Tradutor

Vitor Fernandes Dullens - 16/0148260

Universidade de Brasília

1 Introdução

Este relatório abordará as primeiras etapas do processo de criação de um tradutor, que consistem na implementação do analisador léxico, sintático, semântico e da geração do código de três endereços para uma linguagem específica, como apresentado no livro base da disciplina [ALSU07]. Neste documento serão descritos com detalhes a motivação por trás desta implementação, assim como a descrição das análises e instruções para a compilação e execução do programa. Também será apresentada uma descrição da linguagem a ser analisada.

2 Motivação

Na linguagem C, muitas vezes sentimos falta de operações que facilitam a utilização de conjuntos, como existem em outras linguagens de mais alto nível. Dito isso, a fim de facilitar essas operações, uma implementação de uma nova primitiva de dados para conjuntos foi proposta dentro da linguagem C – `set`, assim como operações para a mesma – `add`, `remove`, entre outras.

Abaixo segue um exemplo de código na nova linguagem proposta.

```
int main() {
    set s;
    s = EMPTY;

    add(1 in add(2 in add(3 in s)));
    /* s = (1, 2, 3) */

    remove(1 in s);
    /* s = (2, 3) */
}
```

Além de operações com conjuntos, também foi adicionado um tipo polimórfico – `elem` que facilita, também, o uso de conjuntos. Mais detalhes sobre a linguagem podem ser encontrados no Apêndice A.

3 Descrição da análise léxica

Para a implementação do analisador léxico foi utilizado o programa *Fast Lexical Analyzer Generator* - FLEX [Est17], que consiste em uma ferramenta geradora de programas que reconhecem padrões léxicos em textos.

No arquivo de nome `lexico.l` é possível visualizar as regras léxicas. Para o tratamento das mesmas são declaradas expressões regulares (regex) que as identificam e após essas declarações existe uma sequência de ações que o analisador executa ao encontrar uma regra. Além das regras e ações, no arquivo `lexico.l` também foram definidas duas variáveis `int linha`, `int coluna`, que representam, respectivamente, a linha e a coluna na qual está acontecendo a ação, e uma variável externa `int erros` que abriga a quantidade de erros que foram encontrados durante toda a execução das análises.

O analisador léxico também é responsável pela atribuição dos tokens que serão utilizados durante a análise sintática. Os tokens são declarados como uma `struct Lexema` que possui os parâmetros `char* corpo` que consiste no lexema propriamente dito, `int linha` e `int coluna` que, respectivamente, representam a linha e coluna daquele token. Essa `struct` é declarada no arquivo do analisador sintático, e pode ser utilizada no analisador léxico utilizando a variável global `yylval`.

4 Descrição da análise sintática

Para a implementação do analisador sintático foi utilizado o programa Bison [DS21], que consiste em um gerador de analisador que utiliza de uma gramática livre do contexto para criar uma derivação LR. Para este trabalho foi utilizada a flag %`define lr.type canonical-lr` para que a derivação realizada seja a LR(1) canônica.

No arquivo de nome `sintatico.y` é possível visualizar a gramática presente no Apêndice A com algumas modificações para se encaixar na sintaxe do próprio Bison.

4.1 Árvore Sintática

Pelo modo como é feita a análise pelo próprio Bison, é possível construir uma árvore sintática abstrata. Para isso, cada não terminal agora é um nó da árvore, e cada terminal é um símbolo, assim como será apresentado na Seção 4.2. Cada um desses nós, consiste em uma estrutura que armazena o símbolo (terminal), qual a regra que foi vista, o nó próximo e o filho. Com essas informações é possível realizar um caminho em profundidade a partir da raiz apresentando cada um dos nós que foram visitados e todos os detalhes necessários.

A Figura 1 representa a árvore gerada para um programa de entrada simples.

4.2 Tabela de Símbolos

Durante a passagem do analisador sintático, ele também é responsável por salvar os símbolos, afim de utiliza-los futuramente na análise semântica.

A tabela de símbolos é em uma lista de símbolos, onde um símbolo consiste em uma estrutura que armazena dados que podem ser úteis na próxima etapa do projeto, como o ID da variável ou função declarada, suas respectivas linha e coluna, tipo, se é uma função, variável ou um parâmetro, e o escopo.

```
------------------ARVORE SINTATICA------------------
program
 | function_declaration--- [ identificador -> main ]
 |  | INT_TYPE
 |  | stmts
 |  |  | var_declaration--- [ identificador -> x ]
 |  |  |  | INT_TYPE
 |  | stmts
 |  |  | assignment--- [ identificador -> x ]--- -> INT
 |  |  |  | sum_exp--- [ operador -> + ]--- -> INT [ CAST -> float2int ]
 |  |  |  |  | CONST--- [ constante -> 1.2 ]--- -> FLOAT
 |  |  |  |  | primal_exp--- [ identificador -> x ]--- -> FLOAT [ CAST -> int2float ]
 |  | return_stmt
 |  |  | CONST--- [ constante -> 1 ]--- -> INT
```

Figura 1. Árvore Sintática Abstrata para uma entrada simples

```
|======================TABELA DE SIMBOLOS=======================|
|= IDENTIFICADOR | LINHA:COLUNA | TIPO | PARAM/VAR/FUNC | ESCOPO =|
|================|==============|======|================|=========|
| f              |   1:5        | INT  | FUNC           | 0       |
| x              |   1:18       | INT  | PARAM          | 1       |
| a              |   1:11       | INT  | PARAM          | 1       |
| main           |   5:5        | INT  | FUNC           | 0       |
| x              |   6:9        | INT  | VAR            | 2       |
| s              |   7:9        | SET  | VAR            | 2       |
| a              |   9:11       | FLOAT| VAR            | 2       |
|=================================================================|
```

Acima é possível visualizar um exemplo da tabela de símbolos que é apresentada ao final do programa. Vale notar a presença de funções (FUNC), parâmetros (PARAM) e variáveis (VAR), contendo também seus respectivos escopos, tipos, identificadores e posições.

5 Descrição da análise semântica

Como já se sabe, a análise semântica é responsável por verificar se o programa "faz sentido"ou não, ou seja, verficação de tipos de variáveis, realização de coersões de tipo, verficação de quantidade de parâmetros, se varáveis foram previamente declaradas, entre outras.

Dito isso, como o Bison [DS21] já faz uma passagem por todo o programa na análise sintática, utilizamos desta mesma passagem para fazer a maior parte das verficações necessárias da análise semântica. Abaixo seguem subseções sobre as principais regras utilizadas para esta etapa.

5.1 Regras de escopo

Para a implementação dos escopos, foi utilizada a estrutura de dados de pilha. Como em C não existe uma estrutura pronta para isso, foi criado uma abstração desta estrutura, utilizando um array junto com um índice, onde o índice aponta sempre para o último elemento do array.

As funcionalidades do escopo são como as esperadas em um programa em C, na Seção 4.2 é possível visualizar a tabela de símbolos para um dos arquivos testes, e nela se encontram o escopo de cada uma das variáveis, parâmetros e funções.

As verificações feitas durante a análise semântica consistem em, verificar se uma variável ou função foi previamente declarada, se não existem múltiplas declarações da mesma variável no mesmo escopo ou múltiplas declarações da mesma função e, também, a verficação se a função `main` foi definida. Todas essas verificações são feitas com o auxílio da tabela de símbolos, que abriga todos os dados necessários para esta análise.

5.2 Regras de passagem de parâmetros

Quando uma função é declarada, são associados seus parâmetros a ela, ou seja, cada símbolo da tabela que for uma função, acompanha uma lista de parâmetros. Além disso, os parâmetros são apresentados na tabela de símbolos e propriamente identificados como é possível ver na Seção 4.2.

Ao ser chamada, uma função passa por duas verificações, a primeira consiste em verificar se a quantidade de argumentos é igual a quantidade de parâmetros que foram declarados na criação da função, caso isso não seja correspondido, o programa gera um erro, apresentando quantos parâmetros foram passados e quantos são esperados para aquela função.

A segunda verificação, é a análise dos tipos, ela será mais abordada na Seção 5.3, mas rápidamente, esta etapa consiste em verificar se os tipos dos argumentos correspondem aos tipos declarados na função, se os tipos não forem iguais, tenta-se fazer a conversão de tipos, caso isso não seja possível, por incompatibilidade dos tipos, um erro é lançado, apresentando o tipo passado e o tipo esperado.

5.3 Decisões sobre conversões de tipos

Como apresentado anteriormente, a motivação para esta linguagem é facilitar a utilização de conjuntos. Dito isso, os tipos presentes na linguagem são `int`, `float`, `set` e `elem`, sendo os dois últimos mais destinados para a utilização de conjuntos.

Para realizar as coersões necessárias, foram adicionadas informações de tipos para cada nó da árvore, como é possível visualizar na Figura 1. Quando há uma coersão de tipo, ela é sinalizada como `CAST` e em seguida é especificado qual coersão foi realizada, por exemplo, `int2float` significa que o nó era do tipo `int` e que fora convertido para o tipo `float`.

Alguns tipos e funções tem suas particularidades, por exemplo, não é possível converter um `int` ou um `float` para o tipo `set`, visto que são tipos muito distintos. Abaixo seguem algumas das regras específicas para cada tipo:

- `int` pode ser convertido para `float`, e vice-versa;
- O tipo `elem` pode ser convertido para qualquer um dos outros tipos, e vice-versa;
- A constante `EMPTY` é tratada como vazio (`NULL`) e só pode ser utilizada em tipos `elem` ou `set`;
- Funções como `is_set` ou para verificação de uma variável dentro de um `set` (i.e: `1 in s`) retornam o tipo `int`.
- Funções como `add` e `remove` retornam o tipo `set` e na parte direita dessas expressões sempre é esperado o tipo `set`;
- A função `exists` retorna o tipo `elem`;
- Não é possível realizar operações aritméticas com o tipo `set`;
- Operações lógicas (`&&` e `||`) e relacionais (==, !=, etc) sempre retornam o tipo `int`, contudo nas operações relacionais dependem do tipo das expressões, enquanto operações lógicas podem ser realizadas com qualquer tipo (i.e: `int i; set s; (s && a);`

6 Descrição dos arquivos testes

Os testes se encontram na pasta `testes/`, os arquivos `sucesso1.c` e `sucesso2.c`, são testes que contém código correto, já os arquivos `erro1.c` e `erro2.c` contém códigos incorretos, sendo os seus erros, respectivamente:

1. ```
   SEMANTIC ERROR  3:9   - '+' operator do not supports
     the type SET
   SEMANTIC ERROR  4:7   - Redeclaration of variable 'b'
   SEMANTIC ERROR  14:3   - Incompatible types can't be
     casted -> SET to FLOAT
   SEMANTIC ERROR  14:3   - Fewer number of arguments in
     function  'a' - expected: 2 , received: 1
   SEMANTIC ERROR  15:3   - Incompatible types can't be
     casted -> INT to NULL
   SEMANTIC ERROR  15:3   - Greater number of arguments in
     function  'a' - expected: 2 , received: 3
   SEMANTIC ERROR  - The program doens't have a 'main' function
   ```

2. ```
   LEXICAL ERROR   2:8   - Unidentified character: #
   SYNTAX ERROR    3:8   - syntax error, unexpected ';',
     expecting '='
   SYNTAX ERROR    3:14  - syntax error, unexpected ')'
   SYNTAX ERROR    9:3   - syntax error, unexpected INT_TYPE
   SEMANTIC ERROR  11:5   - Incompatible types can't be
     casted -> INT and SET
   ```

Basicamente, como é possível observar acima, os erros são apresentados e possuem a linha e coluna na qual eles pertencem (linha:coluna). O primeiro arquivo de erro - `erro1.c` - possui apenas erros semânticos, já o segundo arquivo - `erro2.c` - possui tanto erros léxicos, quanto erros sintáticos e semânticos.

Quando um arquivo possui apenas erros semânticos é apresentada a árvore sintática, porém caso existam erros sintáticos ou léxicos ela não é impressa. A tabela de símbolos gerada (que pode não estar completa dependendo do tipo dos erros) é sempre apresentada, independente do número de erros. Além disso, cada erro é apresentado seguido de uma descrição do mesmo.

7 Geração de Código Intermediário

A geração do código intermediário foi feita baseada nas instruções disponibilizadas na documentação do programa TAC [San15], já que o objetivo final deste trabalho é que seja gerado um código no qual ao ser executado por este programa funcione da forma correta.

Basicamente, passando uma visão geral, o TAC [San15] possui duas seções principais, a tabela de símbolos (`.table`), onde são feitas as declarações de variáveis e constantes e a seção de código (`.code`) que armazena todas as funções e expressões presentes no programa.

Como todas as outras análises são feitas em uma única passagem, nesta também utilizamos da mesma passagem para construir o código de três endereços(3AC). Dito isso, alterações na estrutura do nó da árvore foram necessárias, adicionando agora uma estrutura correspondente ao 3AC que armazena os dados utilizados para a construção do mesmo, como a operação, os argumentos e o destino.

Falando sobre a passagem pela árvore, vale notar que existem alguns tipos diferentes de dados que são armazenados. Para este projeto, sempre que houver uma declaração de variável ou `string`, essa declaração é armazenada na seção `.table` do TAC [San15]. As demais operações são adicionadas na seção `.code` e serão mais descritas abaixo:

- Para variáveis temporárias que são criadas durante a execução do programa, foram utilizados os registradores disponibilizados, estes são utilizados em ordem crescente ($0, $1, $2, ...) e conforme são necessários.
- Todas as operações passam por uma checagem de tipo antes de serem escritas, fazendo assim com que a conversão de tipos, quando necessária, seja executada antes da operação, fazendo com que o código não possua problemas de tipagem.
- Foram criadas e prefixadas, no início da seção, algumas funções que podem ser utilizadas durante a execução do TAC, como por exemplo as funções `write` e `writeln` para que seja possível escrever na tela uma `string` com múltiplos caracteres, ou seja, ao escrever uma `string`, ao invés de criar uma repetição toda vez, utiliza-se da função prefixada, passando como parâmetro o tamanho e o endereço da `string` a ser lida.

– O TAC possui a maioria das operações básicas necessárias, porém algumas não estão inclusas já que podem ser obtidas alterando posições de operadores ou adicionando operações. Dito isso a operação relacional `!=` consiste em uma negação da operação `==` e as operações `>` e `>=` são iguais as de `<` e `<=`, apenas alterando a ordem dos argumentos.
– Algumas operações como `for`, `if` e `else` precisam de rótulos para serem executadas de forma correta, sendo eles `for_&`, `end_for_&`, `end_if_&` e `end_if_else_&`, onde '&' é um número que os diferencia entre si, caso a exista mais de um `if` ou `for`.

8 Instruções para compilação e execução do programa

O programa foi criado e testado em um sistema operacional Linux - Ubuntu 20.04.1 LTS. É necessária a instalação do FLEX [Est17] e do BISON [DS21] para a compilação do programa. Ao executar o programa também deverá ser passado o arquivo que será analisado. Além disso, para poder executar o código de três endereços gerado, é necessária, também, a instalação do TAC [San15].

Comandos para compilação e execução:

```
$ bison sintatico.y
$ flex lexico.l
$ gcc sintatico.tab.c lex.yy.c tabela.c arvore.c tac.c
$ ./a.out testes/<nome-arquivo>.c
$ ./tac output_tac/output.tac
```

Outra alternativa para facilitar a compilação seria utilizar o comando `make`.

Abaixo segue as versões de cada software que foi utilizado para o desenvolvimento:

```
$ gcc --version
gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
$ bison --version
bison (GNU Bison) 3.7.6
$ flex --version
flex 2.6.4
```

Referências

[ALSU07] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, & Tools*. Pearson/Addison Wesley, 2nd edition, 2007.

[DS21] C. Donnelly and R. Stallman. Bison - the yacc-compatible parser generator, 2021. Online; Acessado 18 de março de 2021 https://www.gnu.org/software/bison/manual/bison.html.

[Est17] W. Estes. Flex: Fast lexical analyzer generator, 2017. Online; Acessado 21 de fevereiro de 2021 https://github.com/westes/flex.

[San15] L. Santos. Tac - the three address code interpreter, 2015. Online; Acessado 6 de maio de 2021
https://github.com/lhsantos/tac.

A Gramática

⟨*program*⟩ ::= ⟨*declaration_list*⟩

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

⟨*declaration*⟩ ::= ⟨*function_declaration*⟩ | ⟨*var_declaration*⟩

⟨*var_declaration*⟩ ::= ⟨*type*⟩ ⟨*id*⟩ ‘`;`’

⟨*function_declaration*⟩ ::= ⟨*type*⟩ ⟨*id*⟩ ‘`(`’ ⟨*params_list*⟩ ‘`)`’ ⟨*brackets_stmt*⟩
| ⟨*type*⟩ ⟨*id*⟩ ‘`(`’ ‘`)`’ ⟨*brackets_stmt*⟩

⟨*params_list*⟩ ::= ⟨*type*⟩ ⟨*id*⟩ ‘`,`’ ⟨*param_list*⟩ | ⟨*type*⟩ ⟨*id*⟩

⟨*stmts*⟩ ::= ⟨*stmt*⟩ ⟨*stmts*⟩ | ⟨*stmt*⟩

⟨*stmt*⟩ ::= ⟨*for_stmt*⟩ | ⟨*if_else_stmt*⟩ | ⟨*return_stmt*⟩ | ⟨*io_stmt*⟩
| ⟨*brackets_stmt*⟩
| ⟨*exp_stmt*⟩
| ⟨*set_stmt*⟩
| ⟨*var_declaration*⟩
| ⟨*assignment*⟩ ‘`;`’

⟨*assignment*⟩ ::= ⟨*id*⟩ ‘`=`’ ⟨*exp*⟩

⟨*brackets_stmt*⟩ ::= ‘`{`’ ⟨*stmts*⟩ ‘`}`’

⟨*io_stmt*⟩ ::= read ‘`(`’ ⟨*id*⟩ ‘`)`’ ‘`;`’
| write‘`(`’ ⟨*string*⟩ | ⟨*exp*⟩ | ⟨*char*⟩ ‘`)`’ ‘`;`’
| writeln‘`(`’ ⟨*string*⟩ | ⟨*exp*⟩ | ⟨*char*⟩ ‘`)`’ ‘`;`’

⟨*for_stmt*⟩ ::= for ‘`(`’ ⟨*assignment*⟩ ‘`;`’ ⟨*exp*⟩ ‘`;`’ ⟨*assignment*⟩ ‘`)`’ ⟨*stmt*⟩

⟨*if_else_stmt*⟩ ::= if ‘`(`’ ⟨*exp*⟩ ‘`)`’ ⟨*stmt*⟩
| if ‘`(`’ ⟨*exp*⟩ ‘`)`’ ⟨*stmt*⟩ else ⟨*stmt*⟩

⟨*return_stmt*⟩ ::= return ‘`;`’ | return ⟨*exp*⟩ ‘`;`’

⟨*set_stmt*⟩ ::= forall‘`(`’ ⟨*id*⟩ in ⟨*set_exp*⟩ ‘`)`’ ⟨*stmt*⟩

⟨*exp_stmt*⟩ ::= ⟨*exp*⟩ ‘`;`’ | ‘`;`’

⟨*exp*⟩ ::= ⟨*or_exp*⟩

⟨*set_exp*⟩ ::= add‘`(`’ ⟨*set_in_exp*⟩ ‘`)`’
| remove‘`(`’ ⟨*set_in_exp*⟩ ‘`)`’
| exists‘`(`’ ⟨*set_in_exp*⟩ ‘`)`’

⟨*set_aux_exp*⟩ ::= ⟨*id*⟩ ‘`in`’ ⟨*or_exp*⟩

⟨*set_in_exp*⟩ ::= ⟨*or_exp*⟩ ‘`in`’ ⟨*set_exp*⟩ | ⟨*or_exp*⟩ ‘`in`’ ⟨*id*⟩

⟨*or_exp*⟩ ::= ⟨*or_exp*⟩ ‘`||`’ ⟨*and_exp*⟩ | ⟨*and_exp*⟩ | ⟨*set_in_exp*⟩

⟨*and_exp*⟩ ::= ⟨*and_exp*⟩ '`&&`' ⟨*relational_exp*⟩ | ⟨*relational_exp*⟩

⟨*relational_exp*⟩ ::= ⟨*relational_exp*⟩ ⟨*relational_op*⟩ ⟨*sum_exp*⟩ | ⟨*sum_exp*⟩

⟨*relational_op*⟩ ::= '`<`' | '`>`' | '`>=`' | '`<=`' | '`==`' | '`!=`'

⟨*sum_exp*⟩ ::= ⟨*sum_exp*⟩ '`+`' ⟨*mul_exp*⟩
| ⟨*sum_exp*⟩ '`-`' ⟨*mul_exp*⟩
| ⟨*mul_exp*⟩

⟨*mul_exp*⟩ ::= ⟨*mul_exp*⟩ '`*`' ⟨*primal_exp*⟩
| ⟨*mul_exp*⟩ '`/`' ⟨*primal_exp*⟩
| ⟨*unary_exp*⟩

⟨*unary_exp*⟩ ::= '`!`'? ⟨*primal_exp*⟩ | '`-`' ⟨*primal_exp*⟩
| '`!`'? ⟨*id*⟩ '`(`' arg_list '`)`'
| '`!`'? ⟨*id*⟩ '`(`' '`)`'
| '`!`'? is_set '`(`' ⟨*id*⟩ | ⟨*set_exp*⟩ '`)`'

⟨*primal_exp*⟩ ::= ⟨*id*⟩ | ⟨*const*⟩ | '`(`' ⟨*exp*⟩ '`)`' | ⟨*set_exp*⟩

⟨*arg_list*⟩ ::= ⟨*exp*⟩, ⟨*arg_list*⟩ | ⟨*exp*⟩

⟨*type*⟩ ::= ⟨*basic_type*⟩ | ⟨*elem_type*⟩ | ⟨*set_type*⟩

⟨*const*⟩ ::= ⟨*int_const*⟩ | ⟨*float_const*⟩ | ⟨*empty_const*⟩

⟨*int_const*⟩ ::= ⟨*digit*⟩+

⟨*float_const*⟩ ::= ⟨*digit*⟩+ '`.`' ⟨*digit*⟩*

⟨*empty_const*⟩ ::= '`EMPTY`'

⟨*elem_type*⟩ ::= elem

⟨*set_type*⟩ ::= set

⟨*int_type*⟩ ::= int

⟨*float_type*⟩ ::= float

⟨*string*⟩ ::= `.*`

⟨*char*⟩ ::= '.*'

⟨*id*⟩ ::= [a-zA-Z_][_a-z0-9A-Z]*

⟨*digit*⟩ ::= [0-9]