Relatório do Projeto de Compiladores

Trabalho realizado por Ricardo Guegan e Samuel Machado (2020211358/2020219391)

Compiladores 2022 – Professor Raul Barbosa

Análise Sintática

Começámos por declarar os *tokens* possíveis de receber a partir da análise lexical, bem como copiar a gramática fornecida do *Juc* para o nosso novo ficheiro jucompiler.y. Nos casos de produções com partes opcionais ou opcionalmente recursivas decidimos criar produções auxiliares para nos ajudarem. Deparámo-nos com a existência de conflitos, sendo que precisámos de os resolver antes de prosseguir.

De seguida, tratámos do tocante ao alerta de erros sintáticos. Consideramos que fizemos uma má gestão do tempo, sendo que nos encontrávamos com menos de 24h para a implementação da AST. Nesse tempo, não conseguimos avançar mais nada em termos de pontos no Mooshak, embora tenhamos tentado aproveitar várias horas desse dia.

Assim, começando a época das metas três e quatro, tínhamos ainda primeiro que desenvolver a AST da meta dois.

Para esta implementação precisámos então de desenhar e definir bem a nossa abordagem antes de começar a escrever código. Decidimos criar uma struct “no” que fosse geral, isto é, que funcionasse para todo o tipo de nós que viríamos a criar. Criámos funções auxiliares que seriam necessárias para associar a cada produção da gramática. São estas:

- **create_new_node**, cria um novo nó com o char* e valores de linha e coluna fornecidos (só passamos estes últimos quando podem vir a ser úteis na análise semântica, caso contrário enviamos -1 e -1)
- **add_new**, associa um nó a outro estando no mesmo nível de profundidade
- **add son**, associa um nó a outro estando em níveis contíguos

Conforme mencionado no parágrafo anterior, começámos a codificar a construção da AST em cada produção, ou, por vezes, apenas manipulações da pilha.

Nesta fase, a parte mais trabalhosa foi tratar dos “Blocks”, entender quando surgiam. Procurando entre muitos dos ficheiros de output fornecidos, compreendemos que estes existiam apenas quando tinham um número de filhos igual a 0 ou superior a 1, e no caso de ter apenas 1 filho, em vez de termos um nó Block com um nó filho, tínhamos apenas o nó que seria filho.

Imprimindo a árvore através da nossa função recursiva **print_arvore**, começámos a comparar os resultados do nosso analisador e descobrindo alguns erros na implementação, sendo que os conseguimos corrigir até alcançar 234 pontos no PosMeta2 do Mooshak, o suficiente para passar à análise semântica.

Análise Semântica

Construção das Tabelas de Símbolos

Começámos por analisar a AST e perceber que produções declaram variáveis ou métodos para fazer a construção e preenchimento das tabelas. Esta é feita em duas etapas, uma que procura os métodos e as variáveis globais e outra que adiciona as variáveis locais no método correspondente. As produções necessárias para a construção das tabelas são:

Primeira travessia da árvore:

- Field Declaration (tipo e o nome da variável)
- Method Header (tipo e o nome do método)
- Param Declaration (acrescentamos à tabela do método)

Segunda travessia da árvore:

- Method Header (para sabermos dentro de que método estamos)
- Var Decl (para acrescentar ao método)

Anotação da AST e Deteção de Erros

A anotação da AST e deteção de erros é feita na segunda travessia, em simultâneo com o preenchimento das tabelas dos métodos. Como apenas é necessário anotar alguns nós e não todos, e como para o fazer é necessário conhecer o tipo dos seus filhos, nestes, fizemos a travessia ascendente da AST. Dividimos os nós a anotar da seguinte forma:

- Algébricas(+ - / * %): Pode ter filhos int e double, se pelo menos um for double é tipo double, caso contrário é int.
- Booleanas (&& || ^): Têm tipo boolean se os seus dois filhos forem booleanos, sendo que o xor também pode ser int se os filhos forem int
- Booleanas de comparação (== >= > < <= !=): Têm tipo boolean se os dois filhos forem do mesmo tipo, sendo que apenas "==" e "!=" aceitam filhos do tipo boolean.
- If/While: Tem um filho "Expression" que tem de ser booleano.
- Shifts (<< >>): Têm tipo int se os seus dois filhos forem do tipo int.
- Parse Args: Tem tipo int se os seus dois filhos forem do tipo int e String[].
- Length: Tem tipo int e o seu filho String[].
- Plus e Minus: É do tipo do filho para filho int ou double.
- Not: Tem tipo boolean se o filho for booleano.
- Assign: Tem dois filhos e o seu tipo depende do filho à esquerda, tirando o caso int e double ambos têm de ser do mesmo tipo.
- Call: Tem o tipo da função chamada e é necessário fazer verificação dos parâmetros.
- DecLit: Tipo int.
- RealLit: Tipo double.
- BoolLit: Tipo boolean.
- StrLit: Tipo String.
- Print: Tem um filho do tipo String.

Caso as condições referidas não sejam satisfeitas o tipo é "undef" (salvo exceções) e levanta-se erro.