre parênteses, fecha parênteses, dois pontos, abre colchetes, fecha colchetes, abre chaves e fecha chaves.

Após sabermos tudo isso à respeito da linguagem T++ escrevemos três programas, que são teste1.tpp que é um programa que resolve fatorial, teste2.tpp que também é um programa que resolve fatorial, mas de uma maneira diferente do teste1.tpp e o teste3.tpp que é um selection sort, todos seguem as características da linguagem.

Após todas essas etapas, criamos o arquivo scanner.l que é o que é compatível com LEX. Esse arquivo ele tem algumas particularidades que devem ser respeitadas, mas resumindo ele combina algumas características deles para reconhecer as expressões regulares e as características da linguagem C. Então, definimos as expressões regulares que irão identificar cada uma dessas características da linguagem T++ e imprimimos na tela o caractere ou a palavra e classificamos ele como podendo ser um símbolo, identificador, palavra reservada, inteiro, flutuante, comentário, etc através da função print da linguagem

C. Abaixo podemos ver na Tabela 1 as expressões regulares que criamos.

Tabela 1: expressões regulares da linguagem T++.

	Expressões regulares
Dígito	[0-9]
Letras	[a-zA-ZãäâÄÅÀÕóÓéÉííúÚ_]
Natural	{Dígito}+
Inteiro	{Natural}
Flutuante	{Inteiro}("."{Natural})
Exponencial	(({Inteiro}) {Flutuante})("e" "E") {Inteiro}
Reservada	("se" "então" "senão" "fim" "repita" "flu tuante" "retorna" "até" "leia" "escreve" "inteiro")
Símbolo	("+" "-" "*" "/" "=" " "<" ">" "<=" ">=" "<>" "(")" "["]" {""}")
Identificador	{Letras}({Letras} {Dígito})*
Comentário	("{"")({Letras}* {Dígito}* {" {Símbolo}* ("." !" "@ "#" "\$" "%"})*)*{"")
Espaço	[\n\r\t]+

Algumas informações que vale a pena ressaltar que escrevemos “Dígito” na expressão regular para evitar a repetição de escrever [0-9]. Além disso, o caractere ‘|’ representa uma opção, por exemplo em símbolo pode ser + ou – ou os outros caracteres. Outros dois caracteres que vale a pena ressaltar é o ‘+’ e ‘*’ em que o ‘+’ significa que deve se repetir pelo menos uma vez ou mais, já o ‘*’ ou fecho de Kleen significa que podemos repetir zero vezes ou mais. Um detalhe que irá nos auxiliar na próxima etapa que é o analisador sintático é colocar valores para cada símbolo, palavra reservada, comentário, etc, que está representado em um enum (que seria um conjunto de valores inteiros representados por identificadores), logo no começo do arquivo scanner.l. O último detalhe presente no arquivo scanner.l é que a função yylex é ela que faz a classificação do conteúdo do código-fonte.

Ciente de todas essas informações, podemos compilar os nossos arquivos, que envolve algumas etapas. Primeiramente, compilamos o arquivo LEX com o seguinte comando “lex scanner.l” em que ele resultará em um arquivo chamado lex.yy.c que será compilado pelo GCC, com o seguinte comando “gcc lex.yy.c -o scanner”, após isso gera se um executável chamado scanner. Agora para executarmos é só dar o seguinte comando “./scanner nome_do_arquivo_teste”. Para evitar de fazer todo esse processo quando mexemos no scanner.l fizemos um Makefile, a fim de poupar tempo. Na Figura 1, podemos ver uma parte do resultado em que conseguimos identificar um identificador, símbolo, palavra reservada.

Com essa etapa pronta montamos o nosso autômato no JFLAP usando as expressões regulares, com algumas modificações. Por exemplo, para identificar um identificador que deve ter começar obrigatoriamente com uma letra, fizemos que só vai para o estado de aceitação se tiver essa letra. Isso foi feito para substituir o símbolo '+', que garante que tenha pelo menos um ou mais letras. Abaixo podemos ver o autômato com as peculiaridades da linguagem T++, como podemos ver na Figura 2.

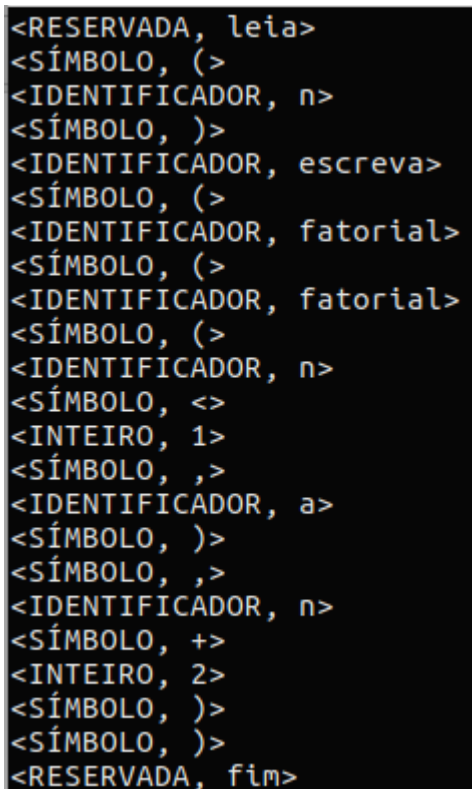


Figura 1: resultado parcial do analisador léxico usando o teste1.tpp.

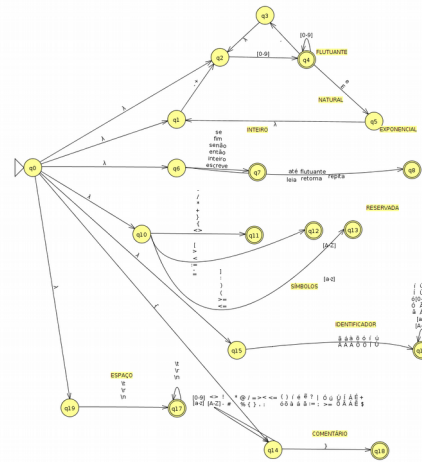


Figura 2: autômato da linguagem T^{++} .

O resultado é que em todos os nossos três testes da linguagem T++, conseguimos fazer com que o nosso analisador léxico conseguisse identificar e classificar tudo que estava presente no código-fonte.

6. Discussão dos resultados

Após os resultados do nosso programa vimos que conseguimos realizar o analisador léxico com sucesso, e que em nossos três testes conseguimos identificar e classificar todo o código-fonte. Podemos afirmar, que essa primeira etapa de identificar cada caractere, símbolo e classificar eles é uma tarefa bem simples. Além disso, tratamos de alguns casos especiais, como o de comentário no qual podemos ter comentário dentro de outro comentário, o de exponencial em que pode ter um “e” ou “E”, para representar a exponenciação. Caso esteja presente um comentário dentro de outro comentário ou uma exponenciação com um desses dois caracteres, o nosso analisador léxico conseguirá classificar cada um deles.

7. Conclusão

Concluimos, que essa primeira etapa para se construir um compilador é muito importante e fundamental. Isso porque, é o analisador léxico que vai colocar os *tokens* de tipos, símbolos, identificadores, valores, comentários, espaços em branco, etc, em cada palavra ou caractere do código-fonte. Além disso, conseguimos perceber que esse passo para desenvolver

um compilador é importante pelo fato que classificar de maneira errada nessa fase pode acarretar em danos nas próximas etapas. Isso porque se passarmos valores errados para o analisador sintático, com certeza ele gerará uma árvore errada.

8. Referências

[1] LOUDEN, K.C. Compiladores – Princípios e Práticas. Ed. Thomson Pioneira, 2004.z

Analizador Sintático

Alexandre Yuji Kajihara¹

¹DACOM -Departamento Acadêmico de Computação do Curso de Bacharelado em
Ciência da Computação – Universidade Tecnológica Federal do Paraná (UTFPR)
Campo Mourão – PR – Brazil
alexandre.ykz@gmail.com

Resumo. Para desenvolver um compilador é necessário várias etapas, e uma delas é o analisador sintático. Para se ter o analisador sintático o único requisito é ter a análise léxica, já que os tokens que são retornados na etapa anterior são utilizados nesse passo, em que cada token retornado verificamos se existe a compatibilidade com a estrutura aceita pela linguagem T++. Na análise sintática, definimos a notação BNF (Backus-Naur Form ou Backus Normal Form) e suas ações, em que isso é necessário verificar se o arquivo de entrada é compatível com o formato do código-fonte aceito pela linguagem T++. Com a BNF definida, começamos a desenvolver a análise sintática em que utilizamos o GNU Bison para essa etapa, e o Flex em que foram necessárias algumas mudanças na parte da análise léxica para obter sucesso nessa etapa, alguns conhecimentos da linguagem de programação C, para que se possa representar a árvore sintática, e conhecimento no formato de arquivo aceito pelo Graphviz dot para que possamos representar a nossa árvore em forma de imagem. Por fim, conseguimos montar a nossa árvore sintática e o resultado é que testamos em oito arquivo de entradas e conseguimos obter sucesso em todos eles. O relatório está dividido em oito partes que são: introdução, objetivo, fundamentação, materiais, procedimentos e resultados, discussão dos resultados, conclusões e referências.

1. Introdução

Para montar o analisador sintático foi necessário saber a estrutura aceita pela linguagem T++. Nessa etapa, por exemplo garante que só podemos fazer três operações fora de funções, que seria definir variáveis, atribuir valores e ter funções. Com todos esses conhecimentos iremos conseguir construir a árvore sintática.

2. Objetivo

O objetivo de desenvolver o analisador sintático é conseguir montar a árvore sintática, verificando se o conteúdo do código-fonte de entrada é aceito pelas notações BNF (Backus-Naur Form ou Backus Normal Form) da linguagem T++.

3. Fundamentação

Os fundamentos necessários para implementar o analisador léxico foi ter a BNF (*Backus-Naur Form* ou *Backus Normal Form*) para verificar se o conteúdo do código-fonte de entrada está adequado com a estrutura de um código-fonte da linguagem T++. Para validar isso utilizamos o GNU Bison, que seria um compilador de compilador, compatível com o YACC e trabalha em conjunto com o Flex. O Bison aceita arquivos no formato .Y, então criamos um arquivo chamado parser.y na qual o conteúdo dele é os tokens que foram retornados na etapa anterior mais as notações da BNF. Além disso, foi necessário relembrar alguns conceitos do Flex e da linguagem de programação C. No Flex foram necessárias algumas mudanças na análise léxica para se adequar a análise sintática, e a linguagem de programação C nos auxiliou a montar a árvore sintática. Vale a pena ressaltar, que tanto no uso do GNU Bison, quanto no uso do Flex, ambos são de fáceis manipulações. Uma ferramenta que nos auxiliou a visualizar a árvore sintática foi a Graphviz dot, em que representamos a nossa árvore em forma de imagem.

4. Materiais

Os materiais utilizados foram um *laptop* com o Sistema Operacional Ubuntu 16.10 64 bits com a configuração Intel Core i7-6500U, 16 GB de memória RAM (*Random Access Memory*), um compilador para linguagem C que foi o GCC na versão 6.2.0, o LEX na versão 2.6.1, gedit na versão 3.22.0, alguns códigos em T++ e o GNU Bison versão 3.0.4 para a análise sintática. Com o gedit que é um editor de texto escrevemos alguns códigos na linguagem T++, o scanner.l que é o que é aceito pelo LEX e o parser.y que é arquivo de formato compatível com o GNU Bison. Com o GCC, LEX e GNU Bison, criamos nosso analisador sintático que aceita a linguagem T++. Além disso, para facilitar a visualização da árvore sintática utilizamos a ferramenta Graphviz dot na versão 2.38.0.

5. Procedimentos e resultados

Para começar o analisador sintático foi definido qual é a estrutura da nossa linguagem T++, através de notações de BNF (*Backus-Naur Form* ou *Backus Normal Form*). Em que podemos ter fora de funções declarações de variáveis, e atribuição de valores e declarações de variáveis que seriam consideradas variáveis globais. Já em declarações de funções podemos ter funções com três tipos de retorno, que seria o ponto flutuante, inteiro e *void* (ou seja, não retorna nada), uma função deve ter um nome, pode ou não ter uma lista de parâmetros, e deve se ter um “corpo” nessa função. Esse “corpo” seria composto de declarações de variáveis, atribuições, expressões de condição, laços de repetição, chamada de funções (que podem ser usadas funções de leitura, escrita, ou as definidas pelo usuário), podemos ter três tipos de valores que seriam um valor de ponto flutuante (números decimais), números inteiros (números sem decimais) e números exponenciais. Todas essas características que um código-fonte da linguagem T++, foram definidas através de notações BNF, como podemos ver uma parte dela na Figura 1.

```

389 fator:
390     ABREPARENTESES expressao FECHAPARENTESES { $$ = criaNo("fator", 1, $2);
391     | var { $$ = criaNo("fator", 1, $1); }
392     | chamada_funcao { $$ = criaNo("fator", 1, $1); }
393     | numero { $$ = criaNo("fator", 1, $1); }
394     ;
395
396 numero:
397     NUMEROINTEIRO
398     {
399         auxiliar = criaNo($1, 0);
400         $$ = criaNo("numero", 1, auxiliar);
401     }
402     | NUMEROFLUTUANTE
403     {
404         auxiliar = criaNo($1, 0);
405         $$ = criaNo("numero", 1, auxiliar);
406     }
407     | EXPONENCIAL
408     {
409         auxiliar = criaNo($1, 0);
410         $$ = criaNo("numero", 1, auxiliar);
411     }
412     ;
413

```

Figura 1: notações BNF e ações para cada uma das notações.

Com as notações BNF definidas, colocamos ações em cada uma dessas notações, que podem ser visualizadas entre colchetes. Em cada uma dessas notações criamos um nó que no final, irá se juntar para formar a árvore sintática. Para visualizar a nossa árvore pode ser feita de duas maneiras, a primeira olhando para o terminal ou pela janela que irá abrir. No terminal iremos imprimir primeiro o pai e depois disso o nó filho ou os nós filhos, e quando tivermos a saída impressa de verde representa que aquele nó é um nó folha, isso pode ser visto na Figura 2.

```

xandao@danoninho: ~/CompiladorEmC
ÁRVORE SINTÁTICA
programa
lista_declaracoes

lista_declaracoes
declaracao_funcao

declaracao_funcao
inteiro cabecalho

inteiro

cabecalho
fatorial lista_parametros corpo

fatorial

```

Figura 2: árvore sintática sendo exibida no terminal.

Como relatamos anteriormente, que após o término da execução do programa irá também se abrir um janela, essa janela irá mostrar uma imagem de como ficou a nossa árvore sintática, isso foi feita com a ferramenta Graphviz dot. Para realizar isso percorremos todos os nós da árvore, e colocamos os seus pais e filhos da árvore em um arquivo .dot, enfim a nossa árvore pode ser visualizada na Figura 3.

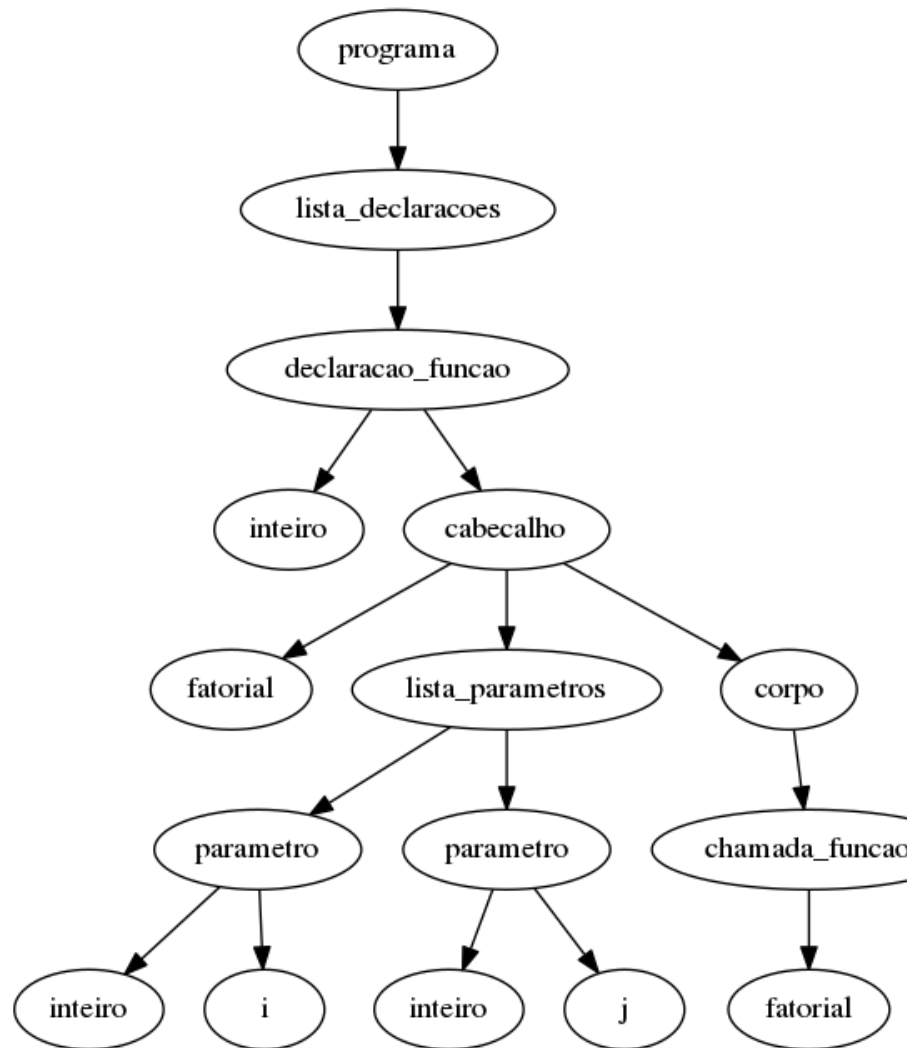


Figura 3: árvore sintática gerada pela ferramenta Graphviz dot.

Após tudo isso, realizamos teste em oito arquivos de entrada .tpp, em que cada um deles primeiro montamos essas árvores em papéis de caderno e depois, comparamos cada um deles com a saída do terminal e com a saída gerada pela ferramenta Graphviz dot, e vale a pena ressaltar que em todos eles conseguimos sucesso.

6. Discussão dos resultados

Após realizar todas esses procedimentos, para a construção dessa segunda etapa de um compilador, vimos que alcançamos o resultado desejado com todos nossos arquivos .tpp, que era a construção de árvore sintática. Vale a pena ressaltar, que quando há um erro de sintaxe no arquivo de entrada, é avisado e não se constrói a árvore sintática. Além disso, geramos um arquivo de LOG, isso caso ele caia em algum caso que possa acarretar em erros na geração da árvore sintática, porém se nenhum erro acontecer ele não irá gerar e nem avisar que foi gerado, porém se gerar emitirá um aviso avisando que foi criado um arquivo de LOG.

7. Conclusão

Concluimos, que essa segunda etapa para se construir um compilador é muito relevante e essencial. Isso porque, é o analisador sintático que vai verificar se o código-fonte de entrada respeita o formato em que a nossa linguagem T++ aceita. Conseguir a árvore sintática é muito importante nesse passo, já que para etapa que vem que é a análise semântica, consiga por exemplo, verificar se aquela variável presente no “corpo” de uma função ela foi declarada, se a função é do tipo void, então não podemos ter uma expressão de retorno. Tivemos alguns problemas logo no começo, de que seria como era estruturado um arquivo .Y, onde teria que definir as notações BNF (*Backus-Naur Form* ou *Backus Normal Form*), porém vendo alguns exemplos conseguimos resolver esse problema. Outro problema que apareceu durante as declarações dos BNF, em que tivemos alguns *warnings* que foram gerados, em que não sabíamos as causas, porém no final conseguimos resolver, em que seria que algumas regras faltavam. Com relação a árvore sintática, refizemos várias vezes até chegar nessa árvore, mas enfim conseguimos gerar a nossa árvore sintática sem problema nenhum. Entretanto, todas essas dificuldades citadas acima foram superadas e consequentemente conseguimos alcançar nosso objetivo.

References

Louden, K.C. Compiladores – Princípios e Práticas. Ed. Thomson Pioneira, 2004.

Análise semântica

Alexandre Yuji Kajiahra¹

¹DACOM - Departamento Acadêmico de Computação do
Curso de Bacharelado em Ciência da Computação
Universidade Tecnológica Federal do Paraná (UTFPR)
Caixa Postal 271 – 87301-899 – Campo Mourão – PR – Brazil

alexandre.ykz@gmail.com

Resumo. *Compiladores são programas de computador que traduzem uma linguagem para outra, e são compostos por algumas etapas. A análise léxica lê o programa-fonte como um arquivo de caracteres e o separa em marcas. A análise sintática determina a sintaxe ou estrutura de um programa. A sintaxe de uma linguagem de programação é normalmente dada pelas regras gramaticais de uma gramática livre de contexto (GLC). A análise semântica, é uma etapa que requer a computação de informações que estão além da capacidade de gramáticas livres de contexto e dos algoritmos padrão de análise sintática. A geração do código intermediário, seria uma maneira de gerar uma nova representação intermediária, a partir da árvore sintática, que se assemelhe melhor ao código-alvo ou substitua a árvore sintática por essa representação intermediária, e em seguida gerar o código-alvo a partir dessa nossa representação. Após termos realizados a análise léxica e sintática com sucesso para a linguagem de programação T++, iremos apresentar a forma que foi empregada para resolver a etapa atual, que no caso é a análise semântica em que utilizamos a árvore sintática e as regras gramaticais, gerada na etapa anterior, para verificar se a árvore foi gerada de maneira correta ou não. Para tal façanha, fizemos algumas modificações no scanner da análise léxica, nas regras gramáticas da análise sintática e percorremos a árvore verificando se a mesma estava de acordo com as regras gramaticais da linguagem de programação T++. Enfim, depois de várias tentativas de realizar a análise semântica, obtivemos sucessos em grande parte dos erros que podem surgir como entrada do nosso futuro compilador.*

1. Introdução

Compiladores são programas de um computador que traduzem de uma linguagem para outra. Um compilador é um programa bastante complexo, que pode ter de 10.000 a 1.000.000 de linhas de código. Um compilador recebe como entrada um programa escrito na linguagem-fonte e produz um programa equivalente na linguagem alvo. [Louden 2004]. A seguir, iremos mostrar as etapas que são compostas um compilador.

A análise léxica, é a fase de um compilador que lê o programa-fonte como um arquivo de caracteres e o separa em marcas, essas são como palavras em um linguagem natural: cada marca é uma sequência de caracteres que representa uma unidade de informação do programa-fonte [Louden 2004].

A análise sintática determina a sintaxe, ou estrutura, de um programa. A sintaxe de uma linguagem de programação é normalmente dada pelas regras gramaticais de uma

gramática livre de contexto, de maneira similar à forma como a estrutura léxica das marcas reconhecidas pelo sistema de varredura é dada por expressões regulares [Louden 2004].

A análise semântica recebe esse nome, pois requer a computação de informações que estão além da capacidade das gramáticas livre de contexto e dos algoritmos padrão de análise sintática [Louden 2004].

Uma representação intermediária é uma estrutura de dados que represente o programa-fonte durante a tradução. Portanto o programador de um compilador pode preferir gerar uma nova forma de representação intermediária, a partir da árvore sintática, que se assemelhe melhor ao código-alvo ou substitua a árvore sintática por essa representação intermediária, e em seguida gerar um código-alvo a partir dessa nova representação. Essa representação intermediária semelhante ao código-alvo é denominada código intermediário [Louden 2004].

A linguagem de programação que foi proposta para que realizássemos um compilador seria a linguagem T++. Assim, como qualquer linguagem ela possui algumas peculiaridades, em que: existem dois tipos de arranjos unidimensionais e bidimensionais, dois tipos de valores de variáveis inteiros e flutuantes, o laço de repetição com a condição repita até, três tipos de retorno de função, sendo eles inteiro, flutuante e *void*, ou seja, retorna nada, etc.

Tendo esses conceitos bem definidas para se produzir um compilador, e as peculiaridades da linguagem T++, iremos apresentar umas das etapas citadas. Com a análise léxica e sintática prontas, iremos apresentar a análise semântica, em que além das regras gramáticas que compõe a linguagem T++, utilizamos regras gramáticas falsas para emitir alertas, modificações na análise léxica para que a mesma pudesse retorna a linha em que estava sendo "varrida", a utilização da tabela de símbolos e regras que percorrem a árvore sintática verificando a existência ou não de anomalias. Após refazer inúmeras vezes, conseguimos tratar grande parte dos casos, além do que, alguns imprevistos também não permitiram de tratar todos os casos.

O trabalho aqui apresentado está organizado da seguinte forma: na seção 2 é apresentado o objetivo desse trabalho que é a realização do análise semântica; na seção 3 são apresentados os fundamentos que foram necessários para realização dessa da análise semântica; na seção 4 é descrito os materiais que são necessários para fazer realizar; na seção 5 os procedimentos que foram feitos para alcançar parte do objetivo; na seção 6 os resultados que foram obtidos a partir na análise semântica; na seção 7 a conclusão após realizarmos mais uma etapa do compilador; por fim, as referências que utilizamos.

2. Objetivo

O nosso objetivo é a implementação de uma das fases de um compilador, e atualmente estamos na análise semântica. Nessa etapa, iremos reportar todos os erros que a linguagem de programação T++ com base nas peculiaridades dela. Esses erros são erros que as regras gramaticais não consegue detectar, para isso iremos utilizar regras gramaticais que ocasionam os erros, e além disso, regras que percorrem a árvore sintática, a tabela de símbolos para verificar a presença de algum detalhe não permitido. Um exemplo de erro que é detectado nessa fase e que não se consegue detectar na fase anterior é índices de arranjos unidimensionais e bidimensionais terem valores diferentes de inteiros, conversão do tipo inteiro para flutuante e vice-versa, etc.

3. Fundamentação

Os fundamentos necessários para realizar mais uma etapa de um compilador para a linguagem de programação T++, foram vários o primeiro seria a manipulação de uma linguagem de programação, que no nosso caso foi a linguagem de programação C, em que utilizamos o básico: laços de repetição, funções de impressão na tela, e também desfrutamos de estruturas de dados, manipulação de arquivos, ponteiros, para conseguir detectar esses erros.

Como dito nas seções anteriores foram necessários algumas modificações nas etapas anteriores, para que se alcançasse o objetivo. Com as modificações feitas tivemos que recompilar a análise léxica, em que utilizamos o seguinte comando:

```
flex -d -T scanner.l
```

Em que *flex* seria a ferramenta que foi utilizada para nos retornar as marcar presentes no código-fonte do programa, e *scanner.l* seria o nosso arquivo das características das palavras aceitas pela linguagem de programação T++. Na fase da análise sintática, devido a algumas alterações também foi necessário a recompilação, que foi feita a partir do comando:

```
bison -y -d -t $^ -o parser.tab.c
```

O *bison* seria a ferramenta que irá verificar as marcas geradas na primeira etapa e verificar se existe a combinação com as regras gramaticais impostas. Então, unimos essas duas etapas, utilizando o compilador da linguagem de programação C, através do seguinte comando:

```
gcc -c parser.tab.c lex.yy.c  
gcc parser.tab.o lex.yy.o syntaxtree.c  
semantic.c -o analiseSemantica
```

O primeiro comando basicamente se gera '.o' dos arquivos gerados nas etapas anteriores, que são utilizados no comando seguinte para novamente unir com mais dois arquivos, em que os arquivos: *syntaxtree.c*, que é a estrutura de árvore sintática, que foi criada na etapa anterior e *semantic.c* que é uma biblioteca criada nessa etapa, justamente para prevenir aqueles erros que não foram detectados nas regras gramaticais. Devido ao fato, que é necessário recompilar várias vezes, até alcançar o objetivo desejado criamos um arquivo *Makefile* com esses comandos, evitando de ter que digitar todos os comandos citados acima, em que apenas o comando abaixo, o mesmo irá realizar os comandos citados anteriormente.

```
make all
```

4. Materiais

Os materiais necessários foram um *laptop* com o sistema operacional Ubuntu 16.10 64 bits com a configuração Intel Core i7-6500U, 16 GB de memória RAM (*Random Access Memory*), o *flex* que seria o gerador de marcas na versão 2.6.1, o *bison* que verifica as regras gramaticais na versão 3.0.4, o compilador da linguagem de programação C, que no nosso caso foi o GCC na versão 6.2.0, o *make* para evitar o processo de repetição de comandos na versão 4.1 e os editores de texto gedit e Sublime Text, na versão 3.22.0 e 3126, respectivamente.

5. Procedimentos e resultados

Como dito em seções anteriores, com a análise léxica e a sintática prontas tivemos que realizar modificações nelas, para que se pudesse alcançar o nosso objetivo que seria um analisador semântico. Com relação a análise léxica, tivemos que adicionar mais uma marca, que no caso seria uma marca de quebra de linha. Isso foi utilizado, para que pudessemos avisar a pessoa que estará utilizando a ferramenta, que ela saiba, em qual linha aproximadamente o erro está presente no código-fonte, em que pode ser visualiza na figura 1.

```
247
248 "\n" {
249     transfereLinha(linha);
250     ++linha;
251 }
```

Figura 1. A nova marca adicionada no arquivo da análise léxica

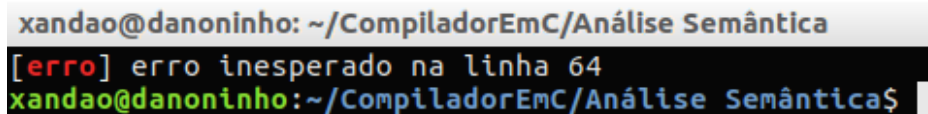
Para relatar se a árvore sintática é adequada ou não, dividimos essa verificação em duas etapas. A primeira delas envolve mudanças na regras gramaticais da análise sintática, em que foram visitadas regra por regra, em que foram criadas regras gramaticais, que ocasionam o erro. Entretanto, adicionando essas regras que acarretavam no erro, acabavam gerando conflito de regras, sendo assim não possível adicionar todas essas regras que geram erros. Abaixo podemos ver regras que foram criadas que ocasionam erros nos parâmetros das funções, em que podemos ter o identificador do parâmetro faltando, o caractere ':' faltando, um fecha colchete ou abre colchete de algum índice faltando, etc 2. Essas regras irão invocar funções que irão relatar o erro, dependendo do que foi imposto.

```
180 parametro:
181     tipo DOISPONTOS IDENTIFICADOR { $$ = criaNo("parametro", 3, $1, criaNo(":", 0), criaNo("$3", 0)); }
182     //[ ]
183     tipo DOISPONTOS IDENTIFICADOR ABRECOLCHETE FECHACOLCHETE { $$ = criaNo("parametro", 5, $1, criaNo(":", 0), criaNo("$3", 0), criaNo("[", 0),
184     criaNo("]", 0)); }
185     //[ ]
186     tipo DOISPONTOS IDENTIFICADOR ABRECOLCHETE { erroIndiceParametro = true; $$ = criaNo("parametro", 4, $1, criaNo(":", 0), criaNo("$3", 0),
187     criaNo("[", 0), criaNo("\033[1m\033[31m\033[0m", 0)); }
188     //[ ]
189     tipo DOISPONTOS IDENTIFICADOR FECHACOLCHETE { erroIndiceParametro = true; $$ = criaNo("parametro", 4, $1, criaNo(":", 0), criaNo("$3", 0),
190     criaNo("\033[1m\033[31m\033[0m", 0), criaNo("]", 0)); }
191     //[ ]
192     tipo DOISPONTOS IDENTIFICADOR ABRECOLCHETE FECHACOLCHETE { erroIndiceParametro = true; $$ = criaNo("parametro", 7, $1,
193     criaNo("$3", 0), criaNo("[", 0), criaNo("]", 0), criaNo("[", 0), criaNo("]", 0)); }
194     //[ ]
195     tipo DOISPONTOS IDENTIFICADOR FECHACOLCHETE ABRECOLCHETE FECHACOLCHETE { erroIndiceParametro = true; $$ = criaNo("parametro", 7, $1,
196     criaNo("$3", 0), criaNo("[", 0), criaNo("]", 0), criaNo("\033[1m\033[31m\033[0m", 0), criaNo("]", 0)); }
197     //[ ]
198     tipo DOISPONTOS IDENTIFICADOR FECHACOLCHETE FECHACOLCHETE { erroIndiceParametro = true; $$ = criaNo("parametro", 7, $1, criaNo(":", 0),
199     criaNo("$3", 0), criaNo("\033[1m\033[31m\033[0m", 0), criaNo("[", 0), criaNo("\033[1m\033[31m\033[0m", 0), criaNo("]", 0)); }
200     //[ ]
201     tipo DOISPONTOS IDENTIFICADOR ABRECOLCHETE ABRECOLCHETE { erroIndiceParametro = true; $$ = criaNo("parametro", 7, $1,
202     criaNo("$3", 0), criaNo("[", 0), criaNo("[", 0), criaNo("\033[1m\033[31m\033[0m", 0), criaNo("[", 0), criaNo("]", 0)); }
```

Figura 2. Regras gramaticais que ocasionam o erro no arquivo da análise sintática

Caso um arquivo de entrada não consiga a combinação de marcas com as regras gramaticais, geramos uma mensagem de erro inesperado, entretanto não conseguimos identificar em qual instante não houve essa combinação, mas em uma linha aproximada da localização do possível erro. Se o arquivo de entrada conseguir combinar as suas marcas com as regras gramaticais que ocasionam ou não o erro, geramos a árvore sintática.

Então criamos uma biblioteca, que recebe essa árvore sintática e percorre a mesma, verificando primeiramente as atribuições e as variáveis globais, ou seja, declarações e atribuições de variáveis que são feitas fora de funções, e consequentemente essas informações preenchem a nossa tabela de símbolos.

A terminal window with a dark background. The title bar at the top reads 'xandao@danoninho: ~/CompiladorEmC/Análise Semântica'. The main area shows a red prompt '[erro]' followed by the text 'erro inesperado na linha 64' in white. Below this, the prompt 'xandao@danoninho:~/CompiladorEmC/Análise Semântica\$' is shown in green, followed by a white cursor block.

```
xandao@danoninho: ~/CompiladorEmC/Análise Semântica
[erro] erro inesperado na linha 64
xandao@danoninho:~/CompiladorEmC/Análise Semântica$
```

Figura 3. Erro devido as marcas que não combinaram com as regras gramaticais

Nessa etapa em que verificamos fora das funções, detectamos os seguintes erros: índices de variáveis sendo do tipo flutuante, ou seja, posições de arranjos unidimensionais e bidimensionais sendo valores reais, redeclaração de variável, a conversão de uma variável do tipo inteiro para uma variável do tipo flutuante ou vice-versa, se o valor de uma atribuição está correto, ou seja, se é uma variável do tipo inteiro, então espera receber uma variável ou um valor que seja do mesmo tipo, variável inutilizada, ou seja, foi declarada e não foi utilizada, se uma atribuição possui alguma variável que não foi declarada, se uma variável é um arranjo bidimensional e está tendo atribuir nela como se fosse uma variável unidimensional ou vice-versa.

Após realizar essas verificações nas variáveis globais, partimos para as funções. Com as funções também repetimos alguns processos que foram utilizados nas variáveis e atribuições globais. Nessa etapa de certificamos que os índices de variáveis sendo do tipo flutuante, ou seja, posições de arranjos unidimensionais e bidimensionais sendo valores reais, redeclaração de variável, ou seja, se uma variável local, já foi declarada globalmente ou é um dos parâmetros da função, a conversão de uma variável do tipo inteiro para uma variável do tipo flutuante ou vice-versa, se o valor de uma atribuição está correto, ou seja, se é uma variável do tipo inteiro, então espera receber uma variável ou um valor que seja do mesmo tipo, variável inutilizada, ou seja, foi declarada e não foi utilizada, se uma atribuição possui alguma variável que não foi declarada, se uma variável é um arranjo bidimensional e está tendo atribuir nela como se fosse uma variável unidimensional ou vice-versa, se existem redefinições de funções, se existe uma função principal, se existem retorno ou nas funções, já que dependendo do tipo da função haverá ou não um retorno, exemplo: se uma função é do tipo *void* ela não pode retornar nada, caso uma função é do tipo inteiro ou flutuante deve se ter uma retorno, além do que verificamos no caso de retorno, se o retorno é válido, ou seja, se a função é do tipo inteiro o valor de retorno tem que ser um inteiro sendo válido para uma função do tipo flutuante. Além disso, foi

tratado as funções de leitura e escrita, já que podemos imprimir uma variável, a soma de um valor, etc, e escrever podemos escrever somente em variáveis, e também a recursão, ou seja, a chamada de uma função simulando uma laço de repetição. Abaixo podemos ver na figura 4 um exemplo de erro, em que a função é do tipo flutuante, e tentamos retornar um inteiro, e quando não se chama uma função tornando a mesma inutilizada. Vale a pena ressaltar que as informações contidas foram armazenadas na tabela de símbolos.

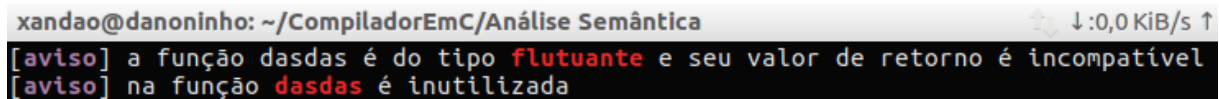
A terminal window with a dark background and light text. The title bar reads 'xandao@danoninho: ~/CompiladorEmC/Análise Semântica'. The terminal output shows two lines of messages: '[aviso] a função dasdas é do tipo flutuante e seu valor de retorno é incompatível' and '[aviso] na função dasdas é inutilizada'. The word 'flutuante' is highlighted in red in the first line, and 'dasdas' is highlighted in red in the second line. In the top right corner, there is a status bar showing a download icon, '↓:0,0 KiB/s', and an upload icon with '↑'.

Figura 4. Erro presente nas funções

O resultado que obtivemos é que conseguimos detectar todos os erros que foram citados anteriormente, em que ficaram apenas poucas funções dentro do corpo da de funções para verificar se as mesmas estavam corretas ou não.

6. Discussão dos resultados

Como dito anteriormente, alcançamos em parte nosso objetivo que era relatar os erros presentes no arquivo de entrada, com base nos detalhes presentes na linguagem de programação T++. Em parte porque os erros que foram citados na seção anterior, o nosso analisador semântico gerou erros ou avisos de qual erro que estava acontecendo, em todas possibilidades que podem acontecer. Nesses que casos que foram tratados, evidenciamos com cores as mensagem de erro visando o que está gerando esse erro. Além do que, dependendo do caso podemos ter uma flecha vermelha indicando a possível localização do erro, e também a linha aproximada da localização do erro, que pode ser visto na figura 5. Todos os erros que acontecem dentro de uma função eles são citados que estão acontecendo em uma função, para facilitar quem irá usufruir desse analisador, como pode ser visto na figura 4.

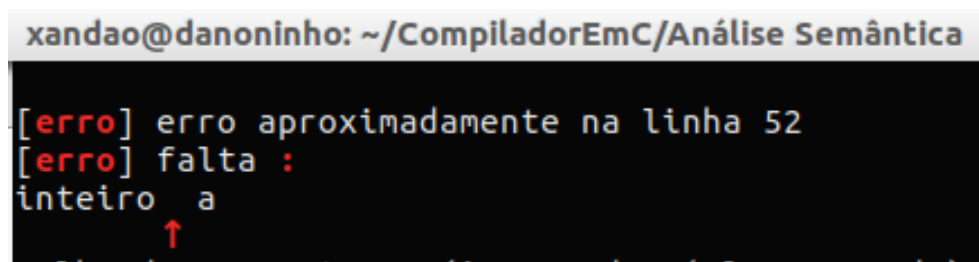
A terminal window with a dark background and light text. The title bar reads 'xandao@danoninho: ~/CompiladorEmC/Análise Semântica'. The terminal output shows three lines: '[erro] erro aproximadamente na linha 52', '[erro] falta :', and 'inteiro a'. A red arrow points upwards from the bottom of the third line towards the colon in the second line.

Figura 5. Erro com possível localização do erro

O que restou para alcançar o nosso objetivo, seria verificar também as expressões condicionais (*if-else*) e o laço de repetição, que no nosso caso é repita até. Depois de realizarmos o analisador semântico, pensamos que não seria muito difícil de realizar, já que ambos os casos não tratados possuem um corpo, e julgamos que seria o mesmo processo que foi utilizado para percorrer o corpo da função, e o que diferenciaria seria as condições presentes no *if* e na condição de repetições do laço repita até. Um outro fator que talvez poderia ter nós auxiliado é complementar a tabela de símbolos, com mais informações com bastante utilidade e consequentemente evitaria, com que realizássemos alguns processos e talvez conseguisse tratar todos os casos.

7. Conclusão

Após realizar todas essas etapas, conseguimos verificar que essa etapa é a mais importante que nós realizamos em relação as demais, pois erros aqui podem ser gravíssimos nas etapas que irão vir a partir dessa. Exemplo, verificar se uma variável foi declarada ou não, pode acarretar em que não iremos alocar um espaço para ela e consequentemente não tenha um lugar para aquele valor.

O objetivo só foi alcançado em partes devido alguns imprevistos que houveram. Como dito na seção anterior, se talvez tivéssemos pensando que os casos que restaram tinham semelhança com o que já tínhamos, poderíamos ter chegado no nosso objetivo, ou chegar próximo dele. Julgamos que talvez se tivéssemos pensando de uma forma mais genérica, ou complementado a tabela de símbolos, conseguiríamos verificar se existem erros em todos os casos.

Enfim, em relação ao nosso analisador semântico, estamos satisfeitos já que os erros que relatamos na seção 5 foram tratados com a maioria das possibilidades de realizar esses erros, porém acreditamos que existem alguns casos que irão acarretar erro, devido que existem inúmeros casos. Consideramos que identificar a localização do erro, indicando a função ou a linha, o que é que falta em uma determinada linha, a possível posição do erro nessa linha, evidenciar com um cor forte, negritada auxiliam a pessoa a ir nessa linha e arrumar ela.

Referências

Louden, K. C. (2004). *COMPILADORES: Princípios e Práticas*. Thomson, 1st edition.

Geração de código intermediário

Alexandre Yuji Kajiahra¹

¹DACOM - Departamento Acadêmico de Computação do
Curso de Bacharelado em Ciência da Computação
Universidade Tecnológica Federal do Paraná (UTFPR)
Caixa Postal 271 – 87301-899 – Campo Mourão – PR – Brazil

alexandre.ykz@gmail.com

Resumo. *Compiladores são programas de computador que traduzem uma linguagem para outra, e são compostos por algumas etapas. A análise léxica lê o programa-fonte como um arquivo de caracteres e o separa em marcas. A análise sintática determina a sintaxe ou estrutura de um programa. A sintaxe de uma linguagem de programação é normalmente dada pelas regras gramaticais de uma gramática livre de contexto (GLC). A análise semântica, é uma etapa que requer a computação de informações que estão além da capacidade de gramáticas livres de contexto e dos algoritmos padrão de análise sintática. A geração do código intermediário, seria uma maneira de gerar uma nova representação intermediária, a partir da árvore sintática, que se assemelhe melhor ao código-alvo ou substitua a árvore sintática por essa representação intermediária, e em seguida gerar o código-alvo a partir dessa nossa representação. Após termos realizados a análise léxica, sintática, semântica para a linguagem de programação T++, iremos apresentar a forma que foi empregada para resolver a etapa atual, que no caso é a geração de código intermediário em que utilizamos a árvore sintática e funções de uma biblioteca chamada LLVM (Low Level Virtual Machine), que é uma coleção de tecnologias de compiladores e ferramentas de ferramentas modulares e reutilizáveis, que será usada para gerar o nosso código intermediário. Para tal façanha, criamos uma nova biblioteca, que só recebe a árvore sintática caso não tenha gerado nenhum erro, e assim percorre a mesma invocando funções da LLVM. Enfim, depois de várias tentativas para a geração de código intermediários, obtivemos sucessos em alguns casos que podem surgir como entrada do nosso futuro compilador.*

1. Introdução

Compiladores são programas de um computador que traduzem de uma linguagem para outra. Um compilador é um programa bastante complexo, que pode ter de 10.000 a 1.000.000 de linhas de código. Um compilador recebe como entrada um programa escrito na linguagem-fonte e produz um programa equivalente na linguagem alvo. [Louden 2004]. A seguir, iremos mostrar as etapas que são compostas um compilador.

A análise léxica, é a fase de um compilador que lê o programa-fonte como um arquivo de caracteres e o separa em marcas, essas são como palavras em um linguagem natural: cada marca é uma sequência de caracteres que representa uma unidade de informação do programa-fonte [Louden 2004].

A análise sintática determina a sintaxe, ou estrutura, de um programa. A sintaxe de uma linguagem de programação é normalmente dada pelas regras gramaticais de uma

gramática livre de contexto, de maneira similar à forma como a estrutura léxica das marcas reconhecidas pelo sistema de varredura é dada por expressões regulares [Louden 2004].

A análise semântica recebe esse nome, pois requer a computação de informações que estão além da capacidade das gramáticas livre de contexto e dos algoritmos padrão de análise sintática [Louden 2004].

Uma representação intermediária é uma estrutura de dados que represente o programa-fonte durante a tradução. Portanto o programador de um compilador pode preferir gerar uma nova forma de representação intermediária, a partir da árvore sintática, que se assemelhe melhor ao código-alvo ou substitua a árvore sintática por essa representação intermediária, e em seguida gerar um código-alvo a partir dessa nova representação. Essa representação intermediária semelhante ao código-alvo é denominada código intermediário [Louden 2004]. Para a geração desse código iremos utilizar uma biblioteca chamada LLVM (*Low Level Virtual Machine*), uma coleção de tecnologias de compiladores e ferramentas de ferramentas modulares e reutilizáveis [LLVM 2017].

A linguagem de programação que foi proposta para que realizássemos um compilador seria a linguagem T++. Assim, como qualquer linguagem ela possui algumas peculiaridades, em que: existem dois tipos de arranjos unidimensionais e bidimensionais, dois tipos de valores de variáveis inteiros e flutuantes, o laço de repetição com a condição repita até, três tipos de retorno de função, sendo eles inteiro, flutuante e *void*, ou seja, retorna nada, etc.

Tendo esses conceitos bem definidas para se produzir um compilador, e as peculiaridades da linguagem T++, iremos apresentar umas das etapas citadas. Com a análise léxica, sintática e semântica prontas, iremos apresentar a geração de código intermediário, em que percorremos a árvore sintática verificando o que é cada um dos nós, para chamar determinadas funções do LLVM, que resultará num código no formato '.bc'. Após refazer inúmeras vezes, conseguimos tratar uma pequena parte dos casos, além do que, alguns imprevistos também não permitiram de tratar todos os casos.

O trabalho aqui apresentado está organizado da seguinte forma: na seção 2 é apresentado o objetivo desse trabalho que é a realização do geração de código intermediário; na seção 3 são apresentados os fundamentos que foram necessários para realização dessa da geração de código intermediário; na seção 4 é descrito os materiais que são necessários para fazer realizar; na seção 5 os procedimentos que foram feitos para alcançar parte do objetivo; na seção 6 os resultados que foram obtidos a partir na geração de código intermediário; na seção 7 a conclusão após realizarmos mais uma etapa do compilador; por fim, as referências que utilizamos.

2. Objetivo

O nosso objetivo é a implementação de uma das fases de um compilador, e atualmente estamos na geração do código intermediário. Nessa etapa, a partir de um arquivo de entrada iremos gerar um código intermediário da linguagem de programação T++, através das funções existentes nas bibliotecas do LLVM (*Low Level Virtual Machine*), que resultará em um arquivo no formato '.bc' gerado, iremos executá-lo.

3. Fundamentação

Os fundamentos necessários para realizar mais uma etapa de um compilador para a linguagem de programação T++, foram vários o primeiro seria a manipulação de uma linguagem de programação, que no nosso caso foi a linguagem de programação C, em que utilizamos o básico: laços de repetição, funções de impressão na tela, e também desfrutamos de estruturas de dados, ponteiros, para percorrer a árvore sintática e conseguir gerar o código intermediário.

As etapas anteriores resultarão em alguns arquivos, sendo algumas bibliotecas criadas nas etapas anteriores e também nessa etapa, iremos criar arquivos no formato '.o' de cada um deles, que pode ser visto no comando abaixo.

```
clang-3.5 -c generate.c syntaxtree.c semantic.c lista.c  
parser.tab.c lex.yy.c 'llvm-config-3.5 --cflags '
```

Com os arquivos '.o' gerado no comando anterior, iremos "unir" eles para que possam gerar um executável. Ambos os comandos tem algumas coisas em comuns como o compilador para a linguagem de programação C que é clang-3.5, e alguns parâmetros para que possamos desfrutar das funções que as bibliotecas do LLVM (*Low Level Virtual Machine*).

```
clang++-3.5 parser.tab.o lex.yy.o generate.o syntaxtree.o  
semantic.o lista.o -lm -ogenerate 'llvm-config-3.5  
--ldflags --libs core executionengine jit interpreter  
analysis native bitwriter --system-libs '
```

Devido ao fato, que é necessário recompilar várias vezes, até alcançar o objetivo desejado criamos um arquivo *Makefile* com esses comandos, evitando de ter que digitar todos os comandos citados acima, em que apenas o comando abaixo, o mesmo irá realizar os comandos citados anteriormente.

```
make all
```

4. Materiais

Os materiais necessários foram um *laptop* com o sistema operacional Ubuntu 16.10 64 bits com a configuração Intel Core i7-6500U, 16 GB de memória RAM (*Random Access Memory*), o compilador da linguagem de programação C, que no nosso caso foi o clang na versão 3.5, com alguns parâmetros para permitir o uso das funções que as bibliotecas do LLVM (*Low Level Virtual Machine*), o *make* para evitar o processo de repetição de comandos na versão 4.1 e os editores de texto gedit e Sublime Text, na versão 3.22.0 e 3126, respectivamente.

5. Procedimentos e resultados

Como a análise léxica, sintática e semântica prontas utilizamos arquivos que foram gerados nelas para chegar no nosso objetivo nessa etapa que seria a geração de código intermediário. Primeiramente criamos uma nova biblioteca, em que tem uma função que recebe a árvore sintática e a tabela de símbolos, se e somente se não for gerado nenhum erro nas etapas anteriores. Caso tenha algum erro nem irá se chamar essa função.

Para gerar o código intermediário primeiramente percorremos a árvore sintática com os nós que não são funções, ou seja, só nos resta as declarações de variáveis e atribuições de variáveis. Com isso analisamos se o tipo é inteiro ou flutuante, e se for um desses se é somente uma variável ou um arranjo unidimensional ou bidimensional. Após isso, verificamos se a variável tem algum valor atribuído, caso tenha já atribuímos esse valor, na qual fizemos tudo isso por meio de invocações das funções do LLVM (*Low Level Virtual Machine*), como pode ser visto na figura 1.

```

123
124 void codigoIRGlobais(LLVMContextRef context, LLVMModuleRef module, LLVMBuilderRef builder, ListaVariaveis *globais, ListaInicializacao
    *inicializacao){
125     char nomeVariavel[MAX];
126     int valor1, valor2;
127     NoVariaveis *auxiliar = globais -> primeiro;
128     while(auxiliar != NULL){
129         memset(nomeVariavel, 0, sizeof(nomeVariavel));
130         if(compareString(auxiliar -> tipo, "inteiro") == 0){
131             strcpy(nomeVariavel, auxiliar -> nome);
132             if(auxiliar -> ehUnidimensional){
133                 valor1 = atoi(auxiliar -> valor1);
134                 LLVMTypeRef type = LLVMArrayType(LLVMInt64Type(), valor1);
135                 LLVMValueRef array = LLVMAddGlobal(module, type, nomeVariavel);
136                 LLVMSetInitializer(array, LLVMConstInt(LLVMInt64Type(), 0, false));
137                 LLVMSetLinkage(array, LLVMCommonLinkage);
138                 LLVMSetAlignment(array, 16);
139             } else if(auxiliar -> ehBidimensional){
140                 valor1 = atoi(auxiliar -> valor1);
141                 valor2 = atoi(auxiliar -> valor2);
142                 LLVMTypeRef type_0 = LLVMArrayType(LLVMInt64Type(), valor1);
143                 LLVMTypeRef type = LLVMArrayType(type_0, valor2);
144                 LLVMValueRef array = LLVMAddGlobal(module, type, nomeVariavel);
145                 LLVMSetInitializer(array, LLVMConstInt(LLVMInt64Type(), 0, false));
146                 LLVMSetLinkage(array, LLVMCommonLinkage);
147                 LLVMSetAlignment(array, 16);
148             } else if((auxiliar -> ehUnidimensional == false) && (auxiliar -> ehBidimensional == false)){
149                 LLVMValueRef array = LLVMAddGlobal(module, LLVMInt64Type(), nomeVariavel);
150                 LLVMSetInitializer(array, LLVMConstInt(LLVMInt64Type(), existeAtribuicaoInteiro(auxiliar -> nome, inicializacao), false));
151                 LLVMSetLinkage(array, LLVMCommonLinkage);
152                 LLVMSetAlignment(array, 16);
153             } else if(compareString(auxiliar -> tipo, "flutuante") == 0){
154                 strcpy(nomeVariavel, auxiliar -> nome);
155                 if(auxiliar -> ehUnidimensional){
156                     valor1 = atoi(auxiliar -> valor1);
157                     LLVMTypeRef type = LLVMArrayType(LLVMFloatType(), valor1);

```

Figura 1. Código que gerar as variáveis e atribuições globais

Com essa etapa pronta, partimos para as funções, na qual percorremos as funções uma por uma e aplicamos assim como na etapa anterior as funções do LLVM, como pode ser visto na figura 2. Após isso criamos dois blocos, um sendo de entrada e outro de saída, e dentro desses blocos, declaramos as variáveis e inicializamos elas, assim como feito na etapa anterior. Feito isso também verificamos se a função presente tinha um tipo, já que caso tivesse deveria ter um valor de retorno.

```

80 void codigoIRFuncao(LLVMContextRef context, LLVMModuleRef module, LLVMBuilderRef builder, NoFuncao *nofuncao, ListaVariaveis* variaveisLocais,
    ListaVariaveis *parametros, ListaInicializacao* inicializacaoLocais){
81     if(compareString(nofuncao -> tipo, "inteiro") == 0){
82         LLVMValueRef Zero64 = LLVMConstInt(LLVMInt64Type(), 0, false);
83         LLVMTypeRef functionReturnType = LLVMInt64TypeInContext(context);
84         LLVMValueRef function = LLVMAddFunction(module, nofuncao -> nome, LLVMFunctionType(functionReturnType, NULL, 0, 0));
85
86         LLVMBasicBlockRef entryBlock = LLVMAppendBasicBlockInContext(context, function, "entry");
87         LLVMBasicBlockRef endBasicBlock = LLVMAppendBasicBlock(function, "end");
88         LLVMPositionBuilderAtEnd(builder, entryBlock);
89
90         codigoIRVariaveis(context, module, builder, variaveisLocais, inicializacaoLocais);
91         LLVMValueRef returnVal = LLVMBuildAlloca(builder, LLVMInt64Type(), "retorno");
92         LLVMBuildStore(builder, Zero64, returnVal);
93
94         LLVMBuildBr(builder, endBasicBlock);
95         LLVMPositionBuilderAtEnd(builder, endBasicBlock);
96         LLVMBuildRet(builder, LLVMBuildLoad(builder, returnVal, ""));
97     }

```

Figura 2. Código que gerar as funções, declarações e atribuição de variáveis e retorno

Após essas duas etapas, os problemas começaram em que um dos problemas foi o seguinte, como atribuir um valor que estava em uma outra variável, qual parâmetro utilizar para invocar essa função, como atribuir um valor sem que seja na hora da inicialização

daquela variável, qual função utilizar em certo momento durante "varredura" na árvore sintática. Então como tentávamos as demais partes que compõe a linguagem de programação T++ e acabamos não conseguindo sucesso, decidimos parar por aqui. Consequentemente, acabamos não conseguindo chegar no objetivo que era a geração do código intermediário, mas de apenas algumas partes do código intermediário, como declarações e atribuições de variáveis globais, declarações de funções, contendo dentro delas as declarações e atribuições das variáveis locais e o retorno das funções.

6. Discussão dos resultados

Como dito anteriormente, alcançamos apenas uma parte do nosso objetivo que era a geração do código intermediário dado um arquivo de entrada, utilizando as funções das bibliotecas do LLVM (*Low Level Virtual Machine*). Quando era alcançado o resultado, conseguimos ver o arquivo no formato '.bc', que pode ser visto na figura 3. Entretanto quando não conseguíamos alcançar esse resultado, uma mensagem de erro mostra qual erro não permitiu a geração do código intermediário e uma mensagem que o arquivo no formato '.bc' não foi criado, que pode ser visualizado na figura 4.

```
; ModuleID = 'meu_modulo.bc'

define i64 @principal() {
entry:
    %b = alloca i64
    store i64 10, i64* %b
    %0 = load i64* %b
    %retorno = alloca i64
    store i64 0, i64* %retorno
    br label %end

end:                                     ; preds = %entry
    %1 = load i64* %retorno
    ret i64 %1
}
```

xandao@danoninho:~/CompiladorEmC/Análise Semântica\$

Figura 3. Resultado da geração de código intermediário, quando não existe erro

```
xandao@danoninho: ~/CompiladorEmC/Análise Semântica
[aviso] a variável a é inutilizada
[aviso] não conseguiu gerar o código intermediário
xandao@danoninho:~/CompiladorEmC/Análise Semântica$
```

Figura 4. Resultado da geração de código intermediário, quando existe erro

Um dos empecilhos que acarretou na geração do código intermediário incompleto, foi os parâmetros que deveriam ser utilizados para compilar utilizando uma determinada versão do LLVM. Isso porque quando colocamos a inclusão de uma biblioteca do LLVM, no código da nossa nova biblioteca apareciam o monte de erros de referência, ou que o valor de uma biblioteca estava incorreto. Uma pequena parte que impediu de gerar o código intermediário, foi não ter tanta familiaridade com as funções presente nessa biblioteca, o que demorou um certo tempo para realizar a geração de certas partes do código de entrada.

7. Conclusão

Após realizar todas essas etapas, conseguimos verificar uma das formas da representação intermediária, já que a partir dele gerará o código-alvo. Algumas partes, que eram geradas pelas funções das bibliotecas LLVM (*Low Level Virtual Machine*) no arquivo do formato '.bc', era muito similares a algumas disciplinas que vimos semestres anteriores, o que nós auxiliou a olhar alguns exemplos para se basear.

Então, julgamos que um dos fatores que acreditamos que talvez puderia nos auxiliar, e consequentemente nos fizesse chegar um pouco mais perto do nosso objetivo, é assim como na etapa anterior, a análise semântica ter um complemento a mais nas tabelas de símbolos, para que talvez não tivéssemos problemas na hora de atribuir um valor presente em uma variável, etc.

Referências

LLVM (2017). Llvm overview.

Louden, K. C. (2004). *COMPILADORES: Princípios e Práticas*. Thomson, 1st edition.