|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | |
|  | |  | |
| Relatório Final  *Compiladores – Licenciatura em Engenharia Informática* | | | |
|  |  | |  |
|  |  | | Gonçalo Almeida, 2020218868  João Santos, 2020218995 |

Índice

[Meta 2 - Gramática reescrita 2](#_Toc121608525)

[Meta 2 e 3 - Algoritmos e estruturas de dados da *AST* e da *Symbol Table* 2](#_Toc121608526)

[Meta 4 – Geração de código 2](#_Toc121608527)

# Meta 2 - Gramática reescrita

A gramática para ser reescrita passou por 4 passos:

* Passar do formato EBNF para BNF
  + Passámos o formato {A} para um novo não terminal B, com B = ε | AB (por exemplo, {COMMA ID}, da produção dos FieldDecl passou para FieldCommaId = ε | COMMA ID FieldCommaId)
  + Passámos o formato [A] para um novo não terminal B, com B = ε | A
  + Passámos o formato A = B(C|D|…)E para A = BCE | BDE | … (como por exemplo, ( MethodInvocation | Assignment | ParseArgs ) nos Statements)
* Escolhemos retirar as produções nulas (do formato A = BCD, C = ε | E passámos para A = BD | BED) para obter um maior controlo sobre certos casos como Statement = LBRACE { Statement } RBRACE para Statement = LBRACE RBRACE | LBRACE Statement RBRACE | LBRACE MultipleStatements Statement RBRACE, onde necessitávamos de saber o número de statements num *Block.*
* Para as expressões (Expr), fizemos um ramo extra, ExprNoAssign, onde não se podem fazer assigns; em concreto, em qualquer expressão que não seja, parêntesis, Assignments, MethodInvocation e ParseArgs, não pode haver assigns. Isto evita situações do tipo a + b = c, permitindo na mesma a + (b = c).
* Adicionar as prioridades e associatividades necessárias para remover a ambiguidade da gramática, com %left quando um token é associativo à esquerda e %right quando um token é associativo à direita, sendo a ordem pela qual aparecem crescente de prioridade. Para distinguir o menos e o mais unário dos não unários, adicionámos “%prec UNARY” aos unários, de forma a aplicar a sua prioridade e associatividade corretamente.

# Meta 2 e 3 - Algoritmos e estruturas de dados da *AST* e da *Symbol Table*

A nossa *AST* é construída com nós, seguindo a seguinte estrutura: uma classe/tipo sintática (*type)*, um valor, um tipo semântico(*true*\_*type*) e uma *flag* de impressão (*print\_true\_type*), os parâmetros (caso represente uma função, caso contrário este parâmetro está a *NULL*; de notar que uma função sem parâmetros é registada como tendo um *param* do tipo “”), um ponteiro para o seu filho (*son*) e outro para o seu irmão (*bro*), dois inteiros correspondentes à linha e coluna para impressão de erros e um inteiro para impedir verificação duplicada de nós com o mesmo nome.

A nossa *symbol table* é baseada em 3 estruturas: a tabela de símbolos (estrutura *SymTab*), que em si que possui um tipo (*type*, return da função), um nome (*scope*), os parâmetros, um ponteiro para o primeiro símbolo e um ponteiro para a próxima tabela; cada símbolo tem um nome, um tipo, os parâmetros (mesma regra que nos nós) e um ponteiro para o próximo símbolo; finalmente, cada parâmetro possui um nome, um tipo, uma flag de validade e um ponteiro para o próximo parâmetro.

No nosso código, inicialmente, declaramos duas tabelas de símbolos, um ponteiro para a tabela de símbolos global e um ponteiro para a primeira tabela de símbolos não global (tabelas de símbolos das funções). Na tabela de símbolos global estão presentes todas as variáveis globais e todas as funções. Na tabela de símbolos de cada função estão as variáveis presentes nesta.

Na parte descendente da travessia recursiva da AST adicionamos os símbolos às respetivas tabelas de símbolos e na análise ascendentes anotamos a AST com os tipos semânticos e fazemos a deteção de erros.

# Meta 4 – Geração de código

A geração do código passa por 2 fases:

1. Declarações iniciais:
   1. Definimos constantes que nos permitem imprimir valores booleanos, inteiros e valores decimais;
   2. Declaramos as Strings, sendo que se uma string for usada várias vezes, não é definida várias vezes mas sim é carregado o valor desta;
   3. Variáveis globais, que vamos buscar à tabela de símbolos global, especificada no ponto anterior. Para as declarar é utililizada a instrução global com o respetivo tipo;
   4. Duas funções da linguagem C (*printf* e *atoi*), e uma função que criámos que imprime booleanos;
   5. Finalmente, definimos uma função main do tipo int, que recebe o argc e argv, chama a função main do java e retorna zero, visto que a *main* do Java é void. É de notar que o .length é sempre igual ao argc-1 (por isso declaramos argc -1 como global), e args[k] em Java é o correspondente a argv[k+1] em C.
2. Geração do código presente nas funções
   1. Percorremos a AST de forma recursiva de modo a gerar as instruções;
   2. Para as variáveis e funções, convencionamos que:
      1. as variáveis globais são declaradas com @ seguido do próprio nome
      2. as variáveis locais são declaradas com % seguido de um contador
      3. as funções são definidas da seguinte maneira: define tipo @nome.tipo\_param1.tipo\_param2…(tipo\_param1 nome\_param1, …)
   3. Para as variáveis locais, é utilizado a instrução *alloca* com o respetivo tipo e guardado na variável %counter, sendo counter uma variável contadora que volta a zero no fim de cada função (os parâmetros das funções também são registados desta maneira);
   4. Para acedermos ao valor das variáveis, temos duas listas ligadas (uma global e uma local que é apagada no fim de cada função) que associam o nome da variável ao respetivo %counter/@nome onde está guardado, através da função load;
   5. Para facilitar o processo de leitura dos valores de literais (int, …), de variáveis e do resultado de expressões, fizemos uma função que prepara o resultado numa string, (para uma expressão, calcula o seu resultado recursivamente e retorna o valor %counter do resultado da expressão,;para uma variável, o seu @/%; para um literal, o seu valor);
   6. Para implementar o short-circuit nos && e ||, utilizamos a instrução *phi*, que retorna um valor baseado na sua precedência de labels/branches.