Geração de Código Intermediário

Gabriel Tomaz Lima[1] - 18/0136607

Universidade de Brasília

1 Motivação

O objetivo deste trabalho é implementar um analisador léxico e sintático para uma linguagem modificada subconjunto da linguagem C, mantendo o padrão de tipos e operações do C e desenvolvendo duas novas primitivas chamadas *set* e *elem*. Segundo [1], a primeira etapa de um compilador é o analisador léxico, tendo objetivo principal ler os caracteres de entrada e produzir uma sequência de tokens como saída que futuramente serão utilizados por um parser na análise sintática. O objetivo da segunda fase de um compilador é a análise sintática ou parsing. Esta etapa usa o primeiro componente do token produzido pelo analisador léxico com o intuito de criar uma árvore sintática abstrata intermediária que representa a estrutura gramatical do fluxo de tokens. Uma forma de representar a árvore de sintaxe que em cada nó configura uma operação e os filhos deste nó configuram os argumentos da operação. A terceira etapa de um compilador é a análise semântica, no qual a partir da definição da linguagem é possível determinar possíveis erros semânticos e coerções de tipos. Bem como definir os escopos a partir do código fonte fornecido.

2 Descrição - análise léxica

O analisador léxico lê uma cadeia de caracteres, esta cadeia é proveniente do código recebido como entrada, então o analisador cria os lexemas que são o agrupando dos caracteres. Para cada lexema o analisador produzirá uma saída no formado abaixo.

$$< token - name; attribute - value > \tag{1}$$

A próxima etapa é o analisador sintático que será tratado em uma fase mais avançada deste projeto. No token, o primeiro componente é o token-name que é um símbolo abstrato que será utilizado na análise de sintaxe, e o segundo componente é o attribute-name que aponta para uma entrada na tabela de símbolo para este token.

Caso uma destas entradas, o token não seja reconhecido o analisador deve sinalizar um erro, indicando a linha e a coluna onde ocorreu. O analisador não pode parar a execução quando ocorrer um erro, deve tratá-lo e seguir com o processamento das entradas.

3 Descrição - análise sintática

Em um compilador o analisador obtém uma string de tokens do analisdor léxico, verifica se a sequência de nomes de tokens pode ser gerado pela gramática de origem. Espera-se que o analisador aponte os erros de sintaxe de forma inteligível e possa recuperar-se dos erros mais comuns para continuar o processamento do restante do programa. O analisador constrói uma árvore de análise e passa para o resto do compilador para processamento posterior.

3.1 Árvore Sintática

Para implementação da árvore sintática foi utilizado o software Bison[2]. Neste software regras são definidas para gerar um analisador sintático do tipo bottom-up parser LR(1) canônico. Árvore sintática é construida e preenchida com os terminais(Tokens) e ao final do processo é apresentada. Dois arquivos foram criados para fornecer a estrutura para árvore tree.c, tree.h.

Nesta etapa do analisador semântico a estrutura da árvore precisou ser alterada para comportar a verificação de tipos, portanto foi incluída a variável *type* para cada nó da árvore. Esta variável foi convencionalmente definida para assumir determinados números que dentro do código do analisador tem um significado, assim sendo 0 *int*, 1 *float*, 2 *set*, 3 *elem*, 10 quando ocorrer um erro de conversão e 99 para o nó que não tiver um tipo definido.

```
struct TreeNodes {
    struct TreeNodes* brotherNode;
    struct TreeNodes* childNode;
    char* value;
    int type;
};
```

3.2 Tabela de Símbolos

Ao passo que a árvore sintática é criada, uma tabela de símbolos também é construída com objetivo de armazenar funções ou variáveis declaradas no código analisado. Neste projeto, foram criados dois arquivos nomeados de symbol_table.c, symbol_table.h contendo a estrutura para construção da tabela.

Nesta etapa a estrutura da tabela de símbolos foi alterada para comportar duas novas variáveis, cujo objetivo é incrementar as informações de um símbolo. As variáveis *isFunction* e *numArgs* informam se o símbolo é uma função e caso seja o número de argumentos que a função recebe.

```
struct Symbol{
    char* id;
    char* type;
    int line;
    int col;
```

```
    int isFunction;
    int numArgs;
    Symbol *next;
    Scope *scope;
};
```

4 Descrição - análise semântica

O analisador semântico utiliza a árvore sintática em conjunto com as informações da tabela de símbolos para verificar a consistência do programa que está sendo lido com base na definição da linguagem. O analisador também reúne as informações de tipo e as salva na tabela de símbolos ou na árvore de sintaxe, para ser utilizada na geração de código intermediário [1].

A verificação de tipo é uma parte importante da análise semântica, no qual o compilador verifica se cada operador possui operandos correspondentes. O compilador deve relatar um erro se um número de ponto flutuante for usado para indexar uma matriz, por exemplo [1].

A definição da linguagem pode permitir coerções que são conversões de tipo. Se o operador for aplicado a um número de ponto flutuante e um inteiro, o compilador pode converter ou forçar o número inteiro em um número de ponto flutuante [1].

4.1 Verificações

Nesta etapa da análise semântica algumas verificações são realizadas.

- Os escopos são detectados, bem como os argumentos, funções e a forma como são utilizados.
- Criação e redeclaração de variáveis e funções, conversão de tipos em expressões lógicas, aritméticas e de comparação caso necessário.
- Verificação da quantidade e dos tipos dos argumentos em uma função, bem como a passagem de parâmetros. Incluindo a checagem do retorno da função para verificar se é do mesmo tipo ou se necessita de alguma conversão.
- Detectar se a função *MAIN* é declarada no código.

4.2 Estrutura do Analisador Semântico

Nesta etapa uma estrutura de escopo foi criada em para auxiliar o analisador semântico, cada escopo tem um nome e duas referências, uma para lista de símbolos correspondente e outra para o escopo pai. Os arquivos symbol_table.c, symbol_table.h foram alterados para comportar o escopo.

```
struct Scope {
    char* scopeName;
    Scope *parentScope;
    Symbol *listSymbol;
};
```

Ao passo que o programa de entrada é lido os escopos são criados em conjunto com suas tabelas de símbolos respectivas. Quando um escopo termina a tabela de símbolos do mesmo é exibida no terminal. Um escopo é criado quando uma { é identificada e fechado quando uma } é detectada. Quando o escopo vem de uma função ele recebe o nome da mesma, caso seja de uma instrução ele receberá o nome dela, por exemplo: *FORALL* o escopo é definido com este nome. Os escopos são exibidos de baixo para cima, ou seja, primeiro os filhos e irmãos do escopo são exibidos com suas tabelas de símbolos respectivas e logo depois o escopo pai.

4.3 Conversão de Tipos

A conversão de tipos é feita da seguinte forma:

- intToFloat => converte int para float,
- intToElem => converte de int para elem.
- floatToInt => converte de float para int.
- floatToElem => converte de float para elem.
- elemToFloat => converte elem para float.
- elemToInt => converte elem para int.

5 Geração de Código Intermediário

Após a árvore sintática ser construída e as etapas da análise sintática e semântica já ocorreram, sínteses adicionais podem ser realizadas avaliando atributos e executando fragmentos de código nos nós da árvore. As instruções são traduzidas em código de três endereços usando instruções de salto para implementar o fluxo de controle como, por exemplo, na instrução *if expr then stmt1* um *jump* é realizado [1].

Como manual de referência foi utilizado a documentação do programa (TAC) fornecido na disciplina para fazer a conversão do arquivo gerado pelo tradutor desenvolvido.[3]

5.1 Descrição

Para implementação da geração de código intermediário foi necessário realizar algumas alterações nas estruturas de dados desenvolvidas neste projeto. Os nós da árvore sintática podem receber mais dois parâmetros, *registrador* armazena o número do registrador que guarda a referência para o nó da árvore e o *regis_ tipo* que armazena o tipo do registrador. Na estrutura da tabela de símbolos também foram criados estes dois parâmetros, pois é necessário para buscar o valor do registrador de uma variável que já foi referenciada antes.

5.2 TAC

Nesta versão do projeto, ao passo que a arvóre sintática é construida a geração do código intermédiario também é executada, um buffer do tipo *char* armazena as instrução do código de três endereços e ao final da execução um arquivo de nome *three_ address*.tac é gerado e o buffer gravado no mesmo. Uma variável global é responsável por implementar um contador de registrador para evitar que uma instrução recebe o mesmo número de registrador.

Caso ocorra um erro durante a construção da arvoré, seja ele léxico, sintático ou semantico o arquivo .tac não será gerado.

5.3 Funções

Os novas funções criadas especificamente para este projeto serão implementadas e inseridas no arquivo .tac caso seja necessário o seu uso. Como, por exemplo, *add*, *remove*, *forall* e etc.

6 Descrição - arquivos de teste

6.1 1 - Exemplo

O primeiro exemplo não apresenta erros, sem menções a caracteres inválidos.

6.2 2 - Exemplo

O segundo exemplo não apresenta erros, sem menções a caracteres inválidos.

6.3 3 - Exemplo

No terceiro exemplo contido na pasta *src/exemplos* o arquivo *exemplo3*. Há ocorrência de 4 erros semânticos, 1 erro sintático e um erro léxico.

Semânticos:

- Linha 3 Coluna 15, variável d não declarada.
- Linha 12 Coluna 12, erro de cast a função "soma" espera o retorno int e o código retorna uma variável do tipo set. É possível ver o erro na árvore sintática.
- Linha 18 Coluna 19, função "soma" espera 2 argumentos e apenas um é enviado.
- Linhas 19 Coluna 16, função "funcB" não declarada.

Sintático:

- Linha 22 Coluna 17, não espera uma string como retorno.

Léxico:

- Linha 20 Coluna 10, & token inválido.

6.4 4 - Exemplo

No quarto exemplo contido na pasta *src/exemplos* no arquivo *exemplo4*. Há ocorrência de 3 erros semânticos, 1 erro sintático e um erro léxico.

Semânticos:

- Linha 4 Coluna 13, variável b redeclarada no mesmo escopo. - Linha 17 Coluna 26, função "sub" espera 2 argumentos, porém são enviados 3. - Função main() não declarada.

Sintático:

- Linha 22 Coluna 16, não espera dois símbolos de operação de multiplicação juntos no retorno da função.

Léxico:

- Linha 20 Coluna 17, $ token inválido.

7 Instruções do Projeto

Para executar o código é necessário possuir instalados o GCC e o pacote FLEX e Bison .

Seguindo os comandos abaixo é possível executar o código, que se encontra na pasta *trab3-analisador-semantico/src* .

Ao executar o comando make(1), os comandos dentro do arquivo Makefile serão executados para compilar o projeto, em seguida um arquivo com nome executável de nome *analyse* será criado, este arquivo será executado utilizando o comando (3), que irá printar no console o resultado da análise sintática, no 3 comando também é necessário informar o arquivo que deseja utilizar como entrada, como exemplo do código abaixo. Os arquivos de teste encontram-se na pasta *exemplos*.

7.1 Versões

```
1 - make (GNU Make 3.81)
2 - flex 2.5.35 (flex-32)
3 - bison (GNU Bison) 3.7.4
4 - gcc clang version 11.0.0 (clang-1100.0.33.8)
```

7.2 Comandos

```
1- make
    - flex lexical/lexical.l
    - bison -d sintatic/sintatic.y -Wcounterexamples
    - gcc lex.yy.c sintatic.tab.c symbol_table.c -o analyse tree.c -Wall -ll

2- ./analyse < exemplos/exemplo1.c

3- ./tac three_adress.tac
```

8 Anexos

8.1 Gramática

```
<digito>     ::= 0| ... |9
<STRING>     ::= a|...|z|A|...|Z| _
<ID>         ::= <STRING><STRING>|<digito>*
<INT>        ::= <digito>+
<FLOAT>      ::= <digito>+. | <digito>+
<add_ops>    ::= + | -
<mult_ops>   ::= * | /
<numero>     ::= <INT> | <FLOAT>
<set_tipo>   ::= set
<elem_tipo>  ::= elem
<tipos_basicos>  ::= <numero> | <set_tipo> | <elem_tipo>
<logical_ops> ::= '<' | '>' | '>=' | '<=' | '==' | '!='
<term>       ::= <ID> | <tipos_basicos> | '(' expr ')'
<list_expr>  ::= <expr> ',' <list_expr> | <expr>
<first_term> ::= <term> | "!" <term>  | <adds_op> <term>
        | <ID> '(' list_expr ')'
        | <ID> '()'
<mult_expr> ::= <mult_expr> <mult_ops> <first_term> | <first_term>
<arithmetic_expr> ::= <arithmetic_expr> <add_ops> <mult_expr> | <mult_expr>
<logical_expr> ::= <logical_expr> <logical_ops> <arithmetic_expr>
        | <arithmetic_expr>
<op_and_expr>  ::= <op_and_expr> '&&' <logical_expr>
        | <logical_expr>
<op_or_expr> ::=  <op_or_expr> '||' <op_and_expr>
        | <op_and_expr>
        | <func_in_expr>
<func_in_expr> ::= <op_or_expr> 'IN' <ID> | <op_or_expr> 'IN' <func_expr>
<is_set_expr>  ::= is_set '(' var ')' | is_set '(' <func_expr> ')'
<func_expr>    ::= add '(' <func_in_expr> ')'
        | remove '(' <func_in_expr> ')'
        | exist '(' <func_in_expr> ')'
<assign> ::= <ID> '=' <expr>
<expr> ::= <op_or_expr> | <func_expr>
<expr_stmt> ::= <expr> ';'
<set_stmt> ::=  forall '(' <ID> 'IN' <func_expr> ')' <simple_complex_block_stmt>
        | forall '(' <ID>  'IN' <ID> ')' <simple_complex_block_stmt>
        | <is_set_expr> ';'
<return_stmt> ::= return ';'
        | return <expr> ';'
<simple_complex_block_stmt> ::= <stmt>
        | <blockStmt>
```

```
<block_cond> ::= <simple_complex_block_stmt>
        | <simple_complex_block_stmt> else <simple_complex_block_stmt>
<condition_expr> ::= if '(' <expr> ')'  <block_cond>
<iteration_expr> ::= for '(' <assign> ';' <expr> ';' <assign> ')' <blockStmt>
<input_output_expr> ::= write '(' <STRING>  ')' ';'
        | write '(' <expr> ')' ';'
        | writeln '(' <STRING>  ')' ';'
        | writeln '(' <expr> ')' ';'
        | read '(' <ID> ')' ';'
<stmt> ::=   <iteration_expr>
        | <condition_expr>
        | <return_stmt>
        | <input_output_expr>
        | <expr_stmt>
        | <set_stmt>
        | <var_declaration>
        | <assign> ';'
<list_statements> ::= <stmt> <list_statements> | <stmt>
<blockStmt> ::= '{' <list_statements? '}'
<list_args> ::= <tipos> <id> ',' <list_args> | <tipos> <ID>
<func_declaration> ::= <tipos> <ID> '(' <list_args> ')' <blockStmt>
        | <tipos> main '(' <list_args> ')' <blockStmt>
<var_declaration> ::= <tipos> <ID> ';'
<main_declaration> ::= <func_declaration> | <var_declaration>
<list_declaration>  ::= <list_declaration> <main_declaration>
        | <main_declaration>
<program> ::= <list_declaration>
```

References

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. Corbett, R., Stallman, R.: Bison - gnu bison - the yacc-compatible parser generator, https://www.gnu.org/software/bison/manual/, accessed: 2021-03-17
3. Santos, L., Nalon, C.: TAC interpretador de código de três endereços - manual de referência. https://github.com/lhsantos/tac, accessed: 2021-05-05