

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) */

    return 0;
}
```

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.1` 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.1` 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 construção 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 um autômato capaz de reconhecer esta linguagem. 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 autômato, 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.

```

-----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

4.2 Tabela de Símbolos

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

A tabela de símbolos é 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, se é uma função, variável ou um parâmetro, e o escopo.

| =====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 |
| p | 11:11 | ELEM | VAR | 3 |
| ===== | | | | |

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, verificação de tipos de variáveis, realização de

coerções de tipo, verificação de quantidade de parâmetros, se variáveis foram previamente declaradas, entre outras.

Dito isso, como o automato construído pelo Bison [DS21] já faz uma passagem por todo o programa na análise sintática, utilizei desta mesma passagem para fazer a maior parte das verificações 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 uma lista junto com um índice, onde o índice aponta sempre para o último elemento da lista.

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: 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 verificação 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 rapidamente, 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 destinados para a utilização de conjuntos.

Para realizar as coerções necessárias, foram adicionadas informações de tipos para cada nó da árvore, como é possível visualizar na Figura 1. Quando há uma coerção de tipo, ela é sinalizada como `CAST` e em seguida é especificado qual coerção 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` (e.g: `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 (e.g: `int i; set s; (s && a);`);

6 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 utilizei 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.

Para que o programa seja executado da forma esperada foram criadas e prefixadas, no início da seção `.code`, 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.

Além disso 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.

Vale abordar também que 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`.

Nas subseções a seguir serão abordados mais afundo decisões de projeto para a criação do código intermediário.

6.1 Variáveis e conversões de tipos

Para este projeto, sempre que houver uma declaração de variável ou `string`, essa declaração é armazenada na seção `.table` do TAC [San15]. Essas declarações são feitas utilizando o nome da variável e seu respectivo escopo (i.e: `int x_0`, variável do tipo `int`, chamada 'x' e no escopo '0'), fazendo com que as operações funcionem da forma esperada, respeitando seus respectivos escopos.

Já para variáveis temporárias, que são criadas para a execução do programa, foram utilizados os registradores disponibilizados dentro da seção `.code`, estes são utilizados em ordem crescente (`$0`, `$1`, `$2`, ...) e conforme são necessários.

Sobre as conversões de tipo, todas as operações passam por uma checagem de tipo antes de serem escritas, esta checagem utiliza das operações de coerção realizadas na análise semântica, traduzindo-as para a linguagem reconhecida pela TAC, definindo, assim, quais devem ser as conversões realizadas e executando-as, quando necessário, antes da operação, fazendo com que o código não possua problemas de tipagem.

6.2 Novas primitivas - `set` e `elem`

Para a nova primitiva `elem` são utilizados três espaços de memória na `heap` disponível no TAC, estes são responsáveis por armazenar o tipo do elemento, o seu valor e o endereço para o próximo elemento, caso necessário.

A primitiva `set` é um conjunto de `elem`'s e para percorrer o `set`, utilizasse do endereço do próximo elemento armazenado dentro do `elem`.

As funções relacionadas a primitiva `set` também são prefixadas no início da seção. Para a função `add` verificamos primeiro se ele já está presente neste conjunto, caso contrário, o adicionamos. Para a função `remove` apenas percorre-se a lista em busca do elemento, caso ele exista, o removemos, caso contrário,

a função retorna nulo. Já para a função `in` dois tipos de tratamentos foram necessários, um quando é utilizado dentro de um `forall`, que neste caso, ele retorna o primeiro elemento do `set` e vai modificando-o ao decorrer das iterações, outro quando é utilizado como uma operação booleana, que no caso deve-se percorrer a lista de elementos, se encontrar o valor solicitado retornar 1, caso contrário 0. Por fim, para a função `is_set` apenas verifica-se o tipo do elemento dentro da heap, se for um `set` retorna 1, caso contrário, 0.

7 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 doesn'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. Cada erro é apresentado seguido de uma descrição do mesmo e o código intermediário não é gerado se o programa possuir qualquer tipo de erro.

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

$$\begin{aligned}
\langle \text{program} \rangle &::= \langle \text{declaration_list} \rangle \\
\langle \text{declaration_list} \rangle &::= \langle \text{declaration} \rangle \langle \text{declaration_list} \rangle \mid \langle \text{declaration} \rangle \\
\langle \text{declaration} \rangle &::= \langle \text{function_declaration} \rangle \mid \langle \text{var_declaration} \rangle \\
\langle \text{var_declaration} \rangle &::= \langle \text{type} \rangle \langle \text{id} \rangle \text{' ; ' } \\
\langle \text{function_declaration} \rangle &::= \langle \text{type} \rangle \langle \text{id} \rangle \text{' (' } \langle \text{params_list} \rangle \text{') ' } \langle \text{brackets_stmt} \rangle \\
&\mid \langle \text{type} \rangle \langle \text{id} \rangle \text{' (' ' ' } \langle \text{brackets_stmt} \rangle \\
\langle \text{params_list} \rangle &::= \langle \text{type} \rangle \langle \text{id} \rangle \text{' , ' } \langle \text{param_list} \rangle \mid \langle \text{type} \rangle \langle \text{id} \rangle \\
\langle \text{stmts} \rangle &::= \langle \text{stmt} \rangle \langle \text{stmts} \rangle \mid \langle \text{stmt} \rangle \\
\langle \text{stmt} \rangle &::= \langle \text{for_stmt} \rangle \mid \langle \text{if_else_stmt} \rangle \mid \langle \text{return_stmt} \rangle \mid \langle \text{io_stmt} \rangle \\
&\mid \langle \text{brackets_stmt} \rangle \\
&\mid \langle \text{exp_stmt} \rangle \\
&\mid \langle \text{set_stmt} \rangle \\
&\mid \langle \text{var_declaration} \rangle \\
&\mid \langle \text{assignment} \rangle \text{' ; ' } \\
\langle \text{assignment} \rangle &::= \langle \text{id} \rangle \text{' = ' } \langle \text{exp} \rangle \\
\langle \text{brackets_stmt} \rangle &::= \text{' { ' } \langle \text{stmts} \rangle \text{' } \\
\langle \text{io_stmt} \rangle &::= \text{read ' (' } \langle \text{id} \rangle \text{') ' ' ; ' } \\
&\mid \text{write ' (' } \langle \text{string} \rangle \mid \langle \text{exp} \rangle \mid \langle \text{char} \rangle \text{') ' ' ; ' } \\
&\mid \text{writeln ' (' } \langle \text{string} \rangle \mid \langle \text{exp} \rangle \mid \langle \text{char} \rangle \text{') ' ' ; ' } \\
\langle \text{for_stmt} \rangle &::= \text{for ' (' } \langle \text{assignment} \rangle \text{' ; ' } \langle \text{exp} \rangle \text{' ; ' } \langle \text{assignment} \rangle \text{') ' } \langle \text{stmt} \rangle \\
\langle \text{if_else_stmt} \rangle &::= \text{if ' (' } \langle \text{exp} \rangle \text{') ' } \langle \text{stmt} \rangle \\
&\mid \text{if ' (' } \langle \text{exp} \rangle \text{') ' } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle \\
\langle \text{return_stmt} \rangle &::= \text{return ' ; ' } \mid \text{return } \langle \text{exp} \rangle \text{' ; ' } \\
\langle \text{set_stmt} \rangle &::= \text{forall ' (' } \langle \text{id} \rangle \text{ in } \langle \text{set_exp} \rangle \text{') ' } \langle \text{stmt} \rangle \\
\langle \text{exp_stmt} \rangle &::= \langle \text{exp} \rangle \text{' ; ' } \mid \text{' ; ' } \\
\langle \text{exp} \rangle &::= \langle \text{or_exp} \rangle \\
\langle \text{set_exp} \rangle &::= \text{add ' (' } \langle \text{set_in_exp} \rangle \text{') ' } \\
&\mid \text{remove ' (' } \langle \text{set_in_exp} \rangle \text{') ' } \\
&\mid \text{exists ' (' } \langle \text{set_in_exp} \rangle \text{') ' } \\
\langle \text{set_aux_exp} \rangle &::= \langle \text{id} \rangle \text{' in ' } \langle \text{or_exp} \rangle \\
\langle \text{set_in_exp} \rangle &::= \langle \text{or_exp} \rangle \text{' in ' } \langle \text{set_exp} \rangle \mid \langle \text{or_exp} \rangle \text{' in ' } \langle \text{id} \rangle \\
\langle \text{or_exp} \rangle &::= \langle \text{or_exp} \rangle \text{' || ' } \langle \text{and_exp} \rangle \mid \langle \text{and_exp} \rangle \mid \langle \text{set in exp} \rangle
\end{aligned}$$

$$\begin{aligned}
\langle and_exp \rangle &::= \langle and_exp \rangle \text{ '&&' } \langle relational_exp \rangle \mid \langle relational_exp \rangle \\
\langle relational_exp \rangle &::= \langle relational_exp \rangle \langle relational_op \rangle \langle sum_exp \rangle \mid \langle sum_exp \rangle \\
\langle relational_op \rangle &::= \text{'<'} \mid \text{'>'} \mid \text{'>='} \mid \text{'<='} \mid \text{'=='} \mid \text{'!='} \\
\langle sum_exp \rangle &::= \langle sum_exp \rangle \text{'+'} \langle mul_exp \rangle \\
&\mid \langle sum_exp \rangle \text{'-'} \langle mul_exp \rangle \\
&\mid \langle mul_exp \rangle \\
\langle mul_exp \rangle &::= \langle mul_exp \rangle \text{'*'} \langle primal_exp \rangle \\
&\mid \langle mul_exp \rangle \text{'/'} \langle primal_exp \rangle \\
&\mid \langle unary_exp \rangle \\
\langle unary_exp \rangle &::= \langle primal_exp \rangle \mid \text{'!'} \langle primal_exp \rangle \mid \text{'-'} \langle primal_exp \rangle \\
&\mid \text{'!?' } \langle id \rangle \text{'(' arg_list ')'} \\
&\mid \text{'!?' } \langle id \rangle \text{'(' ' ')} \\
&\mid \text{'!?' is_set '(' } \langle id \rangle \mid \langle set_exp \rangle \text{' ')} \\
\langle primal_exp \rangle &::= \langle id \rangle \mid \langle const \rangle \mid \text{'(' } \langle exp \rangle \text{' ')} \mid \langle set_exp \rangle \\
\langle arg_list \rangle &::= \langle exp \rangle, \langle arg_list \rangle \mid \langle exp \rangle \\
\langle type \rangle &::= \langle basic_type \rangle \mid \langle elem_type \rangle \mid \langle set_type \rangle \\
\langle const \rangle &::= \langle int_const \rangle \mid \langle float_const \rangle \mid \langle empty_const \rangle \\
\langle int_const \rangle &::= \langle digit \rangle + \\
\langle float_const \rangle &::= \langle digit \rangle + \text{'.'} \langle digit \rangle^* \\
\langle empty_const \rangle &::= \text{'EMPTY'} \\
\langle elem_type \rangle &::= \text{elem} \\
\langle set_type \rangle &::= \text{set} \\
\langle int_type \rangle &::= \text{int} \\
\langle float_type \rangle &::= \text{float} \\
\langle string \rangle &::= \text{.*} \\
\langle char \rangle &::= \text{'.*'} \\
\langle id \rangle &::= [\text{a-zA-Z_}][\text{_a-zA-Z}]^* \\
\langle digit \rangle &::= [0-9]
\end{aligned}$$